Interchangeable
Virtual
Instruments

IVI-3.1: Driver Architecture Specification

December 19, 2022 Edition
Revision 3.8

Important Information

IVI-3.1: Driver Architecture Specification 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

Table of Contents

IVI-3.1 Driver Architecture Specification

IVI Driver Architecture Revision History

This section is an overview of the revision history of the IVI Driver Architecture specification.

Table 1-1. IVI Driver Architecture Specification Revisions

Revision Number	Date of Revision	Revision Notes
Revision 1.0	April 15, 2002	First approved version. Accepted changes; removed draft
Revision 1.1	September 25, 2002	Added changes approved by the IVI Foundation vote of the IVI-3.1 change document. Changes included removal of (1) custom directory copy installations and (2) required use of VISA for serial bus communication.
Revision 1.2	August 22, 2003	Updated spec with changes approved by IVI Foundation vote of IVI-3.1 change document. Changes include (1) modifications to text for multithread safety (both IVI-C and IVI-COM), (2) modified requirements for readme.txt files (both IVI-C and IVI-COM), and (3) renamed section title for pass through functions.
Revision 1.2	October 1, 2004	Editorial revision: IVI-COM drivers do not support multithread locks on sessions.
Revision 1.3	July 15, 2005	Incorporate IVI Conformance working group content regarding use of IVI logo. Change behavior of IVI Shared Component Installer to register with the Windows Add Remove Programs facility.
Revision 1.4	May 30, 2006	Add documentation to the installer requirements to clarify the permissible locations for start menu shortcuts. Incorporate reference to IVI-3.15 LxiSync Specification.
Revision 1.5	October 2, 2006	Remove the following obsolete operating system from target Oss: Win98, WinME, and WinNT4.
Revision 1.5	January 9, 2007	Editorial change: refer users to web site for required service packs. Sections 4.1.1. and 4.2.1.
Revision 1.6	January 11, 2007	Added support for Vista 32 and Vista 64 (32-bit apps) as well as added 64-bit integers to supported data types.
Revision 1.7	April 10, 2007	Add additional requirements for COM drivers packaged with C wrappers. Clarify legal values for the ModulePath attribute in the SoftwareModule entires for IVI Drivers. Editorial change note to section 4.2.1 about porting to other Oss.
Revision 1.7.1	October 22, 2007	• The default IVI standard root directory was changed from `<ProgramFilesDir>\IVI` to `<ProgramFilesDir>\IVI Foundation\IVI`. Defined an IVI data directory. Previously, the Master Configuration Store was assumed to be in the `<IVIStandardRootDir>\Data` directory. For new installations, the IVI data directory is defined to be

Table 1-1. IVI Driver Architecture Specification Revisions

		`<ProgramDataDir>\IVI Foundation\IVI.` • Editorial change to eliminate a potential backward compatibility problem regarding C wrappers on top of IVI-COM drivers (Section 5.15.10.1). • Deprecate Event Server.
Revision 2.0	November 15, 2008	• Add support for 64-bit drivers. • Editorial change to update the IVI Foundation contact information in the Important Information section to remove obsolete address information and refer only to the IVI Foundation web site.
Revision 2.1	February 16, 2009	• Editorial change to clarify issues regarding disambiguating physical identifiers and repeated capability names in the IVI Configuration Store.
Revision 2.2	March 30, 2009	• Editorial change to separate installation content from IVI-3.1 and place it in a separate specification, IVI-3.17
Revision 2.3	February 2010	Editorial changes to add support for Windows 7
Revision 3.0	June 9, 2010	Incorporated IVI.NET
Revision 3.1	October 22, 2010	Editorial change to remove the DLL bitness (Section 5.17.12)
Revision 3.2	November 9, 2010	Added section about copyright notice
Revision 3.2	April 15, 2011	Editorial change – add clarification about throwing derived exceptions from IVI.NET drivers.
Revision 3.2	May 26, 2011	Editorial change to add support for Windows 7 in Section 4.3.2 and to clarify the bitness in sections 4.1.1, 4.2.1, and 5.22.
Revision 3.2	August 25, 2011	Editorial IVI.NET change. Change references to process-wide locking to AppDomain-wide locking.
Revision 3.3	January 18, 2012	Minor changes in Sections 5.9.2 and 5.21 to avoid conflict between physical and virtual names.
Revision 3.3	June 28, 2012	Editorial change in Section 5.16.5, Table 5-10. Reverted the ranges back to their original values.
Revision 3.3	May 7, 2012	Editorial change to remove asterisks from Tables 4.1, 4.2, and 4.3 (Sections 4.1.2, 4.2.2, and 4.3.3)
Revision 3.3	August 6, 2012	Editorial change in Section 5.17.4 to specify implicit rather than explicit implementation for certain interfaces, for consistency with Section 5.17.1.
Revision 3.3	February 7, 2013	Editorial change in Section 5.16.5, Table 5-10 to correct the value of IVI_LXISYNC_ATTR_BASE
Revision 3.4	March 6, 2013	Minor changes: • Updated Sections 2.5 and 3.15 for source code availability • Added Sections 3.13.1, 4.1.11.6, 4.2.9, 4.3.12.6, 5.15.5.1, 5.16.12, 5.17.7.1 for Direct I/O • Added Section 3.16 for functionality covered by IVI drivers

Table 1-1. IVI Driver Architecture Specification Revisions

		• Added new Sections 5.2.2.1.1, 5.2.2.2, 5.2.2.3, and 5.2.2.4, and updated Section 5.23 for testing requirements • Added unsigned 8-bit integer data type in the Table 5-6 • Added section 5.21, Driver Introduction Documentation • Added support for Windows 8
Revision 3.5	October 22, 2013	Minor changes: • Updated Sections 5.17.1, 5.17.4, and 5.17.5 to allow explicit implementation of IiviDriver • Updated Sections 2.7 and 4.4.5.3 to clarify the term "qualified repeated capability identifier" • Added a new "IVI Generation" item in the Compliance Documentation section (Sections 5.23 and 5.23.1) • Updated section 5.14 to allow signed 8-bit integer and array of 8-bit integer values for IVI-C, IVI-COM and IVI.NET • Updated Section 5.12.1 to expand the "Vendor specific errors and warnings" range
Revision 3.5	November 12, 2013	Editorial change to add section 4.3.18, Choosing a Version of the IVI.NET Shared Components for Building the Driver
Revision 3.5	March 28, 2014	Editorial change in Section 4.4.4 to clarify that all repeated capability identifiers within a list of repeated capability identifiers must have the same level of nesting after expansion.
Revision 3.5	January 8, 2015	Editorial changes: • Updates in Section 4.2.7 to remove references to non-existing ivic.h file • Section 4.3.1, added the following sentence, "The .NET Framework Client Profile is not sufficient to meet this condition." • Section 5.17.9, updated AssemblyDescription • Section 5.23, misc updates in the compliance documentation section
Revision 3.5	March 9, 2015	Editorial change in sections 5.15.10, 5.16.14, and 5.17.12 to make the requirement for the help filename format less strict.
Revision 3.5	August 6, 2015	Editorial changes to remove Windows 2000 and add Windows 10 as supported operating system
Revision 3.5	September 24, 2015	Editorial changes in sections 4.2.5 and 5.9.5 to clarify the use of one-based index for C and COM, and zero-based index for .NET for repeated capabilities.
Revision 3.5	October 23, 2015	Editorial change in Section 5.14, Table 5-6 to correct the .NET API type name for an IVI-C or VISA resource descriptor type.
Revision 3.6	June 7, 2016	Minor change to remove support for Windows XP and Windows Vista

Table 1-1. IVI Driver Architecture Specification Revisions

Revision 3.6	October 13, 2016	Editorial changes to add a new section 5.6.1.1: Complementary Attributes and Configure Functions.
Revision 3.7	February 7, 2017	Minor change in Section 5.12.2 to clarify what is expected when underlying I/O software reports an error.
Revision 3.7	May 8, 2017	Editorial changes in Sections 5.14 and 5.16.2 to change all visatype.h references to IviVisaType.h, and to add IviVisaType.h as an appendix to this specification.
Revision 3.8	October 19, 2018	Added ".NET Target Framework Version" to the compliance document.
Revision 3.8	December 19, 2022	Editorial change to add Windows 11 as supported operating system

1. Overview of the IVI Driver Architecture Specification

1.1 Introduction

This section summarizes the *Driver Architecture Specification* itself and contains general information that the reader may need to understand, interpret, and implement aspects of this specification. These aspects include the following:

- Audience of Specification
- IVI Driver Architecture Overview
- References

Terms and acronyms used in this specification are defined in *IVI-5.0: Glossary*.

1.2 Audience of Specification

The intended readers of this document are end users, system integrators, and instrument vendors who are interested in understanding the IVI driver architecture. This document is the starting point to developing and using IVI drivers from both the developer and user standpoint. Therefore, this specification has two primary audiences. The first audience is instrument driver developers who want to implement instrument driver software that is compliant with the IVI Foundation standards. The second audience is instrumentation end users and application programmers who want to implement applications that utilize instrument drivers compliant with this specification. By understanding the IVI driver architecture, end users know how to use an IVI driver and what they can expect when they install one. Similarly, end users can select IVI driver software components that best meet their application needs, based on required and optional behaviors.

1.3 *Organization of Specification*

Section 1.7, *Features and Intended Use of IVI Drivers*, describes the features of IVI drivers from the user perspective. Section 3, *Required and Optional Behavior of IVI Drivers*, provides an instrument driver developer with a high-level understanding of the requirements for creating an IVI driver that implements those features. Section 3 also discusses both the optional and required features and indicates which are required and which are optional. Section *4*, *IVI Driver Architecture*, provides an instrument driver developer with the detailed architecture requirements for developing IVI drivers using the C, COM, and .NET APIs, as well as requirements for handling repeated capabilities. Section *5*, *Conformance Requirements*, contains the precise requirements for IVI drivers. These requirements pertain to the behavior of the drivers as well as the terminology that the drivers use to describe their compliance with the behavioral requirements. *IVI-3.17: Installation Requirements Specification*, provides instrument driver suppliers with installation requirements for IVI drivers.

1.4 IVI Driver Architecture Overview

The IVI Foundation is a group of end-user companies, system integrators, and instrument vendors working together to define standard instrument programming APIs. By defining standard instrument APIs, the IVI Foundation members believe that many of the difficult programming tasks faced by test system developers today, such as instrument interchangeability, execution performance, and simulation, can be solved more easily. This document outlines the basic architecture of IVI drivers and the framework in which they can be used to deliver these benefits. It is important to realize that the IVI Foundation is an organization defining specifications, not products. Many companies will be building products and systems around these specifications. This document specifies the requirements for instrument drivers and the areas that are open for interpretation and implementation strategies.

The IVI Foundation members believe that standard instrument APIs alone do not *guarantee* better performance or instrument interchangeability, but rather form a critical necessary building block that *facilitates* these improvements. This document also specifies the behaviors of IVI instrument drivers and references the required software components that IVI drivers must use.

1.5 Conformance Requirements

This specification provides an IVI driver developer with enough information to write an IVI specific driver by documenting the features that IVI drivers have and their conformance requirements. When appropriate, other specifications are referenced for further detail. IVI drivers can be developed with a COM, ANSI-C, or .NET API. This specification includes general requirements that are applicable to both APIs. When necessary to differentiate between the COM and C APIs, separate requirements are defined.

1.6 References

Several other documents and specifications are related to this specification. These other related documents are the following:

- IVI1: Charter Document
- IVI3.2: Inherent Capabilities Specification
- IVI3.3: Standard Cross Class Capabilities Specification
- IVI3.4: API Style Guide
- IVI3.5: Configuration Server Specification
- IVI3.6: COM Session Factory Specification
- IVI3.8: Locking Component Specification
- IVI3.9: C Shared Components Specification
- IVI3.12: Floating Point Services Specification
- IVI-3.15: IviLxiSync Specification
- IVI-3.17: Installation Requirements Specification
- IVI3.18: IVI.NET Utility Classes and Interfaces Specification
- IVI5.0: Glossary
- IVI Class Specifications
- VPP3.3: Instrument Driver Interactive Developer Interface Specification
- VPP4.3.2: VISA Implementation Specification for Textual Language
- VPP4.3.4: VISA Implementation Specification for COM
- VPP9: Instrument Vendor Abbreviations

1.7 Substitutions

This specification uses paired angle brackets to indicate that the text between the brackets is not the actual text to use, but instead indicates the kind of text that can be used in place of the bracketed text. Sometimes the meaning is self-evident, and no further explanation is given. The following list includes those that may need additional explanation for some readers.

- <ClassName>: The name of an IVI instrument class as defined by an IVI Instrument Class specification. For example, “IviDmm”.
- <ClassType>: The name of an IVI instrument class as defined by an IVI Instrument Class specification, without the leading “Ivi”. For example, “Dmm”.
- <ComponentIdentifier>: For IVI-COM and IVI.NET, the string returned by a specific driver’s Component Identity attribute. This string uniquely identifies the driver. For example, “Agilent34410”.
- <Prefix>: For IVI-C class drivers, the string returned by the driver’s Class Driver Prefix attribute. The class driver prefix will commonly be an IVI class name, but may be different. For example, “IviDmm”.

For IVI-C specific drivers, the string returned by the driver's Specific Driver Prefix attribute. For example, "NI3456"

- <CompanyName>: The name of the driver vendor (not the instrument manufacturer). For example, "Agilent Technologies, Inc".
- <ProgramFilesDir>: The Windows program files directory. This varies across different versions of Windows. In some contexts, it is not intended to differentiate between the 64-bit and 32-bit program files directories found on 64-bit versions of Windows that include Windows On Windows (WOW), but to be understood as a generic reference to the program files directory.
- <ProgramDataDir>: The Windows data directory. This varies across different versions of Windows. It is generally understood to apply to all users.
- <IviStandardRootDir>: The root install directory for the IVI Shared Components, which consists of executables and other files needed to create and run IVI drivers. By default, this directory is "<ProgramFilesDir>\IVI Foundation\IVI".
- <RcName>: The name of a repeated capability. Repeated capabilities may be defined in class specs or by specific driver developers.
- <FwkVerShortName>: The IVI.NET short name for a version of the .NET Framework.

Where it is important to indicate the case of substituted text, casing is indicated by the case of the text between the brackets.

- <ClassName> indicates Pascal casing. For example, "IviDmm".
- <className> indicates camel casing. For example, "iviDmm"
- <classname> indicates all lower case. For example, "ividmm"
- <CLASSNAME> indicates all upper case. For example, "IVIDMM"
- <CLASS_NAME> indicates all upper case with underscores between words. For example, "IVI_DMM".

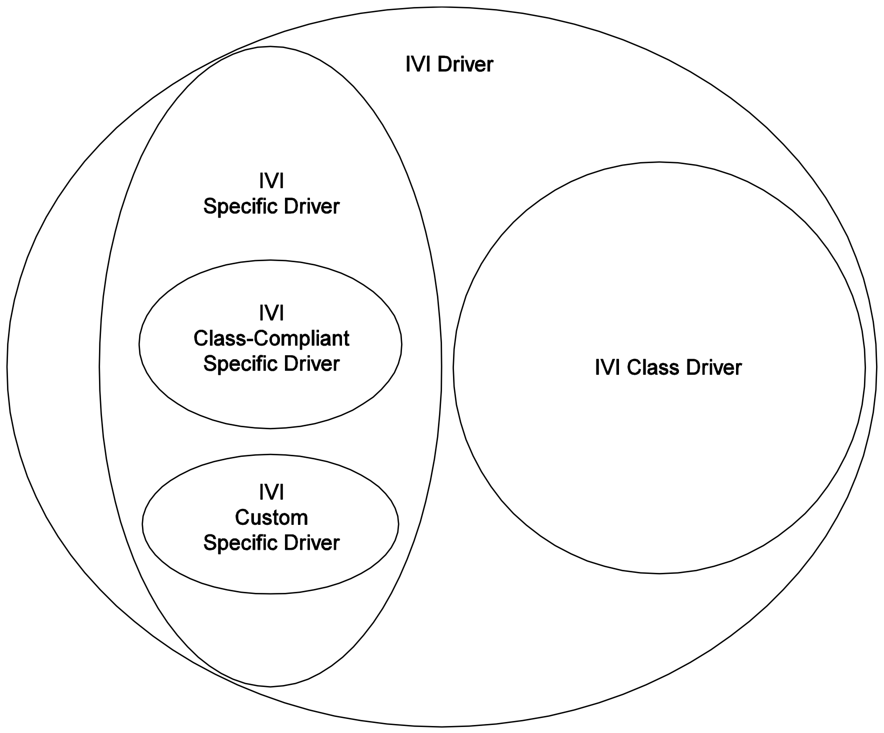
2. Features and Intended Use of IVI Drivers

2.1 Introduction

This section introduces the features and intended use of IVI drivers to test program developers. By providing an overview of the types of IVI drivers available, their architecture, and their features, a user can better understand the benefits and how they can be incorporated into test applications.

2.2 Types of IVI Drivers

As a convenience for readers, this section describes terms that this specification uses to refer to different types of IVI drivers. It is anticipated that these terms will be sufficient to allow end-users to make an informed choice about which type of driver most closely matches their application needs. Formal definitions for these and other terms used in the IVI specifications are included in *IVI-5.0: Glossary*. Figure **2-1** is a Venn diagram depicting the relationship between driver types.


IVI Driver
IVI Specific Driver
IVI Class-Compliant Specific Driver
IVI Custom Specific Driver
IVI Class Driver

Note: When necessary to distinguish between API types, IVI specific drivers are further categorized by replacing "IVI" with "IVI-C", "IVI-COM", or “IVI.NET”.

Figure 2-1. Types of IVI Drivers

IVI Driver

An *IVI driver* is an instrument driver that implements the inherent capabilities detailed in *IVI-3.2: Inherent Capabilities Specification*, regardless of whether the driver complies with a class specification. IVI drivers can communicate directly to the instrument hardware or act as a pass through layer to another IVI driver. An IVI driver is either an IVI specific driver or an IVI class driver.

IVI Specific Driver

An *IVI specific driver* is an IVI driver that contains information for controlling a particular instrument or family of instruments and communicates directly with the instrument hardware. For example, IVI specific drivers control message-based instrument hardware by sending command strings and parsing responses.

IVI Class-compliant Specific Driver

An *IVI class-compliant specific driver* is an IVI specific driver that complies with one of the defined IVI class specifications. For example, an IVI class-compliant specific driver for an oscilloscope exports the API defined by the IviScope class specification. When a driver complies with a particular class specification, the driver is referred to by that class specification name, such as *IviScope Specific Driver* or *IviScope-compliant Specific Driver*. In addition to complying with a standard API for a given instrument class, IVI class-compliant specific drivers also incorporate other features to provide the user with instrument interchangeability. A user should use an IVI class-compliant specific driver when hardware independence is desired.

IVI Custom Specific Driver

An *IVI custom specific driver* is an IVI specific driver that is not compliant with one of the defined IVI class specifications. IVI custom specific drivers cannot be used for hardware interchangeability because they export a custom API. IVI custom specific drivers are typically created for use with specialized instruments, such as an optical attenuator.

IVI Class Driver

An *IVI class driver* is an IVI driver that allows users to interchange instruments when using IVI class-compliant specific drivers. IVI class drivers export an API that complies with one of the defined IVI class specifications. IVI class drivers communicate to instruments through an IVI class-compliant specific driver. For example, an IviScope class driver exposes the functions, attributes, and attribute values defined in the IviScope class specification. An application program makes calls to an IviScope class driver. The IviScope class driver, in turn, makes calls to an IviScope-compliant specific driver that communicates with an oscilloscope. IVI class drivers are necessary for interchangeability when using IVI-C class-compliant specific drivers. IVI class drivers may also communicate to instruments through IVI-COM class-compliant specific drivers.

IVI-C

This document uses the term *IVI-C* in place of *IVI* when referring to IVI drivers that that have a C API. For example, an IVI-C class-compliant specific driver is an IVI specific driver that exports a C API and complies with one of the defined IVI class specifications. IVI-C drivers are distributed on Windows to users in the form of a Win32-DLL. Many commonly used application development environments, such as Agilent VEE, LabVIEW, LabWindows/CVI, and Visual C++ support calling into a C DLL. To achieve interchangeability with an IVI-C specific driver, the user's application program must make calls to an IVI class driver.

IVI-COM

This document uses the term *IVI-COM* in place of *IVI* when referring to IVI drivers that have a COM API. For example, an IVI-COM class-compliant specific driver is an IVI specific driver that exports a COM API that complies with one of the defined IVI class specifications. IVI-COM drivers are distributed on Windows to users in the form of a Win32-DLL. Many commonly used application development environments, such as Agilent VEE, LabVIEW, LabWindows/CVI, and Visual C++ support calling a COM object. To achieve interchangeability with an IVI-COM class-compliant specific driver, the user's application program must make a call to the IVI-COM Session Factory, a software component defined by the IVI Foundation. More details on the use of the IVI-COM Session Factory are included in Section 2.9.2.2, *How Interchangeability Works in COM and .NET.*

IVI.NET

This document uses the term *IVI.NET* in place of *IVI* when referring to IVI drivers that have a .NET API. For example, an IVI.NET class-compliant specific driver is an IVI specific driver that exports a .NET API and complies with one of the defined IVI class specifications. IVI.NET drivers are distributed on Windows to users in the form of a .NET assembly. Many commonly used application development environments, such as C#, VB.NET, LabVIEW, Agilent VEE, and managed Visual C++ support calling into a .NET assembly. To achieve interchangeability with an IVI.NET specific driver, the user's application program must make a call to one of the IVI.NET session factory methods. These factories are defined and implemented by the IVI Foundation. More details on the use of the IVI.NET session factory methods are included in Section 2.9.2.2, *How Interchangeability Works in COM and .NET.*

2.2.1 Specific Driver Wrappers

Some vendors, system integrators, or users may want to develop specific drivers that export a combination of IVI APIs. Such drivers may be implemented using a specific driver with one interface type and one or more wrappers that implement other interface types. For example, if the native interface type of a specific driver is COM, the specific driver developer can create a wrapper that gives the specific driver a C interface, or vice versa. Drivers that export both interfaces comply with IVI-COM requirements and IVI-C requirements, as well as additional requirements for wrappers defined in *IVI-3.2: Inherent Capabilities Specification*.

2.2.2 Custom Class Drivers

Some vendors, system integrators, or users may want to develop custom class drivers, which are class drivers that comply with a class specification developed outside the IVI Foundation and not approved by the IVI Foundation. An IVI custom class driver meets all the requirements of an IVI class driver except it does not comply with a class specification approved by the IVI Foundation.

2.2.3 Special Considerations for IVI Custom Specific Drivers

To address a special market niche, a driver vendor might want to develop an IVI custom specific driver for an instrument that otherwise fits within an instrument class. Vendors are allowed to create IVI custom specific drivers in such cases, but the vendor should also supply an IVI class-compliant specific driver. For example, it would be confusing to users if a general purpose DMM has an IVI custom specific driver but not an available IviDmm-compliant specific driver.

2.3 Functions and Attributes

This document uses the terms *functions*, *attributes*, and *attribute values* to refer to the elements of the API exported by an IVI driver. Unless specified otherwise, functions refer generically to C functions, COM methods, and .NET methods. Similarly, attributes refer to C attributes, COM properties, and .NET properties.

Attributes can be grouped into two categories – *hardware configuration attributes* and *software control attributes*. Generally, each instrument setting is associated with a hardware configuration attribute. Hardware configuration attributes allow the user to set and retrieve values of the associated instrument settings.

Software control attributes control how the instrument driver works rather than representing particular instrument settings. Software control attributes that are common to all IVI drivers are defined in *IVI-3.2: Inherent Capabilities Specification*. For example, *IVI-3.2: Inherent Capabilities Specification* defines software control attributes that allow users to enable and disable features such as range checking and simulation.

2.4 Availability and Installation

Although it is possible for end users to develop their own IVI drivers, it is the intention of the IVI Foundation that users obtain most of their IVI drivers from driver suppliers such as instrument vendors, system integrators, or other software suppliers. These drivers can be downloadable from web sites, shipped with instruments, or distributed with other software applications on a physical storage medium, such as a CD-ROM. Users can expect to have the instrument drivers packaged into an installer and available for deployment on Microsoft Windows. In general, the driver supplier should also distribute or provide links to all the software components that the IVI driver requires, including configuration utilities, IVI Foundation-defined software components, and vendor specific software components.

Users can expect that IVI drivers use the VISA I/O library when communicating over the GPIB and VXIbus interfaces. Driver suppliers are not required to distribute the VISA I/O library. Users might need to install the VISA I/O library separately.

Drivers that communicate over other buses, such as 1394 and PCI, do not have to use the VISA I/O library. If such drivers require I/O libraries that do not come with the operating system, the driver supplier should distribute such libraries with the driver.

The IVI Foundation defines terms to use when requesting drivers. These terms describe levels of compliance and features in the driver. Section 5.23, *Compliance Documentation*, defines the compliance category naming format. It is strongly recommended that all users express their requests in this terminology and avoid using the simple term "IVI driver". For example, if a user wants to develop an interchangeable test program that uses the COM API to DMMs, the user should request an "IVI-COM IviDmm specific driver".

2.5 Source Code Availability

Instrument driver suppliers must make source code available if the source code is a simple translation of the driver calls to a separate publicly documented and officially supported interface and does not include proprietary or confidential content. The compliance document for an IVI specific driver states whether the source code is available and under what conditions.

It is not always practical for instrument driver suppliers to make source code available. Instrument driver suppliers often choose not to distribute source code that contains proprietary algorithms or that is complex to debug. If possible, the driver developer should encapsulate the proprietary or complex software in a support library for which the source code is not available and distribute source code for the remainder of the driver. If the remainder of the driver has little content, then distributing source code has little benefit for the user.

Instrument Driver suppliers who include source code must also provide instructions on rebuilding the driver executables in at least one publicly available development environment.

2.6 Capability Groups

The fundamental goal of IVI drivers is to allow test developers to change the instrumentation hardware on their test systems without changing test program source code, recompiling, or re-linking. To achieve this goal, instrument drivers must have a standard programming interface. Other IVI Foundation specifications define standard functions and attributes for common instrument types such as oscilloscopes, digital multimeters, and function generators. For example, the oscilloscope class contains common attributes for configuring an oscilloscope, such as vertical range and trigger type. The class specification also defines functions for high-level configuration and data retrieval, such as Configure Channel and Read Waveform. Because instruments do not have identical functionality or capability, it is impossible to create a single programming interface that covers all features of all instruments in a class. For this reason, the IVI Foundation recognizes different types of capabilities – *Inherent Capabilities, Base Class Capabilities, Class Extension Capabilities* and *Instrument Specific Capabilities*.

Inherent IVI Capabilities

Inherent IVI capabilities are the functions, attributes, and attribute values that all IVI instrument drivers must implement. Several of the inherent functions are similar to the functions that the VXI*plug&play* Systems Alliance requires. For example, IVI drivers must have Initialize, Reset, Self-Test, and Close functions[1]. Some inherent attributes and functions allow the user to enable and disable performance features, such as state caching, simulation, range checking, and instrument status checking. Other inherent attributes provide information about the driver and the instrument. For instance, users can programmatically retrieve information about specification compliance, driver vendor, and the instrument models that the driver supports. IVI-C and IVI.NET drivers must also have functions to apply multithread locks to sessions (C) or driver instances (.NET). For a detailed explanation of inherent IVI capabilities, refer to *IVI-3.2: Inherent Capabilities Specification*.

[1] In IVI.NET, the Initialize function is replaced by the driver constructor(s).

Base Class Capabilities

Base class capabilities are the functions, attributes, and attribute values of an instrument class that are common across most of the instruments available in the class. The goal of the IVI Foundation is to support 95% of the instruments in a particular class. Decisions regarding base class capabilities are made through consensus, based on the most popular instruments and the most commonly used functions of those instruments in automated test systems. Refer to *IVI-1: Charter Document* for more details. For an IVI class-compliant specific driver to be compliant with a class, it must implement all the base capabilities.

For example, the base class capabilities of the oscilloscope class have functions and attributes that configure an edge-triggered acquisition, initiate an acquisition, and return the acquired waveform.

For a complete description of the base capabilities for a particular class, refer to the individual class specifications, such as *IVI-4.2: IviDmm Class Specification*.

Class Extension Capabilities

Class extension capabilities are groups of functions, attributes, and attribute values that represent more specialized features of an instrument class. In general, IVI interchangeable specific instrument drivers are not required to implement extension groups. For example, although all oscilloscopes have very similar base class capabilities for vertical and horizontal settings, there is a wide variety of trigger modes among oscilloscopes. The IviScope class specification has extensions for different trigger modes, such as TV trigger, runt trigger, width trigger, and so on. The driver for an oscilloscope that can perform TV triggering implements the TV trigger extension group. The driver for an oscilloscope that cannot perform TV triggering does not implement the TV trigger extension group but is compliant with the IviScope class because it implements the IviScope base capabilities group. If an application depends on a function from one of the extension capability groups, the application must restrict itself to drivers that implement the capability group.

Generally, an IVI class-compliant specific driver implements all class extensions the instrument hardware supports. It would be confusing to users if an instrument had certain hardware capabilities that fit into a class extension but the IVI class-compliant specific driver did not implement that class extension.

For a complete description of the class extension capabilities for a particular class, refer to the individual class specifications, such as *IVI-4.2: IviDmm Class Specification*.

Instrument Specific Capabilities

Most instruments that fit into a class also have features that are not defined by the class. Instrument specific capabilities are the functions, attributes, and attribute values that represent those features. For example, some oscilloscopes have special features such jitter and timing analysis that are not defined in the IviScope class specification. The functions, attributes, and attribute values necessary to access the jitter and timing analysis capabilities of the oscilloscope are considered instrument specific capabilities. The IVI Foundation allows for instrument specific features in IVI drivers. In fact, driver developers are encouraged to implement instrument specific features in their drivers. However, the user of an application that accesses instrument specific features of a driver cannot swap in a different instrument without modifying the instrument specific portions of the test program.

Figure **2-2** illustrates how an IVI class-compliant specific driver is divided into various capability groups.

IVI Foundation Defined Capabilities						Instrument Specific Capabilities
IVI Inherent Capabilities	Class-Defined Capabilities					
	Base Class Capabilities	Class Extension Capabilities				
		Ext. #1	Ext. #2	. . .	Ext. #*n*	

Figure 2-2. IVI Class-Compliant Specific Driver

IVI custom specific drivers contain only IVI inherent capabilities and instrument specific capabilities. Figure **2-3** shows the capability groups of an IVI custom specific driver.

IVI Foundation Defined Capabilities	Instrument Specific Capabilities
Inherent IVI Capabilities	

Figure 2-3. IVI Custom Specific Driver

2.7 Repeated Capabilities

Many instruments contain multiple instances of the same type of functionality. For example, many instruments have multiple channels with independent settings. The general term for functionality that is duplicated in an instrument is *repeated capability*.

Repeated capabilities can be complex. An instrument may have multiple sets of repeated capabilities, such as channels and traces, or analog channels and digital channels. Also, repeated capabilities may be nested within other repeated capabilities, for example traces within displays. Furthermore, when working with repeated capabilities that have a large number of instances, such as digital channels, the user may find it convenient to specify a set of instances when calling an IVI driver API.

The IVI Foundation specifies how IVI driver APIs allow the user to access repeated capabilities, including nested repeated capabilities and sets of repeated capability instances. Each IVI class specification specifies the repeated capabilities, if any, that the class-defined APIs export and any nesting of the repeated capabilities. Each specification also defines the functions and attributes to which the repeated capabilities apply and which functions and attributes accept sets of repeated capability instances.

A repeated capability is designated by a *repeated capability name*. The IVI class specifications define repeated capability names for class-defined repeated capabilities. For example, the IviScope class defines "Channel" as a repeated capability name. IVI specific drivers define repeated capability names for instrument specific repeated capabilities. A repeated capability defined in an IVI class specification is also designated by a *qualified repeated capability name*, which is constructed by appending the repeated capability name to the class name specified in the IVI class specification with which the driver complies. In the example above, the qualified repeated capability name is "IviScopeChannel". Similarly, a qualified repeated capability name can also be constructed for a repeated capability defined by a custom class.

An instance of a repeated capability is designated by a *repeated capability identifier*. An IVI specific driver defines a *physical repeated capability identifier* for each repeated capability instance it implements. For example, a driver might define "1" and "2" as the identifiers for a two-channel scope, whereas another driver might define "CH0" and "CH1".

Users can define *virtual repeated capability identifiers* and map them to physical identifiers. For example, the user might define "Acquisition1" and "Acquisition2" as virtual identifiers and map them to the physical identifiers defined by the particular specific driver in use. When calling IVI driver APIs, users can specify repeated capability instances using physical identifiers or virtual identifiers.

2.8 Declaring Conformance, Capabilities, and Requirements

Users learn about the conformance and capabilities of an IVI driver by inspecting the compliance specification section of the instrument driver help documentation. The compliance documentation includes such information as the driver type, bus interface type, instrument models supported, driver version, driver bitness, and supported operating systems. Furthermore, if the IVI driver is compliant with a class specification, the compliance document also includes such information as class specification prefix, class specification version, and extension capabilities supported. The compliance document also specifies the minimum version of support software with which the driver is compatible. For example, if an IVI specific driver uses VISA as the I/O interface, the compliance documentation states the minimum version of VISA that the driver requires.

For a complete description of the compliance documentation that is supplied with a driver, refer to Section 5.23, *Compliance Documentation*.

2.9 Using IVI Instrument Drivers

As with traditional instrument drivers, IVI specific drivers contain the information for controlling a particular instrument model, including the command strings, parsing code, and parameter validation for the particular instrument. However, to develop an application program that is not dependent on the specific driver API, a generic API is necessary. Depending on the type of programming interface used in a program, abstraction is obtained differently.

2.9.1 Using IVI Drivers from the Instrument Specific Perspective

An application program can directly access an IVI specific driver. This is similar to the way in which traditional instrument drivers are used. As with traditional drivers, the IVI specific driver encapsulates the information necessary to control a particular instrument model. With this approach, a user gains such IVI benefits as simulation, configurable status checking, and multithread safety, but not interchangeability.

The user of an application that accesses instrument specific features of a driver cannot swap in a different instrument without modifying the instrument specific portions of the test program.

Even if the developer of an application restricts the use of the driver to the functions and attributes that are defined in a class specification, the user still cannot swap in a different instrument without modifying the test program source code. The reasons for this are different for IVI-COM, IVI-C, and IVI.NET drivers.

An application that directly opens a session to the class-compliant API of an IVI-COM class-compliant specific driver must explicitly identify the specific driver in the program. It does so by specifying a Class ID or Prog ID that the COM libraries use to activate the appropriate object.

Each IVI-C specific driver exports functions and attributes that begin with a prefix that uniquely identifies the IVI-C specific driver. An application program that directly references functions of an IVI-C specific driver contains these unique prefixes.

An application that directly opens a session to the class-compliant API of an IVI.NET class-compliant specific driver must directly instantiate the specific driver. This can be done either by adding a reference to the assembly references or by using reflection and late binding.

2.9.2 Using IVI Drivers to Achieve Interchangeability

To achieve interchangeability without recompiling or re-linking, the user must use other components in addition to the class-compliant specific driver. The user must program to a class API rather than to a specific driver. The user must also identify which specific driver and hardware resource to use without directly referencing them in the program. This requires an external configuration store and the ability to dynamically load the specific driver based on information in the configuration store.

2.9.2.1 Accessing Class and Specific APIs

An application program can access both class-compliant and instrument specific features in the same IVI driver without reinitializing. This allows the user to develop an application in which most of calls to instrument drivers are interchangeable but that accesses instrument specific features when necessary. Swapping an instrument in this case might require recompiling and re-linking the application. Nevertheless, the user benefits from this approach by minimizing the instrument specific code in the program.

2.9.2.2 How Interchangeability Works in COM and .NET

IVI-COM and IVI.NET class-compliant specific drivers contain class compliant APIs. For every IVI-COM or IVI.NET specific instrument driver within a particular class, the class-compliant APIs are identical. Application developers program to these class-compliant APIs.

The presence of a class-compliant API in an IVI-COM or IVI.NET specific driver is not enough to achieve interchangeability without re-linking because instantiating a driver through the class compliant API directly requires specifying a Class ID or Prog ID (IVI-COM) or a driver reference (IVI.NET) for the driver. To overcome this limitation, the IVI Foundation recommends using the IVI-COM Session Factory for IVI-COM or one of the IVI.NET session factory methods for IVI.NET. These are special components that can dynamically load an IVI specific driver without requiring the application program to identify the IVI specific driver directly. An application program identifies a logical name that must match a logical name in the IVI configuration store. In the IVI configuration store, the logical name refers to an IVI specific driver and a specific physical instrument. Figure **2-4** shows how a user achieves interchangeability when using IVI-COM class-compliant specific driver. The scenario for .NET is similar.

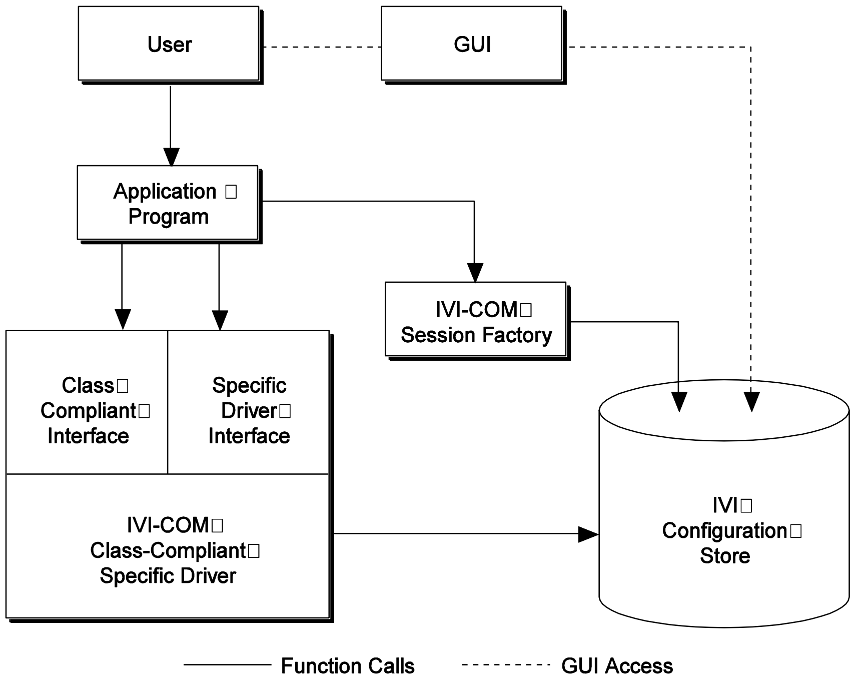


Figure 2-4. Using an IVI-COM Class Compliant Specific Driver

The IVI-COM class-compliant API is the same for all IVI-COM class-compliant specific drivers for a particular class, and the IVI.NET class-compliant API is the same for all IVI.NET class-compliant specific drivers for a particular class. Because of this, the driver exposes all class-defined methods and properties, even those that it does not implement. The methods and properties that the driver exposes but does not implement return an error.

In addition to class-compliant APIs, IVI-COM and IVI.NET class-compliant specific drivers can contain instrument specific APIs. In COM, an application program that uses the class-compliant interface can call into the instrument specific API by calling Query Interface either directly or implicitly as in Visual Basic. In .NET, an application program that uses the class-compliant interface can call into the instrument specific API by using IServiceProvider.GetService() to obtain a reference to the instrument specific API.

2.9.2.3 How Interchangeability Works in C

IVI-C class-compliant specific drivers export the functions, attributes, and attribute values of the class specification for the class capabilities that they implement.

The presence of class defined functions in an IVI-C class-compliant specific driver is not enough to achieve interchangeability without re-linking because the driver prefixes differ from one IVI-C class-compliant specific driver to another. To overcome this limitation, application developers program to an IVI class driver API.

The IVI class driver exports the inherent capabilities, the base class capabilities, and all class extension capabilities. The IVI class driver does not actually *implement* any of the capabilities except for a few inherent capabilities specific to class drivers. Instead, the IVI class driver is a pass-through layer to the IVI class-compliant specific driver. The IVI class driver dynamically loads the specific driver and connects the inherent and class-defined functions and attributes in the IVI-C class-compliant specific driver to the corresponding functions and attributes in the IVI class driver.

When the application program calls the Initialize function of the IVI class driver, it specifies a logical name that must match a logical name in the IVI configuration store. An application program that uses the IVI class driver can also call the IVI-C class-compliant specific driver by calling the Get Specific Driver C Handle function to obtain the handle to the specific driver session. Figure **2-5** shows how a user achieves interchangeability when using IVI-C class-compliant specific drivers.

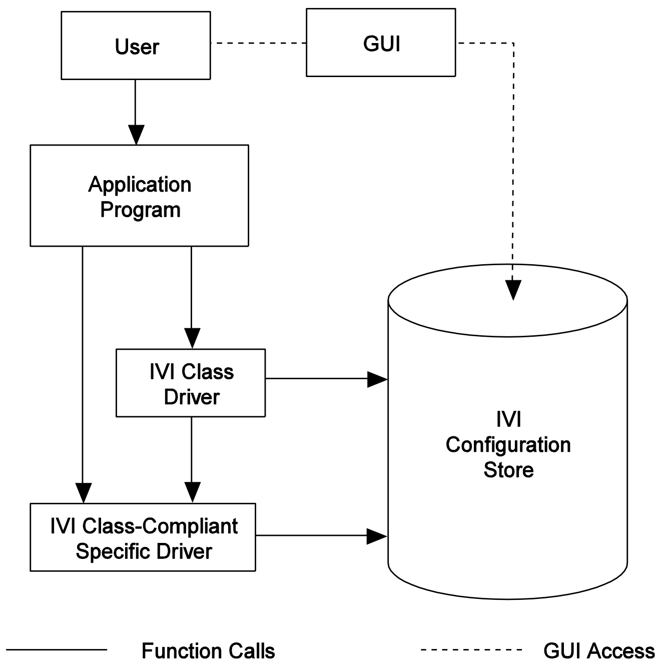


Figure 2-5. Using and IVI-C Class Compliant Specific Driver

2.9.2.4 Interchanging IVI-C, IVI-COM, and IVI.NET Specific Drivers

Table 2-1 shows that two basic mechanisms can be used to interchange drivers.

First, a wrapper that uses one API can be provided for a driver that is implemented with another API. For example, if an IVI-COM specific driver has an IVI-C wrapper, a program that uses an IVI-C class driver can direct the class driver to load the IVI-C wrapper, and use the IVI-COM driver via the wrapper.

Second, IVI-C class drivers can be written so that they can call any type of IVI API. For example, an IVI-C class driver can be written that knows how to call IVI-COM drivers.

Table 2-1. Mechanisms for Interchangeability

Driver Type:	Test program uses this class compliant interface:		
	IVI-C	IVI-COM	IVI.NET
IVI-C class driver	NA	IVI-COM wrapper on the IVI-C driver	IVI.NET wrapper on the IVI-C driver
IVI-COM	(1) IVI-C wrapper on the IVI-COM driver (2) IVI-C class driver calls IVI-COM	NA	IVI.NET wrapper on the IVI-COM driver
IVI.NET	(1) IVI-C wrapper on the IVI.NET driver (2) IVI-C class driver calls IVI.NET	IVI-COM wrapper on the IVI.NET driver	NA

2.10 The IVI Configuration Store

To allow users to swap out instruments without code modifications, application programs refer to *logical names*, rather than to specific instruments and drivers. A logical name refers to an *IVI driver session configuration*. An IVI driver session configuration, in turn, identifies an IVI specific driver and an instrument to use when opening an IVI session. The IVI driver session configuration also specifies settings for configurable features such as simulation, range checking, and state caching. The IVI configuration store contains logical names and IVI driver session configurations. When a user swaps an instrument, the user redirects the logical name in the IVI configuration store to refer to a different IVI driver session configuration. IVI class drivers and the IVI-COM Session Factory and IVI.NET session factory methods read the logical name and IVI driver session configuration information from the common IVI configuration store.

For ease of use, instrument driver suppliers can provide an IVI configuration utility to assist users in modifying the IVI configuration store. The IVI Foundation does not define the IVI configuration utility. Therefore, it is likely that multiple IVI configuration utilities will be available.

For IVI-COM, the IVI Foundation provides a component, the IVI-COM Session Factory, that should be used regardless of the type of IVI-COM specific driver that is being instantiated. This component is described in more detail in *IVI-3.6: COM Session Factory Specification*, and is installed with the IVI Shared Components.

For IVI.NET, the IVI Foundation provides a session factory method for each instrument class, designed for use with IVI.NET class compliant specific drivers that support the class. These are documented in the class capabilities section of each instrument class specification. The IVI Foundation also provides a session factory method that may be used with any driver, including IVI.NET generic specific drivers. This generic session factory is described in section 4.1.3, *IVI.NET IviDriver Session Factory*, of *IVI-3.2: Inherent Capabilities Specification*. Note that for IVI-COM, the Session Factory returns an object reference which the user must cast to the desired class compliant type. For IVI.NET, session factory methods perform the additional step of verifying that the driver supports the specified instrument class, and IVI.NET session factory methods return either the instrument class base interface type (for example, IIviDmm) or the inherent capabilities base interface (IiviDriver).

2.11 Other Considerations for Interchangeability

The previous section discussed the general framework for interchangeability. This section discusses the features that are necessary to assist the user in achieving interchangeability. This section also discusses circumstances in which interchangeability cannot be achieved.

2.11.1 Virtual Names for Channels and Other Repeated Capabilities

Just as users should not reference specific driver APIs directly in their applications, users striving for interchangeability should also refrain from using the physical repeated capability identifiers that specific drivers define. To refer to instrument channels or other repeated capability instances, the user should define virtual repeated capability identifiers in the IVI driver session configuration in the IVI configuration store, map the virtual identifiers to physical identifiers, and use only the virtual identifiers in the application source code.

2.11.2 Configurable Initial Settings

Most instruments have unique features that are not defined by IVI class specifications. Sometimes, an instrument specific feature has an effect on the class-defined capabilities of the instrument, depending on the value of the settings associated with the feature. An initial value for an instrument specific setting might cause the instrument to behave in a non-interchangeable manner. Therefore, to achieve interchangeability, it might be necessary to specify a value for an instrument specific attribute.

On the other hand, a completely interchangeable program must refrain from accessing instrument specific attributes. To resolve this dilemma, an IVI specific driver can provide the user with the ability to specify values for instrument specific attributes in the IVI driver session configuration in the IVI configuration store. The IVI specific driver applies the values during initialization. The set of values that the user specifies for the session is called the *configurable initial settings*.

The following are examples of situations where applying the configurable initial settings aids the user in achieving interchangeability:

Achieving Interchangeability in the Presence of Additional Capabilities

Applying configurable initial settings might be necessary when swapping a two-channel scope for a four-channel scope in an application program that uses only two channels. If the initial state of the four-channel scope enables all channels and the maximum number of points that the scope acquires depends on the number of channels enabled, then it is possible that the user cannot acquire the same number of points as when using the two-channel scope. If the user disables the additional channels, the driver is more likely to provide the requested record length. Disabling the additional two channels in the program source code would limit interchangeability. Instead, the driver should allow the user to disable the additional two channels in the IVI configuration store.

Achieving Interchangeability With Different Kinds of Switch Channels

A configuration channel is a switch channel that the application does not intend to use as end points of a path in the Connect and Set Path functions. If the user declares a channel as a configuration channel, the driver can use the channel to create more routes through the switch module. This increases the number of possible paths. Since the overall topology and numbers of channels differs among switch modules, the driver should allow the user to declare configuration channels IVI configuration store instead of in the program source code.

2.11.3 Interchangeability Checking

To aid users in developing interchangeable applications, IVI drivers may implement a feature called interchangeability checking. When enabled, this feature identifies cases where an application program is in danger of producing a different result when used with a different instrument. The feature generates warnings when an application program does not fully configure the state of the instrument before initiating a measurement operation. Interchangeability checking is most useful during debugging. Once application development is complete, this feature can be disabled.

Interchangeability checking can also aid in detecting potential interchangeability issues between test modules. When developing a complex test system that consists of multiple test modules, it is generally a good idea to design the test modules so that they can run in any order. To do so requires ensuring that each test module completely configures the state of each instrument it uses. If a particular test module does not completely configure the state of an instrument, the state of the instrument depends on the configuration from a previously executed test module. If the test modules execute in a different order, the behavior of the instrument and therefore the entire test module is likely to change. This change in behavior is generally instrument specific and represents an interchangeability problem. To avoid this problem, users can reset interchangeability checking at the beginning of a test module. An interchangeability warning that occurs after the user resets interchangeability checking indicates that the test module did not completely configure the instrument.

2.11.4 Coercion and Coercion Recording

IVI class specifications often allow a continuous range of values for real-valued parameters and attributes. This is true even if some instruments in the class implement only a discrete set of values for the setting. Some instruments that implement only a discrete set of values accept a continuous range of values and coerce user-specified values to the discrete set. Others accept only the discrete set, in which case the specific driver accepts a continuous range and coerces the user-specified value to a discrete value that the instrument accepts.

For each attribute, the IVI class specifications specify how specific drivers coerce the value the user requests. The specifications can provide a recommendation for how class-compliant specific drivers should coerce values. If an instrument performs coercion in a manner different than what the IVI class specifies, the specific driver must ensure that the final value of the attribute on the instrument is sufficient to meet the user's request. For example, if a user specifies a range of 10.01 volts for a measurement, the instrument might coerce this value to 10.0—even though that is less than the user requested value—because the instrument can measure up to 11.0 volts when in the 10.0 volt range.

To aid users in developing interchangeable applications, IVI drivers may implement a feature called *coercion recording*. This feature helps the user discover when the specific driver coerces values that the user requests. Coercion recording applies to numeric scalar attribute values. If the user enables coercion recording, the IVI specific driver maintains a record of each user value that it coerces. The IVI specific driver exports functions for retrieving the coercion records.

2.11.5 Limitations to Instrument Interchangeability Using IVI Drivers

IVI class-compliant drivers do not guarantee interchangeable behavior. IVI drivers make it possible in software to interchange instruments that have interchangeable behavior in the underlying hardware.

While instruments can have the same settings, the range of values they accept can differ. For example, if an application program configures a DMM to use a 100V range, a user cannot swap the DMM with another instrument that only accepts up to a 10V range.

Furthermore, instruments can have the same settings and ranges but still not produce the same result because they have different measurement techniques or algorithms. For example, oscilloscopes use different algorithms to calculate the reference levels for rise-time measurements.

2.12 Leveraging Syntactic Similarities

Users who do not take advantage of IVI class drivers or IVI class-compliant APIs still benefit from a standard programming interface. All IVI class-compliant specific drivers share a subset of functions and attributes. Furthermore, the values passed to the configuration functions are also standardized. For example, the output amplitude for all IviFgen-compliant drivers is set in Volts peak to peak, not Volts RMS or Volts peak. This commonality between instrument drivers results in reduced learning time and faster development time when working with new hardware and instrument drivers.

2.13 Instrument Driver Operational Modes: Simulation, Debug, and Run-time

IVI specific drivers have several features that enable users to run their applications in different modes. The user can choose modes that are optimal for production and development. For example, during development a user might enable instrument status checking and range checking to assist in analyzing and debugging. Also, instrument hardware is often unavailable during application development. A user can enable simulation to proceed with application development without the hardware.

2.13.1 Range Checking

If range checking is enabled, an IVI specific driver checks that input parameters are within the valid range for the instrument. Range checking is most useful during debugging. After users validate their programs, they can disable range checking to maximize performance.

2.13.2 Instrument Status Checking

If instrument status checking is enabled, an IVI specific driver automatically checks the status of the instrument after most operations. If the instrument indicates that it has an error, the driver returns a special error code. The user then calls the Error Query function to retrieve the instrument specific error code from the instrument.

Instrument status checking is most useful during debugging. Once application development is complete, this feature can be disabled to maximize performance.

2.13.3 Simulation

If simulation is enabled, an IVI specific driver does not perform instrument I/O, and the driver creates simulated data for output parameters. This allows the user to execute instrument driver calls in the application program even though the instrument is not available.

IVI specific drivers perform range checking in simulation mode although not necessarily to the same extent that they do when simulation is disabled. The output values that IVI drivers generate in simulation mode are typically very simple.

IVI drivers may also implement more sophisticated simulation with user-configurable output values and status codes. For example, allowing the user to inject simulated errors for status values in an application program helps the user verify that the program properly handles errors.

2.13.4 State Caching

To minimize the number of I/O calls needed to configure an instrument to a new state, IVI specific drivers may implement state caching. IVI specific drivers can choose to implement state caching for all, some, or none of the instrument settings. If the user enables state caching and the IVI specific driver implements caching for hardware configuration attributes, driver functions perform instrument I/O when the current state of the instrument settings is different from what the user requests. This can result in improved performance as compared to test programs written with instrument drivers that do not implement state caching. For example, if an application program performs a simple frequency sweep of an excitation signal, it is inefficient to resend amplitude, waveform shape, phase, and other types of signal information over and over again. With state caching enabled, only the changed frequency settings are sent to the instrument. State caching allows users to write test programs that completely configure an instrument while minimizing performance degradation caused by redundant I/O.

In general, users should use the same setting for the state caching option in debug and runtime mode.

2.14 Multithread Safety

IVI drivers are multithread safe. Multithread safety means that multiple threads in the same process can use the same IVI driver session and that different sessions of the same IVI driver can run simultaneously on different threads.

To access a driver session from multiple threads, the application initializes the driver in one thread and then shares the session handle or object with other threads. If an application wants to treat several calls to an IVI driver as a single operation that other threads must not interfere with, the application must block other threads during the sequence of calls. The application can do this by using synchronization functions provided by the operating system or programming environment. For IVI-C and IVI.NET, the application can use the locking function that IVI-C and IVI.NET drivers provide.

Some of the scenarios in which users can take advantage of multithreaded access to IVI drivers are the following.

- Two threads each run tests on separate test heads, where the test heads share one or more instruments.
- A transmit test and a receive test on the same phone run in parallel in different threads of the same process.
- Multiple threads use the same switch card in a test station.
- Two threads use different channels of a two-channel power meter. The first thread monitors a test level and adjusts it to maintain a very constant level. The second monitors the UUT's output power. The first thread runs as fast as it can while the second thread takes measurements every 30 seconds.

IVI-C and IVI-COM drivers do not provide a degree of multithread safety that allows multiple processes to share the same session. IVI-COM and IVI-C drivers also do not provide any mechanism to synchronize between multiple threads or processes that open multiple sessions on the same physical instrument. To synchronize access to the same physical instrument from multiple processes, applications that use IVI-C or IVI-COM drivers must use resource locking.

IVI.NET drivers, on the other hand, can synchronize access between threads in the same process or between threads in different processes on the same computer. IVI.NET drivers can also synchronize access between drivers that open multiple sessions to an instrument. These extended locking capabilities available with IVI.NET drivers are discussed further in Section 4.3.11, *Multithread Locking*.

2.15 Resource Locking

The IVI Foundation has not defined the requirements for Resource Locking, or for managing simultaneous access from other users.

2.16 Operating Systems

IVI drivers work on one or more of the following Microsoft operating systems: Windows 7 (32-bit), Windows 7 (64-bit), Windows 8 (32-bit), Windows 8 (64-bit), Windows 10 (32-bit), Windows 10 (64-bit), and Windows 11. An IVI driver supplier lists the supported operating systems in the compliance document.

In the context of all IVI specifications, “Windows 8”, “Windows 10”, and “Windows 11” refer to the versions of Windows operating system that run on x86 and x86-64 compatible CPUs and support the full Win32 API. Windows on ARM is not officially supported at this time.

IVI class specifications are operating system independent. Installation and deployment are operating system dependent. The IVI Foundation recognizes that Microsoft Windows is the most commonly used operating system and that both 32-bit and 64-bit versions of Windows exist. Therefore, the specifications define how to create and deploy drivers on the 32-bit and 64-bit editions of the Microsoft Windows operating systems listed above. Nevertheless, specifications do not preclude driver suppliers from deploying on other operating systems. It should be noted the IVI Foundation might, in the future, define deployment requirements for other operating systems. If so, the specifications might conflict with implementations that predate the standards.

Since Windows 7, 8, and 10 are available in 32-bit and 64-bit versions, this specification will append the bitness in parentheses. For example, the 32-bit version of Windows 10 will be identified as “Windows 10 (32-bit)”. Since Windows 11 is only available in 64-bit versions, the bitness is not appended to Windows 11.

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

2.17 Target Application Development Environments

The IVI Foundation has identified application development environments (ADEs) that it wants to ensure that IVI drivers run well on. Those ADEs are Agilent VEE, National Instruments LabVIEW, National Instruments LabWindows/CVI, MathWorks MATLAB, Microsoft Visual Basic 6.0, Microsoft Visual Basic .NET, Microsoft Visual C#, and Microsoft Visual C++.

2.18 Bitness Considerations

The 32-bit versions of Windows (Windows 7 (32-bit), Windows 8 (32-bit), and Windows 10 (32-bit)) can run only 32-bit applications. Windows 7 (64-bit), Windows 8 (64-bit), Windows 10 (64-bit), and Windows 11 can run both 32-bit and 64-bit applications.

Whereas 32-bit applications and drivers can be installed on both 32-bit and 64-bit versions of Windows, 64-bit applications and drivers can be installed only on 64-bit versions of Windows.

With regard to IVI drivers, vendors, system integrators, or users may develop and distribute only a 32-bit driver, only a 64-bit driver, or both a 32-bit and 64-bit driver. Users need to install drivers with the correct bitness for their application needs. Since users can run 32-bit and 64-bit applications on the same machine under Windows 7 (64-bit), Windows 8 (64-bit), Windows 10 (64-bit), and Windows 11 users might need 32-bit and 64-bit drivers on the same machine.

The IVI Foundation allows the 32-bit and 64-bit versions of the same driver to be installed on the same machine, but only if they are from the same vendor and are the same revision. When the 32-bit and 64-bit versions of the same driver are installed, they share the entry that describes the driver in the IVI configuration store. This is so that users can use the same configuration store entries regardless of whether the applications they run are 32-bit or 64-bit.

Some compilers, such as Microsoft Visual C++, allow users to build both 32-bit and 64-bit versions of an application on the same machine under 32-bit and 64-bit versions of Windows. The IVI Foundation wants to enable users to take advantage of this capability in applications that use IVI drivers. Therefore, vendors who distribute both 32-bit and 64-bit versions of a driver are required, when users install the 32-bit driver on a 32-bit version of Windows, to install the components of the 64-bit driver that are necessary to compile 64-bit applications.

The compliance document for an IVI driver states whether the driver is available in a 32-bit version, a 64-bit version, or both.

3. Required and Optional Behavior of IVI Drivers

3.1 Introduction

Section 1.7, *Substitutions*

This specification uses paired angle brackets to indicate that the text between the brackets is not the actual text to use, but instead indicates the kind of text that can be used in place of the bracketed text. Sometimes the meaning is self-evident, and no further explanation is given. The following list includes those that may need additional explanation for some readers.

- <ClassName>: The name of an IVI instrument class as defined by an IVI Instrument Class specification. For example, “IviDmm”.
- <ClassType>: The name of an IVI instrument class as defined by an IVI Instrument Class specification, without the leading “Ivi”. For example, “Dmm”.
- <ComponentIdentifier>: For IVI-COM and IVI.NET, the string returned by a specific driver’s Component Identity attribute. This string uniquely identifies the driver. For example, “Agilent34410”.
- <Prefix>: For IVI-C class drivers, the string returned by the driver’s Class Driver Prefix attribute. The class driver prefix will commonly be an IVI class name, but may be different. For example, “IviDmm”. For IVI-C specific drivers, the string returned by the driver’s Specific Driver Prefix attribute. For example, “NI3456”
- <CompanyName>: The name of the driver vendor (not the instrument manufacturer). For example, “Agilent Technologies, Inc”.
- <ProgramFilesDir>: The Windows program files directory. This varies across different versions of Windows. In some contexts, it is not intended to differentiate between the 64-bit and 32-bit program files directories found on 64-bit versions of Windows that include Windows On Windows (WOW), but to be understood as a generic reference to the program files directory.
- <ProgramDataDir>: The Windows data directory. This varies across different versions of Windows. It is generally understood to apply to all users.
- <IviStandardRootDir>: The root install directory for the IVI Shared Components, which consists of executables and other files needed to create and run IVI drivers. By default, this directory is “<ProgramFilesDir>\IVI Foundation\IVI”.
- <RcName>: The name of a repeated capability. Repeated capabilities may be defined in class specs or by specific driver developers.
- <FwkVerShortName>: The IVI.NET short name for a version of the .NET Framework.

Where it is important to indicate the case of substituted text, casing is indicated by the case of the text between the brackets.

- <ClassName> indicates Pascal casing. For example, “IviDmm”.
- <className> indicates camel casing. For example, “iviDmm”
- <classname> indicates all lower case. For example, “ividmm”
- <CLASSNAME> indicates all upper case. For example, “IVIDMM”
- <CLASS_NAME> indicates all upper case with underscores between words. For example, “IVI_DMM”.

Features and Intended Use of IVI Drivers, describes the features of IVI drivers from the user perspective. This section provides an instrument driver developer with a high-level understanding of the requirements for creating an IVI driver that implements those features. This section assumes that the reader is familiar with the contents of Section 1.7.

IVI instrument drivers are not required to support all features described in Section 1.7, *Substitutions*

This specification uses paired angle brackets to indicate that the text between the brackets is not the actual text to use, but instead indicates the kind of text that can be used in place of the bracketed text. Sometimes the meaning is self-evident, and no further explanation is given. The following list includes those that may need additional explanation for some readers.

- <ClassName>: The name of an IVI instrument class as defined by an IVI Instrument Class specification. For example, "IviDmm".
- <ClassType>: The name of an IVI instrument class as defined by an IVI Instrument Class specification, without the leading "Ivi". For example, "Dmm".
- <ComponentIdentifier>: For IVI-COM and IVI.NET, the string returned by a specific driver's Component Identity attribute. This string uniquely identifies the driver. For example, "Agilent34410".
- <Prefix>: For IVI-C class drivers, the string returned by the driver's Class Driver Prefix attribute. The class driver prefix will commonly be an IVI class name, but may be different. For example, "IviDmm". For IVI-C specific drivers, the string returned by the driver's Specific Driver Prefix attribute. For example, "NI3456"
- <CompanyName>: The name of the driver vendor (not the instrument manufacturer). For example, "Agilent Technologies, Inc".
- <ProgramFilesDir>: The Windows program files directory. This varies across different versions of Windows. In some contexts, it is not intended to differentiate between the 64-bit and 32-bit program files directories found on 64-bit versions of Windows that include Windows On Windows (WOW), but to be understood as a generic reference to the program files directory.
- <ProgramDataDir>: The Windows data directory. This varies across different versions of Windows. It is generally understood to apply to all users.
- <IviStandardRootDir>: The root install directory for the IVI Shared Components, which consists of executables and other files needed to create and run IVI drivers. By default, this directory is "<ProgramFilesDir>\IVI Foundation\IVI".
- <RcName>: The name of a repeated capability. Repeated capabilities may be defined in class specs or by specific driver developers.
- <FwkVerShortName>: The IVI.NET short name for a version of the .NET Framework.

Where it is important to indicate the case of substituted text, casing is indicated by the case of the text between the brackets.

- <ClassName> indicates Pascal casing. For example, "IviDmm".
- <className> indicates camel casing. For example, "iviDmm"
- <classname> indicates all lower case. For example, "ividmm"
- <CLASSNAME> indicates all upper case. For example, "IVIDMM"
- <CLASS_NAME> indicates all upper case with underscores between words. For example, "IVI_DMM".

Features and Intended Use of IVI Drivers. Some behaviors of IVI drivers are optional and some behaviors are required. This section discusses both the optional and required features and indicates which are required and which are optional.

Section 5, *Conformance Requirements*, contains the precise requirements for IVI drivers. These requirements pertain to the behavior of the drivers as well as the terminology that the drivers use to describe their compliance with the behavioral requirements.

IVI-3.17: Installation Requirements Specification, contains the required and optional features for the installation programs that install IVI drivers and IVI shared components on user systems.

IVI-3.2: Inherent Capabilities Specification contains the specific requirements for the inherent API in all IVI drivers.

3.2 API Technology

The IVI Foundation defines requirements for the programming interfaces that an IVI driver exports. An IVI driver has at least a COM API, a C API, or a .NET API. An IVI driver may have -more than one API. An IVI driver may export other API types, such as C++ or LabVIEW. The IVI Foundation specifications do not preclude driver suppliers from developing drivers with other APIs.

Section 4, *IVI Driver Architecture*, contains the precise requirements for COM, C, and .NET APIs.

3.3 Interchangeability

The primary purpose of the IVI Foundation is to enable the development of interchangeable instrument drivers. IVI drivers must comply with several requirements to be interchangeable. The IVI Foundation also defines optional behaviors with which an IVI class-compliant specific driver may comply.

IVI custom specific drivers are not interchangeable and thus do not have to implement any of the behaviors described in this section.

The IVI configuration store plays an integral part in achieving interchangeability. The user specifies in the IVI configuration store which IVI class-compliant specific driver to use when an application initializes a session using a class-defined API. The user also specifies in the IVI configuration store initial values for some of the attributes of the driver. IVI drivers use the IVI configuration store in implementing several of the behaviors described in this section. Refer to Section 3.8, *Configuration of Inherent Features*, for more information on the use of the IVI configuration store to configure IVI features.

3.3.1 Compliance with a Class Specification

To be interchangeable, an IVI class-compliant specific driver complies with one of the IVI class specifications. The driver implements the base capabilities group defined in the IVI class specification. The driver may support zero, one, or more of the extension capabilities groups. If an IVI class-compliant specific driver implements an extension capability group, it implements the extension group completely.

An IVI class-compliant specific Driver should implement all the IVI extension capabilities groups that the instrument supports.

3.3.2 Accessing Class and Specific APIs

Typically instruments have features that go beyond what the class specifications define. An ideal IVI class-compliant specific driver would implement all the features of the instrument that are suitable for programmatic use. Driver developers are encouraged to implement instrument specific features. IVI class-compliant specific drivers should export instrument specific features in a way that is consistent with the class-defined capabilities. The driver should present the instrument specific features in such a way that the user can mix calls to class defined features and instrument specific features in a natural manner.

Exceptions to this guideline can occur if the driver developer chooses to export multiple perspectives on the same instrument. One perspective might be compliant with a class specification while another perspective might be focused on a unique paradigm of the instrument that is not inconsistent with the class-oriented perspective. Nevertheless, it might be feasible to extend the class-oriented perspective with instrument specific features, and drivers should do this where feasible.

When the user initializes the driver using the class-defined API, the user should be able to access the instrument specific features without initializing a new session. Therefore, the driver allows the user to access the instrument specific features without performing another initialization step.

3.3.3 Use of Virtual Identifiers for Repeated Capabilities

The IVI class specifications define repeated capability names, but they do not define repeated capability identifiers. Each IVI specific driver that complies with a class specification that specifies repeated capabilities defines its own physical identifiers for repeated capability instances. The set of physical repeated capability identifiers vary from one instrument to another in the same class. For example, the IviScope specification defines "Channel" as a repeated capability, but it does not define identifiers for channel instances, such as "CH1", "Chan1", "1", and so on. Therefore, it is not possible to develop an interchangeable program using the physical repeated capability identifiers that the IVI specific drivers define.

To refer to repeated capability instances in an interchangeable manner, the user defines virtual repeated capability identifiers and maps them to the physical repeated capability identifiers that the IVI specific driver defines. The user specifies the virtual identifiers and their mappings in the IVI configuration store. During initialization, the IVI specific driver retrieves the virtual identifiers and their mappings from the IVI configuration store. The IVI specific driver applies the mappings when the user calls driver functions that access repeated capabilities.

The value of virtual repeated capability identifiers is not restricted to instrument interchangeability. Using descriptive virtual identifiers can make an application program more readable. Therefore, IVI specific drivers support the use of virtual identifiers for instrument specific repeated capabilities as well as class-defined repeated capabilities.

The values that users pass to driver functions to identify repeated capability instances are called *repeated capability selectors*. Specific driver functions that take repeated capability selector parameters accept both virtual and physical identifiers. When the user passes a virtual identifier, the specific driver uses the virtual identifier mappings from the IVI configuration store to translate virtual identifiers to physical identifiers. Section 4.4, *Repeated Capability Selectors*, describes the syntax for repeated capability selectors.

To avoid naming conflicts, physical repeated capability identifiers should mirror the concise names that the instrument manual or front panel uses. For example, "CH1" or "1" are appropriate physical identifiers for channel 1. It is expected that users will select more descriptive terms for virtual identifiers.

A user might inadvertently choose a virtual identifier that is identical to a physical identifier that the IVI specific driver defines for the same repeated capability. Even if a user is careful not to use duplicate identifiers, conflicts might occur when the user replaces an instrument with a new instrument. Moreover, the user might map a virtual identifier that is the same as one of the physical identifiers to a different physical identifier. In the event that a virtual identifier is identical to a physical identifier, the IVI specific driver treats it as a virtual identifier and applies the user-defined mapping.

Mapping a virtual identifier to another virtual identifier is not allowed. Thus, IVI specific drivers apply the user-defined mappings only once.

Users may map multiple virtual identifiers to the same physical identifier.

An instrument specific function that returns a repeated capability identifier as an output parameter should return the virtual identifier instead of the physical identifier when a virtual identifier is available and it is important to provide the identifiers that the user has defined. If the user maps multiple virtual identifiers to the same physical identifier, the function may return any one of the virtual identifiers. *IVI-3.3: Standard Cross-Class Capabilities Specification* defines standard functions for discoverability of physical identifiers.

3.3.4 Disabling Unused Extensions

A user program might refrain from using a class extension capability group. Such a program should have the same behavior regardless of whether or not the instrument it uses has the extended capabilities. Making this possible imposes additional requirements on IVI class-compliant specific drivers.

If a program uses an instrument that has extended capabilities but the program does not configure the settings for the extended capabilities, the settings are in an unknown state. The unknown state could affect the behavior of the instrument capabilities that the program does use. Furthermore, the unknown state is likely to vary from one instrument to another. Thus, the program is likely to have different behavior when used with different instruments.

It is not reasonable to expect the program to configure the settings for the unused capabilities. To do so would require that all instrument drivers that the program might ever use must implement the extended capability group. Such a requirement would conflict with the principle that drivers have to implement only the extension capability groups that the instrument supports.

Therefore, the responsibility falls on the driver. An instrument driver that implements an extension capability group has the ability to make the instrument behave as if the instrument did not have the extended capabilities. Making instruments behave as if extension capability groups do not exist is called *disabling unused extensions*.

The IVI class specifications suggest how to disable each extension capability group. For example, the IviDmm base capabilities provide functions for taking a single measurement. The IviDmm class defines a multipoint extension group that provides functions for acquiring multiple samples from multiple triggers. If a program uses only the base capabilities with an instrument that has multipoint capabilities, the IviDmm specification suggests that the specific driver disable the multipoint extension group by ensuring that the trigger count and sample count are 1 when the instrument is armed for an acquisition.

Disabling unused extensions is a required behavior of IVI class-compliant specific drivers. The driver disables all extensions in the Initialize and Reset With Defaults functions. Normally, an extension remains disabled until a program explicitly uses it, and therefore the driver does not have to take any other action. However, if an operation on another capability group has the side effect of reconfiguring the disabled extension group, the driver disables the extension group again before the extension group affects the behavior of the instrument.

The driver also disables instrument specific features in the Initialize and Reset with Default functions when the features affect the behavior of the class-defined capabilities.

3.3.5 Applying Configurable Initial Settings from the IVI Configuration Store

An IVI specific driver defines the set of attributes for which it allows users to specify initial values in the IVI configuration store. The installation program for the driver includes special information in the entry for the IVI configuration store. Refer to Section 5.3.3, *Defining Configurable Initial Settings in the IVI Configuration Store*, in *IVI-3.17: Installation Requirements Specification*, for more information.

An IVI specific driver applies the configurable initial settings for the session when the user calls the Initialize and Reset With Defaults functions.

An IVI specific driver defines the order in which it applies the configurable initial settings.

IVI custom specific drivers may also implement this feature.

3.3.6 Interchangeability Checking

Interchangeability checking is an optional feature of IVI drivers.

If an IVI driver implements interchangeability checking and the user enables it, the driver generates an interchangeability checking warning when the driver encounters a situation in which the application program might not produce the same behavior when used with a different instrument. Different kinds of drivers are responsible for different kinds of interchangeability checking. Various conditions can generate interchangeability warnings, and drivers may perform various levels of interchangeability checking.

Three types of conditions can result in interchangeability checking warnings:

Attribute Not in a User-Specified State

A program is not interchangeable if it performs an operation that relies on a state the program does not fully specify. When this happens, the behavior of the operation is likely to vary based on either the initial state of the instrument or the state to which a previous program configured the instrument.

A state is not fully specified when one or more of the attributes that comprise the state are not in a *user-specified* state. An attribute is in a user-specified state when the program explicitly sets the attribute. The attributes remains in a user-specified state until the instrument setting to which the attribute corresponds changes as a result of the program configuring another attribute, or the program calls a reset or automatic setup function. Examples of reset functions include Reset and Reset With Defaults, as defined in *IVI-3.2: Inherent Capabilities Specification*. An example of automatic setup function is Auto Setup as defined in the IviScope class. Setting attribute values according to the configurable initial settings does not put attributes in a user-specified state.

Each IVI class specification defines guidelines for checking for this type of condition. The guidelines describe the operations that depend on instrument states and the attributes that comprise those states.

This form of interchangeability checking is performed on a capability group basis. When interchangeability checking is enabled, the driver always performs this interchangeability checking on the base capabilities group. In addition, the driver performs this interchangeability checking on all extension groups that the user has accessed.

This form of interchangeability checking is an optional feature of IVI class-compliant specific drivers. *IVI-3.2: Inherent Capabilities Specification* defines how IVI specific drivers that do not implement interchangeability checking behave when the user attempts to enable interchangeability checking. A driver that does implement this form of interchangeability checking can do so at either a *minimal* or *full* level.

A driver that implements minimal interchangeability checking considers an attribute to be in a user-specified state if the program has ever set the attribute. The driver does not account for cases where the value of the attribute changes as a side effect of the program configuring another attribute.

A driver that implements full interchangeability checking accounts for cases where the value of the attribute changes as a side effect of the program configuring another attribute.

Instrument Specific Value Used

IVI class-compliant specific drivers implement class-defined functions and attributes. Normally, these functions and attributes are also the basis of the instrument specific API for the driver. Consequently, IVI class-compliant specific drivers sometimes accept instrument specific values for parameters for which the class specification defines a discrete set of values.

IVI class drivers generally do not validate function parameters and attributes that represent instrument settings and thus do not prevent users from passing instrument specific values. However, a program that uses instrument specific values for class-defined function parameters and attributes is not interchangeable. Therefore, some IVI class drivers record interchangeability warnings when the program uses instrument specific values for class-defined function parameters and attributes.

This form of interchangeability checking is optional for IVI class drivers. The class-compliant APIs in IVI-COM and IVI.NET specific drivers may also implement this form of interchangeability checking.

Write Access to Read-Only Attribute

An IVI class-compliant specific driver might implement an attribute as read/write even though the class specification defines the attribute as read-only. However, a program that attempts to set a value for a class-defined read-only attribute is not interchangeable. Therefore, some IVI-C class drivers record interchangeability warnings when the program sets a read-only attribute.

This form of interchangeability checking is optional for IVI-C class drivers. IVI-COM and IVI.NET specific drivers do not implement this form of interchangeability checking because class-compliant

APIs of IVI-COM and IVI.NET specific drivers do not allow programs to set attributes that the class specification defines as read only.

If an IVI-C or IVI-COM driver implements interchangeability checking, it maintains a queue of warnings, and may impose a maximum on the size of queue that holds the interchangeability checking records. If the queue overflows, the driver discards the oldest interchangeability checking record.

If an IVI.NET driver implements interchangeability checking, and if a client has registered to receive the event, it raises a warning event for each warning at the time the warning is detected. Warnings are not raised in the driver constructor. The user must register an interchangeability checking warning event handler in order to receive interchangeability checking warning events.

3.3.7 Coercion Recording

Coercion recording is an optional feature for IVI class-compliant specific drivers.

Regardless of whether an IVI class-compliant driver implements coercion recording, the driver is required to coerce values according to the IVI class specification to which it complies.

IVI custom specific drivers may also implement coercion and coercion recording.

If an IVI-C or IVI-COM driver implements coercion recording, it maintains a queue or warnings, and may impose a maximum on the size of queue that holds the coercion records. If the queue overflows, the driver discards the oldest coercion record.

If an IVI.NET driver implements coercion recording, and if a client has registered to receive the event, it raises a warning event for each warning at the time the warning is detected. Warnings are not raised in the driver constructor. The user must register a coercion recording warning event handler in order to receive coercion recording warning events.

3.4 Range Checking

Range checking is a required feature of IVI specific drivers. IVI specific drivers return an error when a user specifies an invalid value for a function parameter or an attribute and range checking is enabled.

For a parameter or attribute that has a continuous range with a maximum and/or a minimum value, the value that the user passes is invalid if it is greater than the maximum or less than the minimum. For a parameter or attribute that has a set of discrete legal values, the value that the user passes is invalid if it is not in the set of legal values.

When validating a user value against a discrete set of legal values for a `ViReal64` parameter or attribute, IVI specific drivers shall not include a guard band around any of the legal values. Guard bands are unnecessary and make the driver less interchangeable. Refer to Section 3.9, *Comparing Real Numbers*, for information regarding how to handle the imprecision inherent in the computer representation floating point numbers.

3.5 Instrument Status Checking

Some IVI specific drivers can query the status of the instrument through the instrument I/O interface. For example, all IEEE 488.2 devices implement error status registers. Other IVI specific drivers can determine the status of the instrument without performing a separate query. For example, a driver for a plug-in instrument card might maintain the status information in the driver itself. Still other drivers cannot query the instrument status because the instrument provides no mechanism for this.

Instrument status checking is a required feature of IVI specific drivers that can query the status of the instrument. If the driver can query the status of the instrument through the instrument I/O interface, the driver implements code that queries the instrument status, interprets the response, and returns a special error code if the status indicates an error has occurred. The user then calls the Error Query function to retrieve the instrument specific error code from the instrument.

In general, if status checking is enabled, the driver invokes the status checking code from user-callable functions that perform instrument I/O.

3.6 Simulation

Simulation is a required feature of IVI specific drivers.

If simulation is enabled, an IVI specific driver does not perform instrument I/O, and the driver creates simulated data for output parameters. For example, in each function that performs instrument I/O, the I/O code might appear in a block that executes only when simulation is disabled. Functions that return output data contain code that generates simulated data and that executes only when simulation is enabled.

Typically, specific drivers generate very simple output data based on the existing configuration. For example, a driver for a DMM might create simulated output data for the Read operation by generating a random number within the configured measurement range.

If feasible, IVI specific drivers should perform the same range checking when simulation is enabled that it performs when simulation is disabled.

A specific driver that the user initializes with simulation disabled may allow the user to enable simulation at some point after initialization. If a specific driver does not allow a user to enable simulation at some point after initialization, the driver returns an error if the user attempts to enable simulation.

If initialized with simulation disabled, the specific driver creates an I/O session. To ensure that I/O sessions are always closed properly, the Close function in the driver always closes the I/O session regardless of whether simulation is enabled or disabled at the time the user calls the Close function.

A specific driver that the user initializes with simulation enabled does not create an I/O session. Consequently, the driver cannot perform I/O if the user subsequently disables simulation. Therefore, if the user enables simulation during initialization and then subsequently attempts to disable simulation, the driver returns an error.

3.7 State Caching

State caching is an optional feature of IVI specific drivers. For each hardware configuration attribute, an IVI specific driver chooses whether to cache the state of the attribute. An IVI specific driver may cache all, some, or none of its hardware configuration attributes. A driver supports state caching if it caches one or more of its hardware configuration attributes.

If the user enables state caching, the IVI specific driver maintains a software copy of what it believes to be the current instrument setting for each attribute the driver chooses to cache. If the IVI specific driver believes that the cache value accurately reflects the state of the instrument, it considers the cache value to be *valid*. If the instrument driver does not believe the cache value reflects the state of the instrument, it considers the cache value to be *invalid*. The driver avoids performing I/O when the cache value is valid and the user sets the attribute to same value.

3.8 Configuration of Inherent Features

Users can enable or disable many of the inherent features of IVI drivers. Users enable and disable the features through attributes. These configurable inherent features are the following:

- Interchangeability Checking
- Simulation
- Range Checking
- Coercion Recording
- State Caching

- Instrument Status Checking

For each attribute, the user can specify an initial value in the IVI configuration store or through the `OptionsString` parameter of the Initialize function. The IVI driver honors the value that the user specifies in the `OptionsString` parameter of the Initialize function. If the user does not specify a value in the `OptionsString` parameter, the driver honors the value the user specifies in the IVI configuration store. If the user does not specify a value in either the `OptionsString` parameter or the IVI configuration store, the driver uses the default value that *IVI-3.2: Inherent Capabilities Specification* specifies for the attribute.

When a program creates a session through an IVI class driver, the IVI-COM Session Factory, or one of the IVI.NET session factory methods, the user must specify a logical name that identifies the driver, the instrument, and the initial settings.

When a program creates a session through the specific driver directly, the user can specify either a logical name or an I/O resource descriptor. All IVI specific drivers can accept logical names as well as I/O resource descriptors for the Resource Name parameter in the Initialize function. Notice that if the user specifies an I/O resource descriptor, the user cannot expect the driver to use settings from the IVI configuration store.

Table **3-1** summarizes the order of precedence the driver uses to assign initial values for inherent features.

Table 3-1. Precedence Order of Inherent Features Configuration

Type of Resource Name	Logical Name	I/O Resource Descriptor
Source of attribute value, in order of precedence	`OptionsString` parameter	`OptionsString` parameter
	IVI configuration store	default value
	default value	

3.9 Comparing Real Numbers

Because of the imprecision inherent in the computer representation of floating point numbers, it is not appropriate for instrument drivers to determine whether two real numbers are equal by comparing them based on strict equality. Therefore, IVI drivers use fuzzy comparison with approximately 14 decimal digits of precision when comparing any `ViReal64` values with `ViReal64` values that instruments return or that users specify as parameter or attribute values.

IVI-C and IVI-COM drivers that generate or recognize infinity or NaN values use the Floating Point shared component. Refer to *IVI-3.12: Floating Point Services Specification*. IVI.NET drivers use Double.NegativeInfinity, Double.PositiveInfinity, or Double.NaN.

3.10 Multithread Safety

Multithread safety is a required feature of IVI drivers.

All IVI drivers prevent simultaneous access to a session in multiple threads of the same process from interfering with the correct behavior of the driver. This level of multi-thread safety is the only level that IVI drivers are required to implement.

IVI.NET drivers may offer a higher level of multi-thread safety. This is discussed further in Section 5.10.7, *Multithread Safety*.

3.11 Resource Locking

The IVI Foundation has not defined the requirements for Resource Locking, or for managing simultaneous access from other users.

3.12 Events

In some cases it is useful for an instrument driver to notify an application program that an event has occurred.

Drivers can implement events using a callback mechanism. An IVI driver need not implement the callback mechanism unless it uses events to notify client programs. The standard IVI interfaces do not use events. Each driver is responsible for documenting the events it uses.

IVI-COM drivers do not use COM connection points for event notification because connection points require using ActiveX automation and exhibit poor performance, especially when remote DCOM access is involved.

3.13 Use of I/O Libraries for Standard Interface Buses

IVI specific drivers use the VISA I/O library for I/O communication over a GPIB or VXIbus interface. The VISA I/O library has two distinct API types: VISA-C and VISA-COM. The VXI*plug&play* specification *VPP-4.3.2: VISA Implementation Specification for Textual Languages* defines the VISA-C API. The VXI*plug&play* specification *VPP-4.3.4: VISA Implementation Specification for COM* defines the VISA-COM API. IVI drivers that use VISA use one or both of the API types. The IVI Foundation specifications do not preclude driver suppliers from developing drivers that interface with additional I/O libraries, as long as the driver works when VISA is present and the additional I/O libraries are not present. For example, a driver that communicates over GPIB might work as long as either the VISA-C API or the NI-488 API is present.

For bus interfaces other than GPIB or VXIbus, a commonly defined standard I/O library should be used, if available. It is recommended that driver suppliers use the VISA I/O library if the bus and protocol are supported by VISA. If VISA is not used, care should be taken to handle ADE differences and OS differences. If a proprietary I/O library is used, the driver supplier should ensure that it can co-exist with VISA. In the future, the IVI Foundation might require the use of specific I/O libraries, such as VISA, for other interface types without precluding drivers from using additional I/O libraries.

The VISA Shared Components provided by the IVI Foundation include .NET Primary Interop Assemblies (PIAs) that can be easily accessed by developers of IVI.NET drivers.

3.13.1 Direct I/O

IVI drivers are not required to implement the complete set of an instrument's capabilities. Consequently, users of IVI drivers for instruments that communicate via message-based interfaces may want to send messages to and receive results back directly from instruments. To enables this, IVI drivers for message-based instruments are required to provide a general I/O mechanism. IVI-3.4, *API Style Guide*, defines a common API for this purpose.

The Direct I/O API includes read and write functions as well as an optional attribute that holds a reference to the underlying I/O.
The Direct I/O API is not applicable for instruments that are not message-based (for example, PXI and VXI devices). Since the API does not apply to all instruments, IVI drivers that include the API implement it as a part of their instrument specific interfaces.

3.14 Shared Components

The IVI shared components are software modules that enable IVI drivers to adhere to the rules outlined in the IVI specifications. IVI drivers are required to use the shared components to ensure interoperability with other IVI drivers. These components provide:

- Access to the IVI configuration store
- Dynamic loading/instantiation of specific instrument drivers
- Thread-safe session management for IVI-C drivers

- Error reporting services for IVI-C drivers
- A standard way for returning and recognizing infinity and NaN values

For more information on the use requirements for the shared components, refer to the following specifications:

IVI-3.5: Configuration Server Specification
IVI-3.6: COM Session Factory Specification
IVI-3.9: C Shared Components Specification
IVI-3.12: Floating Point Services Specification

3.15 Source Code Availability

IVI drivers shall include source code if the source code is a simple translation of the driver calls to a separate publicly documented and officially supported interface and does not include proprietary or confidential content. Users sometimes find it very valuable to have access to instrument driver source code. Having access to source code allows a user to debug and fix instrument drivers in time critical situations when the user cannot wait for the driver supplier to fix the problem.

A driver supplier that does not distribute source code should make an effort to deliver high-quality drivers and provide comprehensive technical support.

IVI drivers that include source code shall also provide instructions on rebuilding the driver in at least one publicly available development environment.

3.16 Extent of Instrument Functionality Covered by IVI Drivers

IVI drivers for instruments that have an ASCII command set such as SCPI shall implement the full functionality of the instrument available via commands and queries with a few exceptions. Some commands and queries are not suitable for an instrument driver or could even break driver or instrument functionality if exposed to the user. An IVI driver for a SCPI-based instrument, for example, might not implement SCPI commands from the following nodes:

- DIAGnostic
- FORMat (may be used internally but not exposed to users)
- SYSTem:COMMunicate
- Service or Factory Calibration functionality
- Undocumented SCPI (factory use only)
- Other features not normally accessed through the programmatic interface, for example:
  - DISPlay
  - HARDCopy
  - MEMory:STATe
  - CURSor

IVI driver users can send any commands to a message-based instrument using the driver's Direct I/O functions.

IVI drivers for instruments that have an ASCII command set such as SCPI shall document the implemented ASCII commands and characterize the commands that are not implemented, and the characterization should have a level of detail that is consistent with the list of commonly unimplemented functionality provided above.

Partially implemented ASCII commands may be documented as "implemented".

4. IVI Driver Architecture

The IVI Foundation currently standardizes on three interface technologies: COM, ANSI-C, and .NET. IVI drivers conform to one or more of the standard technologies. IVI driver suppliers choose which architectures to support based on the needs of their customers. As computer and software technology evolves, other interface technologies may become popular within the instrument control community. As this change occurs, new interfaces may be defined to incorporate new capabilities.

This section discusses issues specific to the COM, C, and .NET architectures.

Section 5.14.1, *Enumerations*

For all types of IVI drivers, enumeration values shall be explicitly specified in the source code for the enumeration. One of the members shall be assigned a value of zero.

IVI-COM Requirements, contains the precise requirements specific to IVI-COM drivers.

Section 5.16, *IVI-C Requirements*, contains the precise requirements specific to IVI-C drivers.

Section 5.17, *IVI.NET Requirements*, contains the precise requirements specific to IVI.NET drivers.

4.1 IVI-COM Driver Architecture

This section describes how IVI-COM instrument drivers use COM technology. This section does not attempt to describe the technical features of COM, except where necessary to explain a particular IVI-COM feature.[2] This section assumes that the reader is familiar with COM technology.

IVI-COM drivers implement the IVI instrument driver features described in Section 1.7, *Substitutions*

This specification uses paired angle brackets to indicate that the text between the brackets is not the actual text to use, but instead indicates the kind of text that can be used in place of the bracketed text. Sometimes the meaning is self-evident, and no further explanation is given. The following list includes those that may need additional explanation for some readers.

- <ClassName>: The name of an IVI instrument class as defined by an IVI Instrument Class specification. For example, “IviDmm”.
- <ClassType>: The name of an IVI instrument class as defined by an IVI Instrument Class specification, without the leading “Ivi”. For example, “Dmm”.
- <ComponentIdentifier>: For IVI-COM and IVI.NET, the string returned by a specific driver’s Component Identity attribute. This string uniquely identifies the driver. For example, “Agilent34410”.
- <Prefix>: For IVI-C class drivers, the string returned by the driver’s Class Driver Prefix attribute. The class driver prefix will commonly be an IVI class name, but may be different. For example, “IviDmm”. For IVI-C specific drivers, the string returned by the driver’s Specific Driver Prefix attribute. For example, “NI3456”
- <CompanyName>: The name of the driver vendor (not the instrument manufacturer). For example, “Agilent Technologies, Inc”.
- <ProgramFilesDir>: The Windows program files directory. This varies across different versions of

[2] For those who are new to COM technology, we recommend the following books:

- Chappell, David, ***Understanding ActiveX and OLE***, Microsoft Press, 1996. A high level introduction.
- Rogerson, Dale, ***Inside COM***, Microsoft Press, 1997. A technical tutorial with examples.
- Box, Don, ***Essential COM***, AddisonWesley, 1998. A indepth technical discussion of how COM developed and why it’s features are valuable.

Windows. In some contexts, it is not intended to differentiate between the 64-bit and 32-bit program files directories found on 64-bit versions of Windows that include Windows On Windows (WOW), but to be understood as a generic reference to the program files directory.

- <ProgramDataDir>: The Windows data directory. This varies across different versions of Windows. It is generally understood to apply to all users.
- <IviStandardRootDir>: The root install directory for the IVI Shared Components, which consists of executables and other files needed to create and run IVI drivers. By default, this directory is "<ProgramFilesDir>\IVI Foundation\IVI".
- <RcName>: The name of a repeated capability. Repeated capabilities may be defined in class specs or by specific driver developers.
- <FwkVerShortName>: The IVI.NET short name for a version of the .NET Framework.

Where it is important to indicate the case of substituted text, casing is indicated by the case of the text between the brackets.

- <ClassName> indicates Pascal casing. For example, "IviDmm".
- <className> indicates camel casing. For example, "iviDmm"
- <classname> indicates all lower case. For example, "ividmm"
- <CLASSNAME> indicates all upper case. For example, "IVIDMM"
- <CLASS_NAME> indicates all upper case with underscores between words. For example, "IVI_DMM".

Features and Intended Use of IVI Drivers, and present them to test system programmers in the form of COM interfaces.

4.1.1 Target Operating Systems

Refer to Section 2.16, *Operating Systems*.

4.1.2 Target Languages and Application Development Environments

IVI-COM drivers work in the target languages and application development environments listed in Table **4-1**.

Table 4-1. Target languages and ADEs for IVI-COM drivers

32-bit	**64-bit**
Agilent VEE	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++
Microsoft Visual Basic	
Microsoft Visual Basic for Applications	Microsoft Visual Studio Tools for Office (VSTO)
National Instruments LabVIEW	National Instruments LabVIEW
National Instruments LabWindows/CVI	National Instruments LabWindows/CVI

* Note: The intent is to support the 64-bit versions of these ADEs when they are available.

In principle, IVI-COM drivers can work in other development environments in which COM or .NET is supported, including Borland C/C++ and Microsoft .NET Common Language Specification (CLS) compliant languages.

IVI-COM drivers need not support IDispatch based environments such as VB Script, Visual J++, or J Script. However, it is possible to construct wrappers around IVI-COM drivers to support IDispatch and the appropriate data conversions.

4.1.3 IVI-COM Driver Overview

An IVI-COM instrument driver is instantiated as a COM object accompanied, optionally, by COM helper objects. An IVI-COM driver object exposes multiple interfaces, including standard IVI interfaces, instrument specific interfaces, and a limited set of standard COM interfaces.

Standard IVI interfaces include IVI inherent interfaces and IVI class-compliant interfaces for each of the defined IVI instrument classes. Standard IVI interfaces provide a standard syntax through which application programs interact with the driver. This standard syntax, when used in combination with the IVI-COM Session Factory, provides the same level of syntactical interchangeability in the IVI-COM architecture as that provided by IVI.NET and by IVI class drivers in the IVI-C architecture. Thus, IVI class drivers are not required for syntactical interchangeability in the IVI-COM architecture. Refer to Section 2.9.2.2, *How Interchangeability Works in COM,* for more details on achieving interchangeability using IVI-COM drivers.

Instrument specific interfaces provide access to instrument specific functionality. Typically, instrument specific interfaces mirror IVI class-compliant interfaces for the instrument features that are within the scope of the functionality defined by the IVI class specifications. However, instrument specific interfaces also include additional methods and properties that provide access to the features that are beyond the scope of the IVI class specifications.

To use an IVI-COM driver in an IVI-C environment, a layer of code translates IVI-C calls to IVI-COM calls that the underlying IVI-COM driver can recognize. This layer of code conforms to the IVI-C API requirements and to additional requirements specified in Section 7, *Specific Driver Wrapper Functions*, in *IVI-3.2: Inherent Capabilities Specification*.

To use an IVI-C driver in an IVI-COM environment, a layer of code translates IVI-COM calls to IVI-C calls that the underlying IVI-C driver can recognize. This layer of code conforms to the IVI-COM API requirements and to additional requirements specified in Section 7, *Specific Driver Wrapper Functions*, in *IVI-3.2: Inherent Capabilities Specification*.

4.1.4 IVI-COM Interfaces

The methods and properties of IVI-COM drivers are grouped into multiple interfaces based on functionality. This grouping allows for a natural hierarchical structure that organizes the overall driver functionality.

All IVI-COM drivers contain the interfaces for the IVI inherent features as well as interfaces that implement the instrument specific capabilities of the instrument. IVI-COM class-compliant drivers also contain IVI class-compliant interfaces. The interfaces for the IVI inherent features are defined in *IVI-3.2: Inherent Capabilities Specification*. The IVI-COM class-compliant interfaces are defined in the IVI class specifications.

The IVI class-compliant interfaces for a particular instrument class are identical from driver to driver. The interfaces that implement the IVI inherent methods and properties are identical for all IVI-COM drivers and have the same interface ID (IID). Keeping the interfaces identical is what makes the drivers syntactically interchangeable.

Instrument specific interfaces are necessarily different from one instrument to another, whereas class-compliant interfaces are identical for all drivers within the same class. Thus, the class-compliant interfaces in an IVI-COM driver are always different from the driver's instrument specific interfaces. Typically, the IVI class-compliant interfaces are thin layers of code that call instrument specific interfaces. This allows the instrument specific interfaces to leverage the syntax of the class-compliant interfaces.

4.1.5 Interface Reference Properties

Navigating from one COM interface to another can be accomplished in two ways. One way is to use QueryInterface to query directly for a particular interface. The other way is for interfaces to contain properties that refer to other interfaces. These properties are called *interface reference properties*.

The first technique is particularly suited for Microsoft Visual C++ users. Microsoft Visual Basic users can also use the first technique through *implicit casting*. In implicit casting, the user calls QueryInterface implicitly through the `Set` statement.

The second technique can be used from Visual C++, but is particularly useful in Visual Basic. Visual Basic recognizes the interface types of interface reference properties. When a user types in the name of an interface reference property in the Visual Basic editor, Visual Basic uses its knowledge of the interface type to display a list of methods and properties in the interface. This list can contain other interface reference properties. Thus, Visual Basic users can navigate through an arbitrary number of interface reference properties. Consider the following Visual Basic statement:

```
Itf1.Itf2.Itf3.Itf4.Method
```

`Itf1` is a reference to an interface held in the variable `Itf1`. Similarly, `Itf2`, `Itf3`, and `Itf4` are also interface reference properties. As the user types each of these names, Visual Basic displays a dropdown list of methods and properties in the corresponding interface. After typing `Itf1` followed by a period, a list of all of the properties and methods in `Itf1` appears, allowing the user to select one. After selecting `Itf2` and typing the period, a list of the methods and properties in `Itf2` appears, and so on, until the programmer selects `Method`. This behavior is part of Visual Basic's IntelliSense feature.

In addition, IVI-COM drivers occasionally require *helper* objects. For example, IVI-COM drivers include a helper object for each collection. Application programs cannot query the main driver object for a helper object's interface. Instead, the application program accesses the helper object through an interface reference property. Refer to Section 4.1.9, *Repeated Capabilities*, for more information on IVI-COM collections.

IVI-COM drivers make extensive use of interface reference properties to navigate from interface to interface. Any interface that can be reached using an interface reference property can also be reached by calling QueryInterface on the object that implements the interface.

4.1.6 Interface Hierarchy

IVI-COM drivers take advantage of interface reference properties to organize interfaces hierarchically. Each interface has exactly one parent interface and zero or more child interfaces. No circular references or series of references exist. The interface hierarchy is primarily organized by functionality, while also being consistent with COM conventions where possible. The hierarchy for inherent features is documented in the *IVI-3.2: Inherent Capabilities Specification*. The hierarchies for class-compliant interfaces are documented in the corresponding IVI class specifications.

IVI-COM class-compliant interface hierarchies are not necessarily organized according to the capability groups defined in the IVI class specifications. Furthermore, IVI-COM interface hierarchies do not necessarily correspond to the IVI-C function tree (`.fp` file) hierarchies.

4.1.7 Custom vs. Automation Interfaces

IVI-COM drivers expose *custom interfaces*, which inherit directly from IUnknown, rather than *automation interfaces*, which inherit from IDispatch. Using custom COM interfaces instead of automation interfaces leads to COM objects that are simpler to develop, capable of high performance, easier to version, and usable from most application development environments. The subsections that follow explain these advantages in detail.

4.1.7.1 Simpler to Develop

Although, in general, a single COM object can expose many interfaces to the user, a COM object with an automation interface cannot expose multiple interfaces. The IDispatch interface, from which automation interfaces inherit, cannot be implemented to recognize more than one interface per COM object. Thus, a separate COM object must exist for each user-visible automation interface. If IVI-COM drivers used automation interfaces, the drivers would require many COM objects, one for every IVI standard interface and every instrument specific interface supported by the driver. All the objects would have to coordinate access to shared resources, such as I/O, and global variables, such as cached state information.

To summarize, automation implies a fragmented, one-object-per-interface implementation strategy, whereas custom interfaces allow many interfaces to exist in a single, coordinated object.

4.1.7.2 Capable of High Performance

When a single object implements multiple custom interfaces, QueryInterface can be used to obtain interface references. Using QueryInterface is faster than using interface reference pointers. This is particularly true when using DCOM or when obtaining a reference that involves marshalling. Using custom interfaces rather than multiple objects with automation interfaces enables performance-sensitive applications to benefit from the speed of QueryInterface when navigating among the interfaces of an IVI-COM driver.

Every use of an interface reference property results in a separate call to the driver. COM is unable to avoid making this call. The function that implements the interface reference property might call QueryInterface internally to obtain the reference, create a new object each time it executes, or return a cached pointer. On the other hand, QueryInterface has a predictable implementation, and calls to QueryInterface are optimized in a variety of ways when marshalling.

Programmers can minimize performance penalties related to interface reference properties with careful programming. For example, application programmers can cache the value of each pointer or can use a `With` statement. In languages where QueryInterface cannot be called directly, a mechanism that calls QueryInterface indirectly might be available. For example, in Visual Basic, the `Set` command calls QueryInterface if the target of the `Set` statement assignment is declared as an interface type.

4.1.7.3 Easier To Version

In COM, effective versioning involves creating a new interface whenever an interface changes. Application programs that query for the old interface get the old interface, whereas applications that know how to query for the new interface can obtain the new interface. Vendors can upgrade their drivers to recognize new versions of interfaces without breaking applications of the old versions of the driver. Therefore, effective COM versioning involves having multiple interfaces on an object. Since multiple automation interfaces on an object is problematic, custom COM interfaces make versioning easier.

4.1.7.4 Accommodating Automation

Automation is required in some cases, particularly when using scripting languages such as VB Script and J Script.

Because IVI-COM drivers are restricted to automation data types, it is possible to create wrappers that expose automation interfaces. Developers are encouraged to create such wrappers particularly if providing support for scripting languages is an important consideration. These wrappers can be included in the same IVI-COM driver as the custom IVI-COM interfaces.

The resulting architecture, automation wrapper on top of the custom IVI-COM interfaces of an IVI driver, is probably the most efficient way of developing automation drivers. Notice, however, that because automation wrappers are not a requirement for IVI-COM drivers, interchangeability is limited to IVI-COM class-compliant drivers that implement automation wrappers for their class-compliant interfaces.

4.1.8 Data Types

IVI-COM interfaces are restricted to a subset of automation data types. Refer to Section 5.14, *Allowed Data Types*, for the set of data types allowable in IVI-COM driver interfaces. This enables IVI-COM drivers to work well in a variety of ADEs, including automation-based ADEs, and allows easier integration with IVI-C drivers.

4.1.8.1 Enumerations

IDL enumerations for IVI-COM drivers are strongly typed. Two enumerations that otherwise could refer to the same attribute but have a different set of enumeration values are typed differently.

For instance, all IVI-COM drivers that comply with the IviDmm specification have an enumeration for the Function property. However, not all DMMs support the same function values. Drivers for DMMs with different sets of function values have different instrument specific enumerations for the Function property. Furthermore, unless a DMM has exactly the same set of function values as the set defined for the class, the instrument specific enumeration is distinct from the class enumeration.

4.1.8.2 Safe Arrays

Arrays in IVI-COM driver interfaces are implemented as COM safe arrays. For example, an array of longs is declared as `SAFEARRAY(long)`. Safe arrays are self-describing and include the number of dimensions and the size of each dimension. When safe arrays are passed as parameters, separate size parameters are not necessary, nor are parameters that indicate the number of elements in the array. Instead, the safe array is created with the exact size necessary to hold the number of elements in the array.

For input safe arrays, the application program must create the safe array with the exact number of elements necessary to hold to input data. There are no separate parameters to specify array size or number of elements. Suppose an application program must pass an arbitrary waveform of 200 points to the driver. The application creates a safe array with exactly 200 elements, fills the array with the waveform points, and passes the array to the driver.

For output safe arrays, the driver creates the array similarly or modifies an input safe array to achieve the same end. The application program must examine the safe array to determine the number of elements in the array before processing it.

4.1.9 Repeated Capabilities

Repeated capabilities may be represented in two ways in IVI-COM drivers. Repeated capability instances may be specified by a method that selects the active instance (the selector style), or by selecting a particular instance from an IVI-COM collection (the collection style).

IVI-COM collections are similar to, but not the same as, standard ActiveX collections. Standard ActiveX collections are built with automation interfaces. IVI-COM collections are built with custom interfaces. IVI-COM collections involve two interfaces. One interface represents a single item in the collection. The name of the interface is singular. The second interface represents the set, or collection, of individual items, and the name is plural. For instance, the interface for an individual oscilloscope channel is named IIviScopeChannel whereas the interface for the collection is named IIviScopeChannels.

Refer to Section 4, *Repeated Capability Group,* in *IVI-3.3: Standard Cross-Class Capabilities Specification*, for the specific API requirements for the selector and collection styles. Both ways of representing repeated capabilities provide the user with the ability to navigate through repeated capability hierarchies.

Refer to Section 4.4, *Repeated Capability Selectors*, for information on how users specify a repeated capability instance within a hierarchy and a set of repeated capability instances.

Note: IVI-C drivers often use repeated capability name parameters to each method or attribute that accesses the repeated capability (the *parameter style*). Use of the parameter style is discouraged in IVI-COM APIs because parameterized properties become get/set methods in .NET PIA code.[3]

A class specification may use collections in the IVI-COM API and the string parameter approach in the IVI-C API. Section 12, *Repeated Capabilities*, in *IVI-3.4: API Style Guide,* describes each approach and contains guidelines for choosing among the different alternatives.

4.1.10 Session

Session parameters are not used in IVI-COM methods and properties. The object identity serves the same purpose in IVI-COM as the session does in IVI-C.

4.1.11 Interface Requirements

IVI-COM drivers expose their functionality through interfaces. IVI instrument drivers in general have some features that are common to all drivers and others that are common to all drivers of a particular instrument class. In IVI-COM instrument drivers, these features are exposed through well-defined, standard interfaces. IVI-COM specific instrument drivers also may expose instrument specific functionality through interfaces that are specific to that driver.

4.1.11.1 Standard COM Interfaces

IVI-COM drivers implement two standard COM interfaces, ISupportErrorInfo, and IProvideClassInfo2. Driver users do not typically use these interfaces directly.

The names for all interfaces defined by IVI start with IIvi.

[3] The Fgen IVI-COM API uses this technique for historical reasons.

4.1.11.2 Inherent Features

Every IVI-COM driver implements a set of interfaces that expose the IVI inherent features as described in *IVI-3.2: Inherent Capabilities Specification*. An IVI-COM custom driver does not implement any other standard IVI interfaces. IIviDriver is the root of the IVI inherent interfaces. When an IVI-COM driver is instantiated, QueryInterface for IIviDriver always returns a valid interface reference.

The hierarchy of IVI-COM inherent interfaces is described in Section 4.2, *COM Inherent Capabilities*, in *IVI-3.2: Inherent Capabilities Specification*.

4.1.11.3 Class-Compliant Interfaces

Every IVI-COM class-compliant driver implements a set of interfaces that export the IVI class-compliant features defined in the corresponding IVI class specification. For interfaces defined in an IVI instrument class, the interface names begin with I<*ClassName*>, where <*ClassName*> represents the instrument class name. The root interface name has nothing more added to the instrument class name. For other interfaces, additional words are added that, when possible, are the same for the corresponding levels in the C function hierarchy.

I<*ClassName*> inherits from IIviDriver. Interface reference properties to other class-compliant interfaces are included in I<*ClassName*> to provide access to the class-compliant interface hierarchy. In rare cases, commonly used methods and properties for accessing the instrument may also be included in I<*ClassName*>.

When an IVI-COM class-compliant driver is instantiated, QueryInterface for I<*ClassName*> always returns a valid interface reference.

IVI-COM drivers may implement class-compliant interfaces for multiple instrument classes.

4.1.11.4 Instrument Specific Interfaces

Instrument specific interfaces begin with I<*ComponentIdentifier*> where the <*ComponentIdentifier*> is a unique identifier for the driver. The Component Identifier is same as the Component Identifier attribute defined in Section 5.12, *Component Identifier*, in *IVI-3.2: Inherent Capabilities Specification*. The root interface of the primary instrument specific interface is named I<*ComponentIdentifier*>. For example, an instrument specific trigger interface for the Agilent 34401 DMM would be named IAgilent34401Trigger, and the root interface would be IAgilent34401.

Insofar as is practical, the instrument specific interfaces of an IVI class-compliant driver reflect the syntax of the class-compliant interfaces.

4.1.11.5 Default Interfaces

Default interfaces are applicable to Microsoft Visual Basic version 6.0 and Visual Basic for Applications. The default interface for all IVI-COM drivers is I<*ComponentIdentifier*>. This may seem counter-intuitive, since from the perspective of interchangeability it might seem that the root IVI class-compliant interface should be the default. However, interchangeability without application code changes can be achieved only by using the IVI-COM Session Factory, which always returns IUnknown rather than the default interface. The default interface plays no role in interchangeability and thus can be used for cases where the application program wants access to the instrument specific interfaces.

4.1.11.6 Instrument Specific Direct I/O API

A specific IVI-COM driver for device(s) that use message-based communication includes a System interface with methods for reading and writing string and bytes and a property for setting the I/O timeout. It may optionally include a property that provides access to the driver's underlying I/O.

The root interface of such an IVI-COM driver includes a reference to the System interface.

4.1.12 Driver Type Libraries

To allow users to swap instruments without recompiling or re-linking, the IVI Foundation publishes type libraries for standard IVI interfaces. One type library contains the IVI inherent interfaces, and one type library exists for each class specification.

4.1.13 Versioning COM Interfaces

The IVI-COM inherent and class-compliant interfaces are uniquely identified by IIDs and will not change after being published in *IVI-3.2: Inherent Capabilities Specification* or the corresponding class specification.

When the IVI Foundation approves a class specification or the specification for inherent capabilities, it also approves the corresponding type library. After the IVI Foundation distributes a type library, the interfaces that the type library defines are not subject to modifications. Any modifications necessary as a result of a specification change requires the creation of one or more new interfaces. Note that a new version of an existing interface is, in fact, a new interface.

Because interfaces are strongly typed in IDL, an interface reference property changes when the interface to which it refers has a new version. In turn, the interface that contains the interface reference property requires a new version. In an interface hierarchy, this process continues to the top level of the hierarchy.

Enumerations are strongly typed in COM. Any interface that contains any reference to an enumeration requires a new version when any changes are made to the enumeration. Following this rule strictly implies that an enumeration requires a new version whenever values are added or deleted from the enumeration. In practice, drivers may compromise this principle at the expense of displaying an inaccurate list of values in IntelliSense.

An IVI-COM driver may implement multiple versions of the same interface. This means that, when drivers are updated, application programs that access old interfaces will work using the new driver, while new applications can access new interfaces.

4.1.14 Driver Classes

Note: In this section, *class* refers to a COM class rather than an IVI instrument class.

Drivers may consist of more than one class. In fact, multiple classes are necessary if the driver implements IVI collections. The user instantiates the *main* driver class. The main IVI driver class is named <*ComponentIdentifier*>. The main driver class implements all the IVI inherent interfaces and all the IVI class-compliant interfaces other than collection interfaces. This ensures that QueryInterface succeeds for well-known standard interfaces. The main driver class also implements I<*ComponentIdentifier*> and all the instrument specific interfaces that syntactically leverage the standard IVI interfaces.

Typically, only the main driver class is registered. Helper classes, such as classes that implement collections, are not registered. If the driver contains an ActiveX automation wrapper, the automation class that implements I<*ComponentIdentifier*> is also registered.

Driver classes are packaged as 32-bit or 64-bit DLLs. If an IVI-COM driver is a COM wrapper on top of an IVI-C driver, the IVI-COM class and the IVI-C driver may be packaged as one DLL. An ActiveX automation wrapper may be packaged in the same DLL as the main driver class. Refer to Section 5.15.10, *Packaging*, for more information on file and module requirements.

4.1.15 IVI-COM Error Handling

IVI-COM drivers report status using standard `HRESULT` codes and a COM error object. Each type library defines an enumeration of all the status codes that the interfaces defined in the type library return. The enumerations provide constant identifiers for each status value.

Refer to Section 5.12, *IVI Error Handling*, for details of IVI error handling.

4.1.16 Threading

COM drivers are implemented to live in the multi-threaded apartment (MTA).

IVI-COM drivers are thread safe and are registered with the “Both” threading model.

Application program threads that call IVI-COM driver functions are expected to call `CoInitializeEx` with `COINIT_MULTITHREADED` as the value of the 2nd parameter.

4.1.17 Driver Packaging

Refer to Section 5.15.10, *Packaging*, for packaging requirements for IVI-COM drivers.

4.2 IVI-C Driver Architecture

This section discusses issues specific to IVI-C drivers.

4.2.1 Target Operating Systems

Refer to Section 2.16, *Operating Systems*.

IVI-C drivers can work on non-supported operating systems if the following conditions are met:

- A compiled version of the IVIC driver is available, or source code is available and an ANSIC compiler is available for that operating system.
- The C shared components are compiled and available for that operating system.
- An I/O library that the IVIC driver uses is available for that operating system.
- Any other support libraries that the driver uses are available for that operating system.

To enable use on non-supported operating systems, IVI-C drivers should avoid making operating system specific calls.

Note: The IVI Foundation does not define vendor-interoperability and cross-vendor interchangeability for drivers that are ported to other operating systems. Therefore, the IVI Foundation considers such drivers to be non-compliant when used on other operating systems.

4.2.2 Target Languages and Application Development Environments

IVI-C drivers work in the target languages and application development environments listed in Table **4-2**. In principle, IVI-C drivers can work in other ADEs that allow calls to dynamic link libraries, such as Borland C/C++.

Table 4-2. Target languages and ADEs for IVI-C drivers

32-bit	64-bit
Agilent VEE	Agilent VEE*
MathWorks MATLAB	MathWorks MATLAB
Microsoft Visual C++	Microsoft Visual C++
National Instruments LabVIEW	National Instruments LabVIEW
National Instruments LabWindows/CVI	National Instruments LabWindows/CVI

* Note: The intent is to support the 64-bit versions of these ADEs when they are available.

IVI-C drivers can also work in Microsoft Visual C# and Visual Basic .NET, but only with additional development effort or the use of 3rd party tools.

4.2.3 IVI-C Driver Overview

This section provides a general overview of the different types of IVI-C drivers and how they work together in a system.

4.2.3.1 Class and Specific Drivers

An IVI-C driver exports a C API. All IVI-C drivers export functions and attributes that comply with *IVI-3.2: Inherent Capabilities Specification*. The additional functions and attributes that an IVI-C driver exports depend on the type of driver. An IVI-C driver is a class driver, a class-compliant specific driver, or a custom specific driver.

- An IVIC class driver exports the complete set of functions and attributes defined in one of the IVI class specifications, including the base capabilities and all extension capabilities. The include file for an IVIC class driver contains C definitions for all the attribute values and error codes that the class specification defines.
- An IVIC classcompliant specific driver exports functions and attributes for the class capabilities that it implements. It may also export instrument specific functions and attributes.
- An IVIC custom specific driver does not comply with any of the defined class specifications. It exports instrument specific functions and attributes.

Although IVI-C class drivers export inherent, base, and extension capabilities, they do not actually implement them. Except for a few inherent functions and attributes defined exclusively for class drivers, class driver functions and attributes provide a pass-through layer to the IVI-C specific driver. An IVI-C specific driver is responsible for implementing the operations of functions and attributes and for communicating with the instrument. The IVI-C specific instrument driver contains the information for controlling the instrument, including the command strings, parsing code, and valid ranges of each instrument setting.

4.2.3.2 Sessions

When using an IVI-C driver, an application program creates and initializes an instrument driver session in a single call to the Initialize function. The application program closes and destroys the instrument driver session by calling the Close function.

IVI-C drivers use unique integer handles of type `ViSession` to identify an instrument driver session. The Initialize function returns the handle that application programs use to reference the instrument driver session in subsequent calls to instrument driver functions.

4.2.3.3 Interchangeability

Interchangeability for IVI-C drivers is achieved through IVI-C class drivers. An application program makes calls to an IVI-C class driver, which, in turn, dynamically loads the IVI-C class-compliant specific driver that the user specifies in the IVI configuration store. The IVI-C class driver communicates through the IVI-C specific driver to control the instrument.

By using the IVI-C class driver API in the application program, the user can interchange IVI-C specific instrument drivers and corresponding instruments without affecting test code. When using an IVI-C class driver, the user designates which IVI-C specific driver to use by specifying a logical name. The user configures the logical name in the IVI configuration store. Refer to Section 2.9.2.3, *How Interchangeability Works in C*, for more details on how users achieve interchangeability without recompiling or re-linking.

4.2.3.4 Accessing Instrument Specific Functions after Class Driver Initialization

An application program that uses an IVI-C class driver can also access instrument specific functionality by obtaining the session handle for the IVI-C specific driver. After initializing the IVI-C class driver session, the application program can call the Get Specific Driver C Handle function on the class driver session to obtain the handle to the specific driver session. The application program then uses this handle to call functions directly in the IVI-C specific driver. Typically, the application uses the IVI-C specific driver session handle only to call instrument specific functions or access instrument specific attributes.

4.2.3.5 Accessing Specific Drivers Directly

Application programs may use an IVI-C specific driver without the presence of an IVI-C class driver. In this case, the application program opens the session through the IVI-C specific driver. Typically, a user takes this approach when interchangeability is not a requirement or when programming to an IVI-C custom specific driver.

4.2.3.6 Leveraging VXI*plug&play* Driver Standards

The IVI-C architecture leverages the architecture standards defined by the VXI*plug&play* Alliance. Some IVI-C requirements remain the same as those defined by the VXI*plug&play* Alliance, such as function panel format and sub file format. In these instances, the appropriate sections provide references to the VXI*plug&play* specifications. Other requirements, such as error handling and naming formats, build on the existing VXI*plug&play* specifications. For these requirements, the IVI Foundation redefines the rules and conventions. Section 5.12, *IVI Error Handling*, provides the error handling requirements for all IVI drivers. Refer to Section3.6, *IVI-C* Requirements, and *IVI-3.4: API Style Guide* for conventions regarding naming and function prototypes for IVI-C drivers.

4.2.4 Use of C Shared Components

This section describes how IVI-C drivers use C shared components.

Refer to *IVI-3.9: C Shared Components Specification* for more details on the C Shared Components APIs.

4.2.4.1 Creating and Destroying Sessions

IVI-C drivers use the Session Management API to create and destroy instrument driver sessions. The IVI-C driver calls the `IviSession_New` function during initialization to create a session. The IVI-C driver then uses the `IviSession_SetDataPtr` function to associate a pointer with the session handle. The pointer provides access to the instrument driver data that is specific to the particular session. Subsequent function calls use the `IviSession_GetDataPtr` function to retrieve the pointer to the session data.

When an application program calls the Close function on the IVI-C driver, the IVI-C driver destroys the session by calling the `IviSession_Dispose` function. The IVI-C driver is responsible for deallocating the instrument driver data that is specific to the session.

When an application program uses an IVI-C class driver, the IVI-C class driver and the underlying IVI-C specific driver create separate sessions. The IVI-C class driver stores the handle for the specific driver session in the class driver session data. The application can obtain the specific driver session handle by calling the Get Specific Driver C Handle function, which is defined in *IVI-3.2: Inherent Capabilities Specification*.

4.2.4.2 Dynamic Driver Loading

Application programs can use IVI-C class driver functions without referencing IVI-C specific drivers in the source code. To make this possible, IVI-C class drivers dynamically load IVI-C specific drivers at run time. IVI-C class drivers use the Dynamic Driver Loader API for this purpose.

The application program passes a logical name to the Initialize function. This function traverses the IVI configuration store to find the IVI-C specific driver that is associated with the logical name. The IVI-C class driver passes the IVI-C specific driver's module path to the `IviDriverLoader_New` function. This function then loads the specific driver.

4.2.4.3 Function Pass-Through

Each IVI-C class driver function that the application program calls acts as a pass-though layer to an IVI-C specific driver function. When the application program invokes an IVI-C class driver function, the class driver obtains the address of the corresponding specific driver function by calling the `IviDriverLoader_GetFunctionPtr` function. If the specific driver does not export the function, the `IviDriverLoader_GetFunctionPtr` function returns a null pointer for the address. Otherwise, the IVI-C class driver function uses the address to call the associated IVI-C specific driver function. The class driver function passes to the specific driver all the parameters that the user passed to the class driver.

An alternative approach is for the IVI-C class driver to call the `IviDriverLoader_GetFunctionPtr` function for each user-callable function during initialization. The IVI-C class driver then stores the addresses of the user-callable functions of the specific driver in the class driver session data. The IVI-C class driver uses the stored address to call the specific driver when the user invokes an IVI-C class driver function.

4.2.4.4 Multithread Locking

IVI-C drivers allow application programs to use the same session in multiple threads. IVI-C drivers accomplish this by locking session resources while a call on the session is active. IVI-C driver functions that take an IVI session handle as an input parameter lock the IVI session on entry using the `IviSession_Lock` function and unlock it on exit using the `IviSession_Unlock` function. An exception to this is the Close function, which calls the `IviSession_Unlock` function prior to calling the `IviSession_Dispose` function.

4.2.4.5 Error Handling

Each IVI driver sets and retrieves errors in a consistent manner. When an error condition occurs, the IVI-C driver passes an error code and an error description string to the `IviSession_SetError` function. The IVI-C driver function also returns the error code as the return value of the function. When handling the error, the application program can retrieve the error code and description by calling the IVI-C driver's Get Error function, which, in turn, calls the `IviSession_GetError` function.

To create an error description string, an IVI-C driver may use the `IviErrorMessage_Get` and the `IviErrorMessage_FormatWithElaboration` functions.

4.2.5 Repeated Capabilities

Repeated capabilities may be represented in two ways in IVI-C drivers. Repeated capability instances may be specified by a string parameter to each function that accesses the repeated capability (parameter style) or by a function that selects the active instance (selector style). Section 12, *Repeated Capabilities*, in *IVI-3.4: API*

Style Guide, describes the two approaches and contains guidelines for choosing between them. In general, the parameter style is prevalent in IVI-C drivers.

IVI-C drivers provide functions to discover the string names for repeated capability instances. These functions take a one-based index.

Refer to Section 4, *Repeated Capability Group,* in *IVI-3.3: Standard Cross-Class Capabilities Specification,* for the specific API requirements for each approach. Both ways of representing repeated capabilities provide the user with the ability to navigate through repeated capability hierarchies.

Refer to Section 4.4, *Repeated Capability Selectors*, for information on how users specify a repeated capability instance within a hierarchy and a set of repeated capability instances.

4.2.6 Accessing Attributes

IVI-3.2: Inherent Capabilities Specification defines a set of attribute accessor functions for setting and getting attribute values using IVI-C drivers. To provide for type safety, *IVI-3.2: Inherent Capabilities Specification* defines a separate attribute accessor function for each data type. The generic names for these sets of functions are Set Attribute <type> and Get Attribute <type>.

4.2.6.1 Repeated Capabilities for Attributes

The attribute accessor functions include a repeated capability selector parameter (called `ChannelString`) for use with channel-based attributes or attributes of repeated capabilities. When using attribute accessor functions on attributes that do not apply to repeated capabilities, application programs pass `VI_NULL` or an empty string for the parameter.

4.2.7 Include Files

An include file for an IVI driver contains the following:

- C prototypes for all functions that the driver exports.
- C constant definitions for all attributes and attribute values that the driver exports.
- C constant definitions for all status codes that the driver can return.

The functions, attributes, attribute values, and status codes that an IVI-C driver exports may be inherent, class-defined, or instrument specific.

The following are examples of how the naming of inherent and class-defined attributes and status codes are handled in the include file for an IVI-C driver.

- The definition of an inherent attribute (or attribute value) in an IVI driver include file is the same as the definition for the attribute in *IVI-3.2: Inherent Capabilities Specification*, except that *`PREFIX`* in the constant name is replaced by the valid driver prefix. For example, if the following definition is in *IVI-3.2: Inherent Capabilities Specification*,

  ```
  #define PREFIX_ATTR_RANGE_CHECK      IVI_INHERENT_ATTR_BASE + 2
  ```

  then the following definition of the Range Check attribute appears in the include file for an IVI specific driver for the Agilent 34401A:

  ```
  #define AG34401A_ATTR_RANGE_CHECK    1050002
  ```

- The definition of a class-defined attribute (or attribute value) in an IVI driver include file is the same as the definition for that attribute or attribute value in the IVI specification for that class, except that class prefix in the constant name is replaced by the driver prefix. For example, if the following definition is in *IVI-4.2: IviDmm Class Specification*,

```
#define IVIDMM_ATTR_TRIGGER_SOURCE    250004
```

then the following definition of the Trigger Source attribute appears in the include file for an IVI specific driver for the Agilent 34401A:

```
#define AG34401A_ATTR_TRIGGER_SOURCE  250004
```

- The definition of a class-defined status code in an IVI driver include file is the same as the definition for that status code in the IVI specification for that class, except that class prefix in the constant name is replaced by the driver prefix. For example, if the following definition is in *IVI-4.2: IviDmm Class Specification*,

```
#define IVIDMM_WARN_OVER_RANGE      0x3FFA2001
```

then the following definition of the Over Range warning appears in the include file for an IVI specific driver for the Agilent 34401A:

```
#define AG34401A_WARN_OVER_RANGE    0x3FFA2001
```

The inherent status codes defined in *IVI-3.2: Inherent Capabilities Specification* do not begin with a replaceable prefix. An example is `IVI_ERROR_INVALID_VALUE`. The include file for an IVI-C driver does not provide instrument specific versions of inherent status code names. Instead, it includes a common include file that defines those names.

Notice that by providing driver specific C constants for class-defined attributes, attribute values, and status codes, as well as driver specific function names, an IVI-C class-complaint specific driver can be used without reference to class names. An application program developer can use the class-defined features of the driver as if they were instrument specific. Developers do not have to understand which features are class-defined and which are instrument specific.

An application program developer who uses class drivers can use the C constants from the class driver include file for class-defined features and C constants from the specific driver include file for instrument specific features. The constant and function names for the class-defined features have class prefixes whereas the constant and function name for the instrument specific features have instrument prefixes. This helps the developer identify the non-interchangeable portions of the application program.

All include files for IVI-C drivers define constants as macros.

Note: An IVI driver supplier may define functions, attributes, attribute values, and status codes that are common among multiple drivers. Such functions, attributes, attribute values, and status codes are referred to as *vendor specific*. Unless otherwise noted, vendor specific functions, attributes, attribute values, and status codes are treated as instrument specific in the IVI specifications.

4.2.8 Interactive Development Interface

An interactive development interface is a tool that allows users to operate an instrument driver function interactively. Interactive operation of an instrument driver function helps the user understand the behavior of the function, the function prototype, and the meanings and valid values of parameters. The interactive development interface for IVI-C drivers is a set of *function panels*.

A function panel presents an instrument driver function graphically, with help text, and allows the user to execute the function. An IVI-C driver organizes its function panels in a hierarchy to assist users in locating functions.

Function panels are not available in all programming environments. For environments that do not support function panels, documentation such as Windows help files help users learn how to use driver functions.

Function panels are also required for VXI*plug&play* drivers. IVI-C function panels are consistent in format and style with VXI*plug&play* function panels.

Each IVI-C driver defines its interactive development interface in one *function panel* (`.fp`) file and one *sub* (`.sub`) file. The function panel contains information on each function that the driver exports, including the parameters of each function. The sub file contains the information on each attribute the user can access through the Set Attribute <type> and Get Attribute <type> functions.

4.2.8.1 Function Panel File

A function panel (`.fp`) file contains the following information:

- a function tree, which represents the function hierarchy
- data types of each function parameter and return value
- size and placement of function parameters and return values on each panel
- help documentation for each function and parameter

Refer to Section 6, *Function Panel File Format*, in the VXI*plug&play* specification *VPP-3.3: Instrument Driver Interactive Developer Interface Specification* for information on the function panel file format.

4.2.8.2 Function Hierarchy

The function hierarchy provides valuable information to the user. IVI-C drivers that conform to standard function hierarchies makes it easier for users to understand new function hierarchies more quickly. Insofar as is practical, the function hierarchies of IVI class-compliant instrument specific drivers reflect the hierarchies defined in the class specifications.

Section 4.3, *C Inherent Capabilities*, in *IVI-3.2: Inherent Capabilities Specification*, specifies the hierarchy for the IVI inherent functions.

Each IVI class specification specifies a function hierarchy for the class-defined functions.

Section 13.1, *C Function Hierarchy*, in *IVI-3.4: API Style Guide*, contains guidelines on grouping functions into a hierarchy.

4.2.8.3 Sub File

A sub (`.sub`) file describes the attributes that users can access through the Set Attribute <type> and Get Attribute <type> functions. In particular, a sub file contains the following information:

- an attribute hierarchy, which organizes attributes into logical groups
- the name and data type of each attribute
- the valid values for enumerated attributes
- help documentation for each attribute and attribute value

Refer to Section 7, *Function Panel Sub File Format*, in the VXI*plug&play* specification *VPP-3.3: Instrument Driver Interactive Developer Interface Specification* for information on the sub file format.

4.2.8.4 Attribute Hierarchy

The attribute hierarchy provides valuable information to the user. IVI-C drivers that conform to standard attribute hierarchies make it easier for users to understand new attribute hierarchies more quickly. Insofar as is practical, the attribute hierarchies of IVI class-compliant instrument specific drivers reflect the hierarchies defined in the class specifications.

Section 4.3, *C Inherent Capabilities*, in *IVI-3.2: Inherent Capabilities Specification*, specifies the hierarchy for the IVI inherent attributes.

Each IVI class specification specifies an attribute hierarchy for the class-defined attributes.

Section 13.1, *C Attribute Hierarchy*, in *IVI-3.4: API Style Guide*, contains guidelines on grouping attributes into a hierarchy.

4.2.9 Instrument Specific Direct I/O API

A specific IVI-C driver for device(s) that use message-based communication includes two functions for reading and writing data and a timeout attribute for controlling the I/O timeout. It may also optionally include an attribute that provides access to the driver's underlying I/O.

Both the function and attribute hierarchies of specific IVI-C drivers for devices that use message-based communication include a level 1 category named `System`. Functions or attributes related to direct I/O, including the read and write functions and the timeout and session attributes, shall be placed in this level of the hierarchy.

4.3 IVI.NET Driver Architecture

This section describes how IVI.NET instrument drivers use .NET technology. This section does not attempt to describe the technical features of .NET, except where necessary to explain a particular IVI.NET feature. This section assumes that the reader is familiar with .NET technology.

IVI.NET drivers implement the IVI instrument driver features described in Section 1.7, *Substitutions*

This specification uses paired angle brackets to indicate that the text between the brackets is not the actual text to use, but instead indicates the kind of text that can be used in place of the bracketed text. Sometimes the meaning is self-evident, and no further explanation is given. The following list includes those that may need additional explanation for some readers.

- <ClassName>: The name of an IVI instrument class as defined by an IVI Instrument Class specification. For example, "IviDmm".
- <ClassType>: The name of an IVI instrument class as defined by an IVI Instrument Class specification, without the leading "Ivi". For example, "Dmm".
- <ComponentIdentifier>: For IVI-COM and IVI.NET, the string returned by a specific driver's Component Identity attribute. This string uniquely identifies the driver. For example, "Agilent34410".
- <Prefix>: For IVI-C class drivers, the string returned by the driver's Class Driver Prefix attribute. The class driver prefix will commonly be an IVI class name, but may be different. For example, "IviDmm". For IVI-C specific drivers, the string returned by the driver's Specific Driver Prefix attribute. For example, "NI3456"
- <CompanyName>: The name of the driver vendor (not the instrument manufacturer). For example, "Agilent Technologies, Inc".
- <ProgramFilesDir>: The Windows program files directory. This varies across different versions of Windows. In some contexts, it is not intended to differentiate between the 64-bit and 32-bit program files directories found on 64-bit versions of Windows that include Windows On Windows (WOW), but to be understood as a generic reference to the program files directory.
- <ProgramDataDir>: The Windows data directory. This varies across different versions of Windows. It is generally understood to apply to all users.
- <IviStandardRootDir>: The root install directory for the IVI Shared Components, which consists of executables and other files needed to create and run IVI drivers. By default, this directory is "<ProgramFilesDir>\IVI Foundation\IVI".

- <RcName>: The name of a repeated capability. Repeated capabilities may be defined in class specs or by specific driver developers.
- <FwkVerShortName>: The IVI.NET short name for a version of the .NET Framework.

Where it is important to indicate the case of substituted text, casing is indicated by the case of the text between the brackets.

- <ClassName> indicates Pascal casing. For example, “IviDmm”.
- <className> indicates camel casing. For example, “iviDmm”
- <classname> indicates all lower case. For example, “ividmm”
- <CLASSNAME> indicates all upper case. For example, “IVIDMM”
- <CLASS_NAME> indicates all upper case with underscores between words. For example, “IVI_DMM”.

Features and Intended Use of IVI Drivers, and present them to test system programmers in the form of .NET interfaces.

4.3.1 Target .NET Framework Versions

IVI.NET drivers require the full version of Microsoft .NET Framework; neither .NET Framework Client Profiles nor .NET Core are sufficient. The minimum version of Microsoft .NET Framework that the IVI.NET driver supports cannot be less than 2.0, and the recommended version is 4.0. Table 4-2 lists the relevant framework versions, along with the full version number and the IVI.NET Framework version short name, *<FwkVerShortName>*.

Table 4-2. .NET Framework Versions

.NET Framework Version	Full Version	<FwkVerShortName>
2.0	v2.0.50727	Fx20
3.0	v3.0	Fx30
3.5	v3.5	Fx35
4.0 (Recommended)	v4.0.30319	Fx40
4.5	v4.5.50709	Fx45

4.3.1.1 IVI.NET Framework Version Short Name

The IVI.NET Framework version short name, *<FwkVerShortName>*, is used to provide .NET Framework version-specific names for registry keys, Software Module Table entries, and Start Menu folders. The format is Fx*<framework major version><framework minor version>*.

4.3.2 Target Operating Systems

Refer to Section 2.16, *Operating Systems*.

4.3.3 Target Languages and Application Development Environments

IVI.NET drivers work in the target languages and application development environments listed in Table 4-3.

Table 4-3. Target languages and ADEs for IVI.NET drivers

32-bit	**64-bit**
Agilent VEE	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

* Note: The intent is to support the 64-bit versions of these ADEs when they are available.

IVI.NET drivers comply with the Common Language Specification (CLS), so that in principle, IVI.NET drivers can work in other development environments in which the .NET CLR is supported.

4.3.4 IVI.NET Driver Overview

An IVI.NET instrument driver is instantiated as a .NET class accompanied, optionally, by .NET helper classes. An IVI.NET driver class implements a variety of interfaces and classes including standard IVI interfaces and instrument specific classes and interfaces.

Standard IVI interfaces include IVI inherent interfaces and IVI class-compliant interfaces for each of the defined IVI instrument classes. Standard IVI interfaces provide a standard syntax through which application programs interact with the driver. This standard syntax, when used in combination with one of the IVI.NET session factory methods, provides the same level of syntactical interchangeability in the IVI.NET architecture as that provided by IVI-COM and by IVI class drivers in the IVI-C architecture. Thus, IVI class drivers are not required for syntactical interchangeability in the IVI.NET architecture. Refer to Section 2.9.2.2, *How Interchangeability Works in COM and .NET,* for more details on achieving interchangeability using IVI.NET drivers.

Instrument specific classes and interfaces provide access to instrument specific functionality. Typically, instrument specific classes and interfaces mirror IVI class-compliant interfaces for the instrument features that are within the scope of the functionality defined by the IVI class specifications. However, instrument specific classes and interfaces also include additional methods and properties that provide access to the features that are beyond the scope of the IVI class specifications.

4.3.5 IVI.NET Interfaces

The methods and properties of IVI.NET drivers are grouped into multiple interfaces based on functionality. This grouping allows for a natural hierarchical structure that organizes the overall driver functionality.

All IVI.NET drivers contain the interfaces for the IVI inherent features as well as interfaces that implement the instrument specific capabilities of the instrument. IVI.NET class-compliant drivers also contain IVI class-compliant interfaces. The interfaces for the IVI inherent features are defined in *IVI-3.2: Inherent Capabilities Specification*. The IVI.NET class-compliant interfaces are defined in the IVI class specifications.

The IVI class-compliant interfaces for a particular instrument class are identical from driver to driver. The interfaces that implement the IVI inherent methods and properties are identical for all IVI.NET drivers. Keeping the interfaces identical is what makes the drivers syntactically interchangeable.

Instrument specific classes and interfaces are necessarily different from one instrument to another, whereas class-compliant interfaces are identical for all drivers within the same class. Thus, the class-compliant interfaces in an IVI.NET driver are different from the driver's instrument specific classes and interfaces. The IVI class-compliant interfaces may be thin layers of code that call instrument specific classes and interfaces.

This allows the instrument specific classes and interfaces to leverage the syntax of the class-compliant interfaces.

One of the ways that IVI.NET differs from IVI-COM is that IVI.NET drivers may expose instrument specific *classes* in scenarios where IVI-COM would expose *interfaces*. This is a subtle difference, but allows the IVI.NET driver developer more implementation flexibility in the instrument specific API.

4.3.6 Navigating IVI.NET Hierarchies

There are three cases to consider.

1. Navigating from class-compliant API to instrument specific API and vice versa – To accomplish this, it is necessary to use IServiceProvider.GetService() to navigate from one set of interfaces to the other.

2. Navigating from one IVI class-compliant API to another IVI class-compliant API in the same driver – To accomplish this, it is necessary to use IServiceProvider.GetService() to navigate from one set of interfaces to the other.

3. Navigating within either the class-compliant or instrument specific API – To accomplish this, it is best to use interface reference properties (properties that return a reference to another class or interface). In some cases it is possible to cast from one interface to another, but this may or may not succeed depending on implementation.

Interface reference properties create a hierarchy that the user can easily navigate. C# or VB.NET can navigate through an arbitrary number of reference properties. Consider the following code:

```
Itf1.Itf2.Itf3.Itf4.Method
```

`Itf1` is a reference to a class or interface referenced by the variable `Itf1`. Similarly, `Itf2`, `Itf3`, and `Itf4` are also reference properties. As the user types each of these names, IntelliSense displays a dropdown list of methods and properties in the corresponding class or interface. After typing `Itf1` followed by a period, a list of all the properties and methods in `Itf1` appears, allowing the user to select one. After selecting `Itf2` and typing the period, a list of the methods and properties in `Itf2` appears, and so on, until the developer selects `Method`.

In addition to the driver's main class, IVI.NET drivers may implement additional classes. For example, IVI.NET drivers include a class for each repeated capability collection. Application programs must access the helper object through a reference property. Refer to Section 4.1.9, *Repeated Capabilities*, for more information on IVI.NET collections.

IVI.NET drivers make extensive use of reference properties to navigate to other classes and interfaces supported by the driver. Again, note that IVI.NET drivers may reference classes directly using reference properties in the instrument specific classes and interfaces, a feature which allows IVI.NET driver implementations to be a bit more flexible. In the inherent and class-compliant APIs, only interfaces are allowed (in order to provide syntactical interchangeability) and so all reference properties are interface reference properties.

4.3.7 Interface Hierarchy

IVI.NET drivers take advantage of reference properties to organize classes and interfaces hierarchically. Each class or interface has exactly one parent class or interface and zero or more child classes or interfaces. No circular references or series of references exist. The hierarchy is primarily organized by functionality, while also being consistent with .NET conventions where possible. The hierarchy for inherent features is documented in the *IVI-3.2: Inherent Capabilities Specification*. The hierarchies for class-compliant interfaces are documented in the corresponding IVI class specifications.

IVI.NET class-compliant interface hierarchies are not necessarily organized according to the capability groups defined in the IVI class specifications. IVI.NET interface hierarchies are similar to IVI-COM hierarchies, but do not necessarily correspond to the IVI-C function tree (`.fp` file) hierarchies.

4.3.8 Data Types

IVI.NET APIs are restricted to .NET Common Language Specification (CLS) compliant data types.

4.3.8.1 Enumerations

Enumerations for IVI.NET drivers are strongly typed. Two enumerations that otherwise could refer to the same attribute but have a different set of enumeration values are typed differently.

For instance, all IVI.NET drivers that comply with the IviDmm specification have an enumeration for the Function property. However, not all DMMs support the same function values. Drivers for DMMs with different sets of function values have different instrument specific enumerations for the Function property. Furthermore, unless a DMM has exactly the same set of function values as the set defined for the class, the instrument specific enumeration is distinct from the class enumeration.

4.3.9 Repeated Capabilities

Repeated capabilities may be represented in two ways in IVI .NET drivers. Repeated capability instances may be specified by a method that selects the active instance (the *selector style*) or by selecting a particular instance from an IVI .NET collection (the *collection style*).

IVI.NET repeated capability collection interfaces must derive from Ivi.Driver.IIviRepeatedCapabilityCollection<T>, where T is the type of the collection member class or interface. IVI.NET interfaces that represent instances of a repeated capability (i.e., collection members) must derive from Ivi.Driver.RepeatedCapabilityIdentification. Refer to Section 11, *Repeated Capability Collection Base Interfaces*, in *IVI-3.18: IVI.NET Utility Classes and Interfaces Specification*, for a full description of these interfaces.

Refer to Section 4, *Repeated Capability Group*, in *IVI-3.3: Standard Cross-Class Capabilities Specification*, for the specific API requirements for the selector and collection styles. Both ways of representing repeated capabilities provide the user with the ability to navigate through repeated capability hierarchies.

Refer to Section 4.4, *Repeated Capability Selectors*, for information on how users specify a repeated capability instance within a hierarchy and a set of repeated capability instances.

Note: IVI-C drivers often use repeated capability name parameters to each method or attribute that accesses the repeated capability (the *parameter style*). Use of the parameter style is discouraged in in IVI.NET APIs because parameterized properties are not allowed in CLS compliant .NET code.[4] A class specification may use collections in the IVI .NET API and the string parameter approach in the IVI-C API. Section 12, *Repeated Capabilities*, in *IVI-3.4: API Style Guide,* describes each approach and contains guidelines for choosing among the different alternatives.

4.3.10 Session

Session parameters are not used in IVI.NET methods and properties. Object identity serves the same purpose in IVI.NET as the session does in IVI-C.

[4] The Fgen IVI.NET API uses this technique, but only to provide compatibility with the original Fgen capability class design.

4.3.11 Multithread Locking

IVI.NET drivers offer three modes of multithread locking. The mode used by an IVI.NET driver is determined by the parameters passed to the IVI.NET driver constructor. The type of locking used by an IVI.NET driver instance is established at construction time and cannot be changed during the lifetime of the driver instance. Each mode of locking is explained in the sections below.

4.3.11.1 Per-Instance Locking

This level of multithread locking is required for IVI.NET drivers.

Per-instance locking ensures that access to the same instance of an IVI.NET driver is synchronized between multiple threads in the same AppDomain. Threads accessing different instances of the same IVI.NET driver are not synchronized, nor are threads accessing the driver in different AppDomains. This means that multi-threaded applications that wish to synchronize access to a driver must take care to share a single instance amongst threads, rather than creating different instances on different threads.

Per-instance locking is used when the client application invokes the IVI.NET driver constructor with LockType.AppDomain specified for the lockType parameter and an empty string supplied for the accessKey parameter.

4.3.11.2 AppDomain-Wide Locking

This level of multithread locking is optional for IVI.NET drivers.

AppDomain-wide locking ensures that all instances of an IVI.NET driver created with the same access key are protected from simultaneous access by all threads within an AppDomain. This means that client applications that wish to synchronize access to a driver can do so even in the face of of multiple instances, so long as the instances that need to be synchronized share the same access key. Threads accessing instances of the IVI.NET driver from different AppDomains are not synchronized, nor are threads accessing instances that were created with different access keys.

The access key is user-specified and serves as a lock identifier for the lock that must be obtained before a thread can invoke a driver function. There are no specific requirements regarding the format or content of the access key. However, memory is allocated and may not be freed for each unique access key used. Thus, it is not good practice to generate arbitrarily unique access keys. Instead, users should generate a key that logically represents the resource being locked. For example, the access key may correspond to the I/O resource name , such as a VISA resource descriptor.

AppDomain-wide locking is used when the client application invokes the IVI.NET driver constructor with LockType.AppDomain specified for the lockType parameter and a non-empty string supplied for the accessKey parameter.

Note that drivers that implement AppDomain-wide locking must preserve the integrity of driver state data between multiple instances of the driver. This may include requiring the client to reset the instrument to a known state each time they are granted access to the driver.

4.3.11.3 Machine-Wide Locking

This level of multithread locking is optional for IVI.NET drivers.

Machine-wide locking ensures that all instances of an IVI.NET driver created with the same access key are protected from simultaneous access by all threads within all AppDomains and processess on the same machine.

As with AppDomain-wide locking, the access key is user-specified and serves as a lock identifier for the lock that must be obtained before a thread can invoke a driver function. There are no specific requirements regarding the format or content of the access key. For example, the access key may correspond to the I/O resource name, such as a VISA resource descriptor.

Machine-wide locking is used when the client application invokes the IVI.NET driver constructor with LockType.Machine specified for the lockType parameter and a non-empty string supplied for the accessKey parameter.

Note that drivers that implement machine-wide locking must preserve the integrity of driver state data between multiple instances of the driver. This may include requiring the client to reset the instrument to a known state each time they are granted access to the driver.

4.3.12 Class and Interface Requirements

IVI.NET instrument drivers expose three kinds of features. Inherent capabilities are features common to all drivers. Class-compliant interfaces are common to all drivers of a particular instrument class. IVI.NET specific instrument drivers also may expose instrument specific functionality through classes and interfaces that are specific to that driver.

4.3.12.1 Naming and .NET Namespaces

IVI.NET drivers have their own namespaces; therefore names do not need to contain as much distinguishing information as do IVI-C or IVI-COM names. For example, in IVI-C and IVI-COM instrument specific APIs, the component identifier is needed to distinguish class, interfaces, and enumeration names across multiple drivers. In IVI.NET the component identifier is not needed to distinguish class, interfaces, and enumeration names.

4.3.12.2 Inherent Features

Every IVI.NET driver implements a set of interfaces that expose the IVI inherent features as described in *IVI-3.2: Inherent Capabilities Specification*. An IVI.NET custom driver does not implement any other standard IVI interfaces. IIviDriver is the root of the IVI inherent interfaces. When an IVI.NET driver is instantiated, a legal cast to IIviDriver always returns a valid reference.

The hierarchy of IVI.NET inherent interfaces is described in Section 4.1, *.NET Inherent Capabilities*, in *IVI-3.2: Inherent Capabilities Specification*. The namespace for the IVI.NET inherent interfaces is Ivi.Driver.

4.3.12.3 Class-Compliant Interfaces

Every IVI.NET class-compliant driver implements a set of interfaces that export the IVI class-compliant features defined in the corresponding IVI class specification. I<*ClassName*> is the root of this set of interfaces. For example, IIviScope is the root of the interfaces that the IviScope class specification defines.

I<*ClassName*> extends IIviDriver. Interface reference properties to other class-compliant interfaces are included in I<*ClassName*> to provide access to the class-compliant interface hierarchy. In rare cases, commonly used methods and properties for accessing the instrument may also be included in I<*ClassName*>.

When an IVI.NET class-compliant driver is instantiated, calling IServiceProvider.GetService() with *typeof* I<*ClassName*> always returns a valid reference.

IVI.NET drivers may implement class-compliant interfaces for multiple instrument classes.

The namespace for an IVI class-compliant API is Ivi.<*ClassType*>. Note that the Ivi.<*ClassType*> in the namespace name and the <*ClassName*> in the interface names are redundant. <ClassName> has been retained in IVI.NET class-compliant interfaces to keep continuity with IVI-COM interface names. <*ClassName*> need not appear in other class type definitions, including exceptions, supporting classes, and enumeration names.

4.3.12.4 Instrument Specific Classes and Interfaces

The namespace for instrument specific drivers is *<CompanyName>.<ComponentIdentifier>*, where *<CompanyName>* is the name of the driver vendor. Since *<ComponentIdentifier>* is part of the namespace name, it need not be used in IVI.NET in names where it would be used in IVI-COM.

One of the ways that IVI.NET differs from IVI-COM is that IVI.NET drivers may expose instrument specific *classes* in scenarios where IVI-COM would expose *interfaces*. This is a subtle difference, but allows the IVI.NET driver developer more implementation flexibility in the instrument specific API.

IVI.NET instrument specific interface names must begin with I. For example, an instrument specific trigger interface for the Agilent 34401 DMM could be named ITrigger.

IVI.NET class names should follow Microsoft's published naming guidelines for classes. There are no other requirements for IVI.NET instrument specific class names other than the class at the root of the instrument specific reference hierarchy. The root of an IVI.NET reference hierarchy is always a class named *<ComponentIdentifier>*. For example, an instrument specific trigger class for the Agilent 34401 DMM could be named Trigger, but the root class would be Agilent34401.

Insofar as is practical, the instrument specific interfaces of an IVI class-compliant driver reflect the syntax of the corresponding class-compliant interfaces.

4.3.12.5 Repeated Capability Interfaces

IVI.NET interfaces that represent a single instance of a repeated capability consist of the appropriate prefix, as described in the previous two sections, followed by the the name of the repeated capability. For class-compliant interfaces, this is *I<ClassName><RcName>*. For instrument specific interfaces, this is *I<ComponentIdentifier><RcName>*. For example, "IIviPwrMeterChannel" or "IAgilent34410Trace".

IVI.NET repeated capability collection interfaces consist of the appropriate prefix, as described in the last two sections, followed by the name of the repeated capability, followed by "Collection". For class-compliant interfaces, this is *I<ClassName><RcName>Collection*. For instrument specific interfaces, this is *I<ComponentIdentifier><RcName>Collection*. For example, "IIviPwrMeterChannelCollection" or "IAgilent34410TraceCollection".

4.3.12.6 Instrument Specific Direct I/O API

A specific IVI.NET driver for device(s) that use message-based communication includes a System interface with methods for reading and writing string and bytes and a property for setting the I/O timeout. It may optionally include a property that provides access to the driver's underlying I/O.

The root interface of such an IVI.NET driver includes a reference to the System interface.

4.3.13 Standard Inherent and Class Assemblies

To allow users to swap instruments without recompiling or re-linking, the IVI Foundation publishes .NET assemblies for standard IVI.NET API definitions. One assembly contains the IVI inherent interfaces, along with the IVI.NET utility classes and interfaces, and one assembly exists for each class specification.

4.3.14 Versioning .NET Interfaces

The IVI.NET inherent and class-compliant assemblies are not changed or deleted after being published in *IVI-3.2: Inherent Capabilities Specification* or the corresponding class specification. The only exception is that new members may be added to enumerations..

When the IVI Foundation approves a class specification or the specification for inherent capabilities, it also approves the corresponding .NET assembly.[5] After the IVI Foundation distributes an assembly, the interfaces that the assembly defines are not changed.

New versions of standard IVI.NET assemblies include policy files so that programs written to access the old version(s) of the assemblies will also work with the new versions.

4.3.15 Driver Classes

Note: In this section, *class* refers to a .NET class rather than an IVI instrument class.

IVI.NET specific drivers may consist of more than one class. In fact, multiple classes are necessary if the driver implements IVI repeated capability collections. The user instantiates the *main* driver class. The main IVI driver class is named <*ComponentIdentifier*>. Once the client program has a reference to the main IVI driver class, the IIviDriver interface and the root interface of any class-comliant hierarchy that the driver implements can be accessed using IServiceProvider.GetService().

Driver classes are packaged as .NET assembly DLLs.

4.3.16 IVI.NET Error Handling

IVI.NET specific drivers report errors by throwing exceptions. Warnings are reported by a .NET Warning event defined in the inherent capabilities.

In general, existing .NET exceptions are used when an appropriate one exists, in order to reduce the number of exceptions that are specific to IVI inherent capability, instrument class, and instrument specific interfaces. However, some standard IVI exceptions are used for reporting errors from the underlying I/O software and the Configuration Server. IVI.NET exceptions are derived from the System.Exception class. Refer to Section 5.12.2, *IVI.NET Error Handling*, for details of IVI.NET error handling.

Warnings are reported by a .NET Warning event defined in the inherent interfaces. Refer to Section 11.1, *IVI.NET*, in *IVI-3.4: API Style Guide*, for details of IVI warnings. Refer to Section 9, *IVI.NET Event Descriptions*, in *IVI-3.2: Inherent Capabilities Specification*, for the definition of the warning event.

4.3.17 Driver Packaging

Refer to Section 5.17.12, *Packaging*, for packaging requirements for IVI.NET drivers.

4.3.18 Choosing a Version of the IVI.NET Shared Components for Building the Driver

When a driver developer builds an IVI.NET driver against a particular version of the IVI.NET Shared Components, that version or a later version of the shared components must be installed on the end-user's system for the driver to function.

A driver developer may build an IVI.NET driver against the earliest version of the IVI.NET Shared Components that meets the requirements for the IVI.NET driver being developed, or against any later version. A driver built against an earlier version of the IVI.NET Shared Components will work with a greater number of subsequent IVI.NET Shared Component versions than a driver built against a later version of the IVI.NET Shared Components, and so is more likely to work with a version that the end-user may already have installed.

[5] The exception, of course, is specifications that were approved prior to the creation of the IVI.NET standards. For these specifications, IVI.NET material will be added and approved coincident with the approval of this specification.

With these facts in mind, driver developers are encouraged to build against the oldest version of the IVI.NET Shared Components that meets the requirements of their IVI.NET driver and their development process.

4.4 Repeated Capability Selectors

Repeated capabilities can be represented in three different ways in IVI APIs. The parameter style allows users to select repeated capabilities with parameters to every method and property that applies to the repeated capability, The selector style allows users to select repeated capabilities using methods that select the active instance(s). The collection style allows users to select one repeated capability from a collection. Not all APIs are capable of using all of the methods.

IVI-COM drivers can represent repeated capabilities using all three methods, but the parameter style is discouraged.

IVI-C drivers can represent repeated capabilities as the parameter style or the selector style, but not using the collection style.

IVI.NET drivers can represent repeated capabilities using all three methods, but the parameter style is discouraged.

For all approaches, users identify repeated capability instances using repeated capability selectors. This section describes the syntax and use of repeated capability selectors.

For the purpose of repeated capability selector syntax, the parameter style and selector style work in the same way. Accordingly, the rest of this section refers to the *parameter/selector* approach.

4.4.1 Simple Repeated Capability Selectors

To specify a single, non-nested repeated capability instance, a repeated capability selector consists of a single physical or virtual repeated capability identifier. The selector is the same regardless of whether the user is specifying a parameter or using a collection. However, the selector is used in different ways in the two approaches.

For example, to specify the physical identifier “chan1” to the Read Waveform function of the IVI-C API for the IviScope class, the user passes “chan1” as the second parameter.

```
ReadWaveform (vi, “chan1”, 1024, 5000, waveform, &count, &initX, &incrX);
```

To specify “chan1” to the Read Waveform method of the IVI-COM API for the IviScope class, the user passes “chan1” as the selector for the IVI-COM collection.

```
Measurements.Item(“chan1”).ReadWaveform (5000, waveform, initX, incrX);
```

To specify “chan1” to the Read Waveform method of the IVI.NET API for the IviScope class, the user passes “chan1” as the selector for the IVI.NET collection.

```
Measurements[“chan1”].ReadWaveform (waveform);
```

4.4.2 Representing a Set of Instances

To specify a set of repeated capability instances, repeated capability selectors use *repeated capability ranges* and *repeated capability lists*.

A repeated capability range consists of a *lower bound repeated capability identifier*, followed by a hyphen (-), followed by an *upper bound repeated capability identifier*. The range indicates all instances from the lower bound to the upper bound, inclusively. Each driver that allows repeated capability ranges specifies an ordering of the physical repeated capability identifiers that it defines. In a valid range, the lower bound identifier is less than or equal to the upper bound identifier.

A repeated capability list is a comma-separated sequence of repeated capability identifiers, a comma-separated sequence of repeated capability ranges, or a comma-separated list of identifiers and ranges. In a valid list, no identifiers repeat and no ranges overlap. However, identifiers may appear in any order. White space after commas is ignored. A repeated capability list may be enclosed within square brackets (`[]`).

The repeated capability identifiers used in ranges and lists may be physical identifiers or virtual identifiers.

The following selectors represent the same set of repeated capability instances:

```
    "1, 2, 3, 6, 8, 9, 10"

    "[1-3, 6, 8-10]"

    "1-3, 6-6, 8-10"

    "8-10, 3, 2, 1, 6"
```

The following selectors are invalid:

```
    "6-3"

    "1, 2, 1"

    "1-3, 3-5"
```

When used with as a function parameter, a repeated capability range or list is used in the same way as a simple selector. An example might be:

```
EnableChannel (vi, "1-10, 21-30");
```

When used with IVI-COM collections, a repeated capability range or list is used in the same way as a simple selector. An example might be:

```
Channels.Item("1-10, 21-30").Enable ();
```

IVI.NET collections do not use repeated capability ranges. Per the normal .NET collection syntax, collections are restricted to simple repeated capability identifiers.

The repeated capability identifiers used in ranges and lists may be physical identifiers or virtual identifiers.

4.4.3 Representing Nested Repeated Capabilities

The representation of nested repeated capabilities differs depending on whether the parameter/selector approach or the collection approach is being used.

4.4.3.1 Representing Nested Repeated Capabilities in the Parameter/Selector Approach

When using the parameter/selector approach for specifying nested repeated capabilities, all the information needed to navigate the hierarchy is represented in a single selector string. The repeated capability identifiers at each level in the hierarchy are concatenated using colons as separators. Each identifier may be physical or virtual. The identifier for the repeated capability instance at the top level of the hierarchy appears first in the string. White space around colons is ignored. Such selectors are called *hierarchical repeated capability selectors*.

As an example, consider a power supply with four output channels, each of which has two configurable external triggers. To configure a specific trigger, the user specifies the output channel and the trigger. A function call might look like the following:

```
ConfigureExternalTrigger ("Out1:Trig1", Source, Level);
```

where “`Out1:Trig1`” represents a specific trigger (Trig1) for a specific output (Out1), and where “`Trig1`” and “`Out1`” are physical identifiers for the respective repeated capability instances.

4.4.3.2 Representing Nested Repeated Capabilities in the Collection Approach

When using IVI-COM or IVI.NET collections to represent nested repeated capabilities, each level in the hierarchy is modeled as a separate collection. To select an item in the collection, the user identifies the instance of the repeated capability for that level only. Each collection in a hierarchy is accessed separately.

Consider the example described in the previous section. Using IVI-COM collections, the code might appear as follows:

```
Outputs.Item(“Out1”).Triggers.Item(“Trig1”).Configure(Source, Level);
```

Using IVI.NET collections, the code might appear as follows:

```
Outputs[“Out1”].Triggers[“Trig1”].Configure(source, level);
```

4.4.4 Mixing Hierarchy with Sets

Selectors for nested repeated capabilities may contain lists or ranges at any level of the hierarchy. Mixing hierarchy with lists or ranges is syntactically complex because it requires using the colon (`:`), comma (`,`), and hyphen (`-`) operators in the same selector. The interpretation of such a selector can be ambiguous unless the order of precedence is clear. The use of square brackets (`[]`) may be required to resolve ambiguity between the colon (`:`) and comma (`,`) operators.

The order of precedence is square brackets (`[]`), hyphen (`-`), colon (`:`), and comma (`,`). Each operator is evaluated from left to right.

For example, `“a1-a3:b2:[c5,c7]”` expands to the following list:

```
“a1:b2:c5, a1:b2:c7, a2:b2:c5, a2:b2:c7, a3:b2:c5, a3:b2:c7”,
```

whereas `“a1-a3:b2:c5,c7”` evaluates to

```
“a1:b2:c5, a2:b2:c5, a3:b2:c5, c7”.
```

Note: Although both examples are syntactically correct, only the first example is valid. All repeated capability identifiers within a list of repeated capability identifiers must have the same level of nesting after expansion.

4.4.5 Ambiguity of Physical Identifiers

This section discusses rules for preventing ambiguity in the physical identifiers defined by a driver.

4.4.5.1 Uniqueness Rules for Physical Identifiers

Each physical identifier must be unique:

- within a single repeated capability,
- across multiple repeated capabilities that are not nested, and
- across multiple repeated capabilities that are nested at the same level under the same parent repeated capability instance.

For purposes of this rule, repeated capability identifiers shall be case insensitive.

This rule does not apply to repeated capabilities that are nested under different parent repeated capability instances or that are nested at different levels in a repeated capability hierarchy. In these cases, repeated capabilities may use the same physical identifiers if the driver can reliably distinguish which repeated capability instance is intended. Normally the context in which a parameter or collection appears is sufficient for the driver to determine the intended repeated capability instance.

Table 4-4 contains a valid set of physical repeated capability identifiers for an IVI driver that has nested repeated capabilities. In this case, two trigger instances are named `trig1` but are nested under separate parents, `out1` and `out2`. Therefore, the `trig1` instances are unambiguous.

Table 4-4. Example of Unambiguous Nested Repeated Capabilities

First Level		Second Level	
Repeated Capability Name	Physical Name of Instance	Repeated Capability Name	Physical Name of Instance
Output	`out1`	Trigger	`trig1`
			`trig2`
			`trig3`
	`out2`	Trigger	`trig1`
			`trig2`
			`trig3`

4.4.5.2 Sharing a Repeated Capability across Class-Compliant Interfaces

Duplicate physical identifiers can occur in a driver that implements multiple class-compliant interfaces, each of which has a similar repeated capability. If the repeated capabilities refer to the same physical entities, the driver may represent them with a single repeated capability, thereby avoiding the possibility of duplicate physical identifiers.

For example, consider an IVI-COM driver that exports the IviScope and IviDigitizer class-compliant interfaces, each of which has a Channel repeated capability that refers to the same set of physical channels on the instrument. The driver developer may define one Channel repeated capability that the IviScope and IviDigitizer interfaces share.

4.4.5.3 Disambiguating Physical Identifiers

In cases where using the same physical identifier across multiple repeated capabilities seems natural but violates the rules specified in Section 4.4.5.1, *Uniqueness Rules for Physical Identifiers*, the following two approaches may be used:

- The driver defines different, uniquely-named physical identifiers.
- The driver defines *qualified physical identifiers*. A qualified physical identifier consists of a *physical name qualifier*, such as a repeated capability name or a qualified repeated capability name, two exclamation points (`!!`), followed by the physical identifier. Driver functions that take repeated capability selector parameters must accept the qualified physical identifiers. Driver functions may also accept unqualified identifiers if the driver can reliably determine which repeated capability the user intended. Notice that a qualified repeated capability name is required when the same repeated capability name is used in multiple classes with which the driver complies.

Consider an IVI-COM driver that exports the IviScope and IviSwtch class-compliant interfaces, each of which has a Channel repeated capability. Although the repeated capabilities refer to different physical entities on the instrument, the driver developer wants to use "`ch0`" and "`ch1`" within each repeated capability. To avoid violating the uniqueness rules specified in Section 4.4.5.1, *Uniqueness Rules for Physical Identifiers*, the driver developer may do either of the following:

- Use different, unqualified physical identifiers, such as "`scopeCh0`", "`scopeCh1`", "`swtchCh0`", and "`swtchCh1`".

- Use the following qualified physical identifiers:
    - `IviScopeChannel!!ch0`
    - `IviScopeChannel!!ch1`
    - `IviSwtchChannel!!ch0`
    - `IviSwtchChannel!!ch1`

  Because the driver is always able to distinguish between IviScope and IviSwtch functions and attributes, the driver accepts channel selector parameters of both forms ("`IviScopeChannel!!ch0`" and "`ch0`").

4.4.6 Expanding Virtual Identifiers

When specifying a virtual identifier in the IVI configuration store, the user can specify a mapping to more than just a simple physical identifier. For example, a user might map a virtual identifier to a set of physical identifiers or to a hierarchical selector containing only physical identifiers.

Typically, IVI configuration utilities do not validate the strings to which virtual names are mapped. After the IVI specific driver replaces the virtual identifiers in a repeated capability selector with the strings to which the identifiers are mapped, the driver validates the resulting expression.

When IVI drivers expand the mapping of a virtual identifier, the driver inserts brackets around mapped strings that contain at least one comma (`,`) but no colons (`:`). Inserting the brackets ensures that the proper order of precedence is maintained. For example, assume that the user creates the following virtual identifier mappings in the IVI configuration store:

```
MyWindow = Display2:Window1
```

```
MyTraces = Trace1,Trace3
```

If the user passes "`MyWindow:MyTraces`" as a selector, the driver expands the mappings to result in the following selector:

```
Display2:Window1:[Trace1,Trace3]
```

After the IVI specific driver expands the virtual identifiers in a selector, the result is called a *physical repeated capability selector*.

Notice that mapping virtual identifiers to hierarchical selectors is of no value when using IVI-COM collections to represent repeated capabilities.

4.4.7 Formal Syntax for Repeated Capability Selectors

The following describes the formal syntax for repeated capability selectors.

A syntactically valid repeated capability selector consists of zero or more *repeated capability path segments* separated by colons (`:`). White space around colons is ignored. When used with IVI-COM collections, repeated capability selectors have exactly one repeated capability path segment. In other words, colons (`:`) are not allowed in repeated capability selectors used with IVI-COM collections.

A repeated capability path segment consists of one or more *repeated capability list elements*, separated by commas (`,`). White space after commas is ignored. A repeated capability path segment may be enclosed in square brackets (`[]`).

A repeated capability list element consists of a *repeated capability token* or a repeated capability range.

A repeated capability range consists of two repeated capability tokens separated by a hyphen (`-`).

The order of precedence of operators is square brackets (`[]`), hyphen (`-`), colon (`:`), and comma (`,`). Each operator is evaluated from left to right.

A repeated capability token is a physical repeated capability identifer or a virtual repeated capability identifier.

A syntactically valid physical or virtual repeated capability identifier consists of one or more of the following characters: `a-z`, `A-Z`, `0-9`, !, and _.

5. Conformance Requirements

5.1 Introduction

The IVI Foundation defines standard APIs for IVI drivers. These APIs include IVI inherent capabilities and the base and extended capabilities for each IVI instrument class. The IVI Foundation also defines requirements for how drivers that implement these APIs behave.

The IVI Foundation allows for some flexibility in implementing the standard APIs. Two types of flexibility exist. Some elements of these APIs are optional. For example, an IVI class-compliant specific driver does not have to implement the extended class capability groups of its class. However, if it does implement an extension group, it complies with all requirements for that extension group. Another example is interchangeability checking. IVI class-compliant specific drivers are not required to implement this feature. If an IVI driver implements interchangeability checking, it implements all the interchangeability checking functions that *IVI-3.2: Inherent Capabilities Specification* defines. IVI.NET, IVI-COM and IVI-C class-compliant specific drivers differ in how they handle optional functions they do not implement. IVI-C class-compliant specific drivers do not export the functions. IVI-COM class-compliant specific drivers export the methods, but the methods return the Function Not Supported error. IVI.NET class-compliant specific drivers export the methods, but the methods throw a Function Not Supported exception.

Another type of flexibility lies in the extent to which an IVI driver implements a feature. For example, IVI specific drivers are required to implement the attribute for enabling and disabling state caching. However, each IVI specific driver has the choice of implementing state caching for all, some, or none of its attributes. Interchangeability checking is another example. IVI class-compliant specific drivers that implement the interchangeability-checking API comply with the interchangeability checking rules defined in the class specifications. An IVI driver that tracks the state of the instrument and thus recognizes when instrument settings become invalid, can provide more complete interchangeability checking than a driver that does not track the state of the instrument. However, tracking the state of the instrument is not required in implementing interchangeability checking.

Range checking is another example of flexibility that the IVI Foundation allows in the implementation of a feature. IVI specific drivers are required to validate all parameters to the extent that it is feasible. In many cases IVI specific drivers can completely validate parameters that represent instrument settings. In some cases, however, the valid range of an instrument setting might depend on the interrelationship of many state variables in the instrument. The algorithm that the instrument uses to determine the valid range for the parameter might be so complex that it is unreasonable to replicate in the driver. In this case, the driver should at least verify that the parameter falls within the maximum and minimum allowable values.

This section enumerates the required and optional features of IVI drivers. This section also identifies the features for which the IVI Foundation allows IVI drivers flexibility in implementation and describes the types of flexibility allowed. This section also contains requirements for how IVI drivers document their level of compliance with the specifications.

5.2 Conformance Verification Process And IVI Conformance Logo Usage

5.2.1 Purpose of Conformance Verification

Conformance verification is necessary for two reasons: first to verify that the driver complies with the requirements of the various applicable IVI specifications, and second, to provide the documentation necessary to allow the IVI Foundation to grant the IVI Conformant logo.

5.2.2 Verification Process

IVI Drivers shall be evaluated and tested to verify that they meet all applicable IVI requirements.

Appendix B – *Example: IVI Conformance* Tests contains a detailed list of all potential dimensions along which IVI drivers *should* be tested. Section 5.2.2.1 describes the minimum set of tests that IVI driver suppliers *shall* perform.

5.2.2.1 Requirements for Testing IVI Drivers

This section describes the minimum testing that IVI driver suppliers shall perform on an IVI driver before releasing it.

5.2.2.1.1 Unit Test Procedure

Every entry point in the driver shall be tested in simulation mode and connected to at least one of the driver's supported instruments. The complete unit test shall be run at least once on one *test setup* as defined below.

- Instrument Model and Firmware Revision
- Bus Interface
- Operating System and Service Pack
- OS Bitness and Application Bitness
- VISA Vendor and Version (if VISA is required by the driver)
- IVI Shared Components Version

Using the driver in simulation mode, the driver tester shall do the following:

- Call all implemented functions/methods at least once and verify that they return without failure.
- Use at least one legal value for each instrument setting and verify that the driver accepts the value(s).

Using the driver with an instrument, the driver tester shall run one or more client programs that call the driver and verify that the driver and instrument respond as expected, either by reading values back from the instrument or through external means. The client programs(s) shall test the driver in the following ways:

- The client program(s) shall test each function/method with the values listed below for each parameter that represents an instrument setting. The program(s) or tester shall verify that all output values are reasonable for each set of input parameters tested.
  - At least one legal value
  - If the driver performs range checking over one or more continuous ranges, then each of the following:
    - legal values at the limits of each range
    - illegal values at the limits of each range
    - one legal value within each range
  - If the driver coerces inputs to discrete numeric settings, then each of the following:
    - at least one value between each discrete setting
    - legal values at the limits of the entire range
    - at least one illegal value
  - For parameters with a discrete set of explicitly specified legal values (such as enumerations and Booleans), each legal value and at least one illegal value

- The client program(s) shall test each attribute as if it were a function/method, that is, as the single parameter to a "SetAttribute" call and a return value from a "GetAttribute" call.

- The client program(s) shall test at least one function/method and one attribute with each repeated capability instance specifier legal for the setup and at least one illegal repeated capability instance specifier.

5.2.2.2 Driver Installation Testing

The installation shall be tested by:

- Installing the driver

- Instantiating and running the driver using the configuration store
- Instantiating and running the driver by instantiating it directly by client code
- Testing every example included with the driver

Installation testing shall be done on at least two operating systems, unless the driver only supports a single operating system. Unless the driver supports only one bitness, at least one tested operating system shall be 32-bit, and at least one shall be 64-bit. Each operating system tested must have been updated with a service pack that is within 6 months of being the most recent, unless no service packs are available.

5.2.2.3 Driver Buildability Testing

For drivers that include source code, the driver tester shall re-build the driver from the installed source code according to the documented instructions. The driver tester shall verify that the re-built driver works on at least one combination of operating system, service pack state and bitness.

5.2.2.4 Documentation of Testing

The compliance documentation for the driver shall contain:

- A list of the test setups on which the complete set of unit tests were run
- A list of the operating system on which installation testing was done
- A list of the operating systems on which driver buildability testing was done
- A list of known issues that reflect all test failures

5.2.3 Driver Registration

Driver providers wishing to obtain and use the IVI Conformance logo shall submit to the IVI Foundation the driver compliance document described in Section 5.23, *Compliance Documentation*, along with driver information and a point of contact for the driver. The information shall be submitted to the IVI Foundation website: complete upload instructions are available on the site. Driver vendors who submit compliance documents will receive an email from the IVI Foundation containing IVI Conformant logo graphics.

The IVI Foundation may make some driver information available to the public for the purpose of promoting IVI drivers. All information is maintained in accordance with the IVI Privacy Policy, which is available on the IVI Foundation website.

5.2.4 Permissible Uses of The IVI Conformant Logo

The IVI Conformant Logo may be used for promotion and publicity of registered IVI Specific Drivers and IVI Class Drivers. The Logo shall not be used to promote Specific Driver Wrappers, Custom Class Drivers, or any other IVI-related or IVI-enabled products.

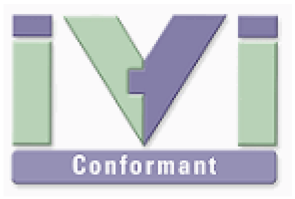


Figure 5-1. IVI Conformant Logo

5.3 API Types

IVI drivers shall export at least one of the following API types: .NET, COM or C. These API types are described in Sections 4.1, *IVI-COM Driver Architecture*, 4.2, *IVI-C Driver Architecture*, and 4.3, *IVI.NET Driver Architecture*. If the IVI driver exports a COM API, it shall comply with the requirements of Section 5.14.1, *Enumerations*

For all types of IVI drivers, enumeration values shall be explicitly specified in the source code for the enumeration. One of the members shall be assigned a value of zero.

IVI-COM Requirements. If the IVI driver exports a C API, it shall comply with the requirements of Section 5.16, *IVI-C Requirements*. If the IVI driver exports a .NET API, it shall comply with the requirements of Section 5.17, *IVI.NET Requirements*.

An IVI driver that exports multiple APIs from an IVI driver may implement one as native and one or more as a wrapper. *IVI-3.2: Inherent Capabilities Specification* defines a few extra functions that wrapper APIs shall implement.

5.3.1 IVI Class Driver API Types

An IVI class driver shall export a C API.

An IVI class driver may export a COM API.

An IVI class driver may export a CLS compliant .NET API.

An IVI class driver shall be able to load and call into IVI-C class-compliant specific driver of the same class.

An IVI class driver may load and call into IVI-COM class-compliant specific drivers of the same class.

An IVI class driver may load and call into IVI.NET class-compliant specific drivers of the same class.

5.4 Compliance with Other Specifications

All IVI drivers shall comply with *IVI-3.2: Inherent Capabilities Specification*. IVI drivers that claim conformance with an IVI instrument class shall comply with the specifications for that class. See Section 5.5, *Compliance with Class Specifications*, for more details. Note that some IVI class specifications reference *IVI-3.3: Standard Cross Class Capabilities Specification* for requirements that are shared among multiple classes. IVI custom specific drivers may follow *IVI-3.3: Standard Cross Class Capabilities Specification* when applicable to the instrument specific features. IVI specific drivers shall export instrument specific features in a manner that is compliant with *IVI-3.4: API Style Guide*.

5.5 Compliance with Class Specifications

This section describes the requirements an IVI specific driver shall follow to be compliant with an IVI class specification. In addition, it provides requirements for IVI class-compliant specific drivers to comply with a capability group.

5.5.1 Minimum Class Compliance

An IVI class-compliant specific driver shall implement the base class capabilities and zero or more of the class extension capability groups for the IVI class with which the driver claims compliance. The IVI class-compliant specific driver shall comply with the requirements of the base capability group. If an IVI driver implements a class extension capability group, it shall comply with all the requirements of the extension capability group.

Each IVI class specification contains requirements for minimum compliance that may go above and beyond the requirements specified in this section.

5.5.2 Requirements for IVI-C, IVI-COM, and IVI.NET APIs

An IVI-COM or IVI.NET class-compliant specific driver shall export all APIs for the IVI class with which the driver claims compliance. If the IVI-COM or IVI.NET driver does not implement one or more of the API elements that the class specification defines, the driver returns the appropriate Not Supported error from the unsupported APIs, methods, and properties. *IVI-3.2: Inherent Capabilities Specification* defines the Not Supported error.

An IVI-C class-compliant specific driver shall export all functions and attributes that the capability groups that it implements require. All IVI-C specific drivers shall precede all functions and attribute names with the specific driver prefix. Refer to Section 5.16.4, *Prefixes*, for more information.

5.5.3 Capability Group Compliance

For an IVI specific driver to be compliant with a capability group defined within the IVI class specification for which the driver claims compliance, the IVI specific driver shall comply with the following rules:

- The IVI specific driver shall implement all attributes that the capability group defines. Refer to Section 5.6.1, *Attribute Compliance Rules*, for details.
- The IVI specific driver shall implement all functions that the capability group defines. Refer to Section 5.6.2, *Function Compliance Rules*, for details.
- The IVI specific driver shall implement the behavior model that the capability group defines.
- The IVI specific driver shall prevent the presence of an extended capability from affecting the behavior of the instrument unless the application program explicitly uses the extension capability group. Refer to Sections 3.3.4 and 5.10.1.4 for more detailed requirements on Disabling Unused Extensions.
- The IVI specific driver shall comply with any additional compliance rules or exceptions that the class specification defines for the capability group.

Note: If the class specification defines additional compliance rules or exceptions to the above rules for a capability group, the additional rules and exceptions appear in compliance notes section for the capability group in the class specification.

5.5.4 Coercion

For attributes that allow for a continuous range of values, the IVI driver shall coerce user-specified values if the instrument implements only a discrete set of values.

In general, an IVI class-compliant specific driver shall coerce user-specified values in accordance with the IVI class specification with which it complies.

Typically, for each real-valued attribute, IVI class specifications define a coercion direction in which an IVI specific driver coerces a user-specified value. Possible coercion directions are “Up”, “Down”, and “None”.

- Up – indicates that an IVI specific driver may coerce a user-specified value to the nearest value that the instrument supports that is greater than or equal to the user-specified value.
- Down – indicates that an IVI specific driver may coerce a user-specified value to the nearest value that the instrument supports that is less than or equal to the user-specified value.
- None – indicates that the IVI specific driver is shall not coerce a user-specified value. If the instrument cannot be set to the user-specified value, the IVI specific driver shall return an error.

In certain cases, an IVI class-compliant specific driver may coerce a user-specified value a manner different from the IVI class specification. The driver may do this when the instrument can satisfy the user’s request more appropriately with a value that is different from the value that the class specification coercion rules suggest. For example, if a user specifies a range of 10.01 volts for a DMM measurement and the IviDmm class specification specifies that the value be coerced up, the instrument might coerce this value down to 10.0 because the instrument can measure up to 11.0 volts when in the 10.0 volt range.

The driver may rely on the instrument to coerce values if the instrument coerces user-specified values in a manner consistent with the IVI class specification.

5.6 Attribute and Function Compliance Rules

This section describes the compliance requirements for all user-accessible attributes and user-callable functions defined in *IVI-3.2: Inherent Capabilities Specification* and the IVI class specifications.

5.6.1 Attribute Compliance Rules

To comply with a particular attribute that a specification defines, an IVI specific driver shall comply with the following rules:

- The IVI specific driver shall implement the behavior that the specification defines for the attribute.
- If the attribute has defined values, the IVI specific driver shall implement at least one of the defined values.
- If an attribute has defined values and the IVI specific driver adds instrument specific values for the attribute, the IVI specific driver shall define the instrument specific values to be equal to or greater than the base extension value that the attribute defines for instrument specific values.
- If the attribute does not have defined values, the IVI specific driver is required to support only the values that the instrument supports.
- The IVI specific driver shall comply with any additional compliance rules or exceptions that the specification defines for the attribute.

Note: If the specification defines additional compliance rules or exceptions to the above rules for an attribute, the additional rules and exceptions appear in compliance notes section for the attribute in the specification.

To comply with a particular attribute that a specification defines, an IVI class driver shall comply with the following rules:

- If the specification also allows IVI specific drivers to implement the attribute, the IVI class driver attribute acts as a passthrough to the specific driver attribute.
- If the specification does not allow IVI specific drivers to implement the attribute, the IVI class driver implements the behavior that the specification defines for the attribute.
- If an attribute has defined values and an IVI custom class driver adds additional values for the attribute, the IVI custom class driver shall define the additional values to be equal to or greater than the base extension value that the attribute defines for this purpose.

5.6.1.1 Complementary Attributes and Configuration Functions

IVI driver configuration function parameters that set an instrument state variable shall have a corresponding read/write attribute. Exception: For state variables closely coupled in the instrument, the attributes may be read-only.

The following naming conventions shall be observed for attributes that correspond to instrument state variables:

- For IVI-COM and IVI.NET, the name of the property shall correspond to the name of the parameter. For example, if a configure function includes a parameter named “foo”, the corresponding property might be named `Foo` or `TriggerFoo`.

- For IVI-C, the corresponding attribute shall correspond to the name of the parameter. For example, if a configure trigger function includes a parameter named “foo”, the corresponding attribute name might be `<CLASS_NAME>_ATTR_TRIGGER_FOO`.

5.6.2 Function Compliance Rules

To comply with a particular function that a specification defines, an IVI specific driver shall comply with the following rules:

- The IVI specific driver shall implement the behavior that the specification defines for the function.
- If the IVI specific driver returns status codes other than the status codes that the specification defines for the function, the actual values of the instrument specific status codes shall be within the instrument specific function status code range as specified in Table 5-3. Status Code Types and Ranges. Note: This requirement is not applicable to IVI.NET.
- If the specification specifies that the IVI specific driver use the value of an input parameter to set a particular attribute, the IVI specific driver shall implement the parameter in accordance with the same compliance requirements that the specification defines for the attribute. Notice that class specifications do not restrict IVI specific drivers from defining instrument specific values for attributes.
- If the specification specifies that the IVI specific driver return the value of a particular attribute in an output parameter, the IVI specific driver shall implement the parameter in accordance with the same compliance requirements that the specification defines for the attribute.
- If the specification defines values for a function parameter, the IVI specific driver shall support at least one of the defined values.
- If the specification defines values for a function parameter and the IVI specific driver defines instrument specific values for the parameter, the IVI specific driver shall define the instrument specific values to be equal to or greater than the base extension value that the parameter defines.
- If the specification does not define values for a function parameter, the IVI specific driver is required to implement only the values that the instrument supports.
- The IVI specific driver shall comply with any additional compliance rules or exceptions that the specification defines for the function.

Note: If the specification defines additional compliance rules or exceptions to the above rules for a function, the additional rules and exceptions appear in compliance notes section for the function in the specification.

To comply with a particular function that a specification defines, an IVI class driver shall comply with the following rules:

- If the specification also allows IVI specific drivers to implement the function, the IVI class driver function acts as a passthrough to the specific driver function.
- If the specification does not allow IVI specific drivers to implement the function, the IVI class driver implements the behavior that the specification defines for the function.

5.7 Use of Shared Components

All IVI drivers except IVI class drivers shall use the IVI Configuration Server shared component to retrieve user-configured values for inherent attributes, configurable initial settings, and user mappings for channel names and other repeated capabilities.

All IVI-C or IVI-COM drivers that generate infinity or NaN values or that perform comparisons on such values shall use the Floating Point shared component. *IVI-3.12: Floating Point Services Specification* specifies the API for generating and recognizing infinity and NaN values. For IVI.NET, PositiveInfinity, NegativeInfinity, and NaN are part of the floating point types.

All IVI-C drivers shall use the Session Management and Error Message components. IVI class drivers shall use the Dynamic Driver Loading component to load IVI-C specific drivers. The Session Management, Error Message, and Dynamic Driver Loading components are defined in *IVI-3.9: C Shared Components Specification*.

IVI class drivers that load IVI-COM specific drivers shall use the IVI-COM Session Factory. The IVI-COM Session Factory is defined in *IVI-3.6: COM Session Factory Specification*. IVI class drivers that load IVI.NET specific drivers shall use one of the IVI.NET session factory methods. The IVI.NET session factory methods are defined in *IVI-3.2: Inherent Capability Specification*, and in the individual instrument class specifications.

5.7.1 Use of the IVI Configuration Server

IVI-3.5: Configuration Server Specification specifies COM and C APIs for the IVI configuration store. IVI drivers shall not access the IVI configuration store except through the IVI Configuration Server. IVI.NET drivers may access the IVI configuration store using the standard IVI Configuration Server PIAs.

Multiple IVI configuration store files can exist on a system. Refer to Section 3.2.3, *Instantiating the Right Configuration Store From Software Modules*, in *IVI-3.5: Configuration Server Specification* for details on how to an IVI driver correctly instantiates the configuration store.

IVI drivers shall not write to the IVI configuration store. IVI drivers shall not read the IVI configuration store after the Initialize function returns.

For more information on the inherent settings that are configurable through the IVI configuration store, refer to Section 3.8, *Configuration of Inherent Features*. For more information on configurable initial settings that are configurable through the IVI configuration store, refer to Section 3.3.5, *Applying Configurable Initial Settings from the IVI Configuration Store*.

5.8 Use of I/O Libraries for Standard Interface Buses

If an IVI specific driver communicates with a device using a GPIB or VXIbus interface, it shall use the VISA-C API, VISA-COM API, or VISA-COM PIAs for I/O communication, as defined in VXI*plug&play* specifications *VPP-4.3.2: VISA Implementation Specification for Textual Languages* and *VPP-4.3.4: VISA Implementation Specification for COM*. The driver may also use another I/O library in addition to the VISA I/O library, as long as the driver works when VISA is present and the additional I/O library is not present.

If an IVI specific driver communicates with a device using a bus other than GPIB or VXIbus, VISA may be used as the I/O library.

5.9 Repeated Capability Identifiers and Selectors

This section specifies the requirements for defining physical repeated capability identifiers and parsing repeated capability selectors.

5.9.1 Defining Physical Repeated Capability Identifiers

An IVI specific driver that contains repeated capabilities shall define exactly one unqualified physical identifier for each statically-known repeated capability instance. This applies even if the driver defines only one repeated capability instance.

An IVI specific driver shall comply with the uniqueness rules specified in Section 4.4.5, *Ambiguity of Physical Identifiers*.

5.9.2 Applying Virtual Identifier Mappings

During initialization, an IVI specific driver shall retrieve the virtual repeated capability identifiers and their mappings from the IVI configuration store.

When a user passes a repeated capability selector to an IVI driver function or IVI-COM or IVI.NET collection item, the IVI specific driver shall replace each instance of a virtual repeated capability identifier in the selector with the corresponding mapped string. An IVI specific driver shall not replace an instance of a virtual identifier that appears as a substring of another virtual identifier. For example, if the session configuration in the IVI configuration store defines a mapping for `Chan3`, the driver does *not* apply that mapping in the selector `"MyChan3"`.

The driver shall replace a virtual identifier with its mapped string even if the virtual identifier is also a valid physical identifier for an instance of the repeated capability.

Note: It is important for the user to avoid the case where a virtual identifier is mapped to a physical identifier that is identical to a different virtual identifier in the same repeated capability.

Refer to Appendix A – *Example: Applying Virtual Identifier* Mappings, for an example of how an IVI specific driver applies virtual identifier mappings in a repeated capability selector.

5.9.3 Validating Repeated Capability Selectors

After applying the virtual identifier mappings to a repeated capability selector, an IVI specific driver shall validate the fully resolved selector.

IVI specific drivers may perform partial validation on repeated capability selectors before or during the application of virtual identifier mappings. However, because the mapped string for a virtual identifier may contain operators, the IVI specific driver is not able to fully validate the selector until it applies all the mappings. Refer to Section 4.4.7, *Formal Syntax for Repeated Capability Selectors*, for a list of the valid selector operators.

If the IVI driver finds an invalid condition in the repeated capability selector, the driver shall return one of the following error codes: Badly-Formed Selector, Invalid Number of Levels in Selector, Invalid Range in Selector, Unknown Name in Selector, Unknown Physical Identifier, or Unknown Channel Name.

5.9.4 Accepting Empty Strings for Repeated Capability Identifiers

An IVI specific driver that defines exactly one instance of a non-nested repeated capability shall accept empty string as a valid physical repeated capability selector. For example, consider an IVI specific driver that complies with the IviFgen class specification and that interfaces to an instrument that has only one channel. The IVI specific driver defines a physical identifier for the channel. The IVI specific driver accepts the physical identifier and empty string as valid physical selectors for the channel. In the IVI-C architecture `VI_NULL` can be used in place of empty string.

An IVI specific driver shall not allow empty string in any other case. In particular, if an IVI driver defines exactly one instance of a repeated capability that is part of a hierarchy, the IVI specific driver shall require that the physical repeated capability selector contain the physical identifier for that repeated capability.

5.9.5 Indexing Repeated Capabilities

Whenever a repeated capability instance has a corresponding index, the IVI-C and IVI-COM indexes shall be one-based and IVI.NET indexes shall be zero-based. This includes

- Properties and methods that return a repeated capability instance name given a particular index
- Collection indexes (IVI-COM and IVI.NET)

5.10 IVI Features

This section describes the feature requirements for IVI drivers. These requirements pertain to the behavior of the drivers. Some behaviors are required of all IVI drivers, while others are required only of class-compliant drivers.

5.10.1 Interchangeability

Interchangeability is a feature of IVI class-compliant specific drivers and IVI class drivers, but not IVI custom specific drivers.

5.10.1.1 Consistency of Instrument Specific APIs with Class API

When an IVI class-compliant specific driver implements instrument specific capabilities, the driver should export those capabilities in a way that is consistent with the class-defined capabilities.

5.10.1.2 Accessing Specific APIs without Reinitializing

When a user initializes an IVI driver using a class-defined API, the driver shall allow the user to access instrument specific features without performing another initialization step.

5.10.1.3 Use of Virtual Identifiers for Repeated Capabilities

Refer to Section 5.9, *Repeated Capability Identifiers and Selectors*.

5.10.1.4 Disabling Unused Extensions

An IVI class-compliant specific driver shall disable all extension capability groups in the Initialize and Reset With Default functions. An IVI class-compliant specific driver shall ensure that instrument settings that correspond to a particular extension capability group do not affect the behavior of the instrument until one of the following conditions occurs:

- The application program calls a function that belongs to the extension capability group.
- The application program sets an attribute that belongs to the extension capability group.
- The application program sets an attribute in another capability group to a value that requires the presence of the extension capability group. This applies regardless of whether the application sets the attribute directly or through a high-level function call.

When implementing this feature, an IVI class-compliant specific driver may optimize the implementation as it sees fit.

An IVI class-compliant specific driver shall also disable instrument specific features in the Initialize and Reset with Default functions if the features affect the behavior of the class-defined capabilities.

5.10.1.5 Applying Configurable Initial Settings from the IVI Configuration Store

At initialization, an IVI specific driver shall retrieve the configurable initial settings from the IVI driver session configuration in the IVI configuration store. For each attribute and associated value in the configurable initial settings, the IVI specific driver shall set the attribute to the value in the Initialize and Reset With Defaults functions. Refer to Section 5.3.3, *Defining Configurable Initial Settings in the IVI Configuration Store*, in *IVI-3.17: Installation Requirements Specification*, for details on how the installation program for an IVI specific driver sets up configurable initial settings information in the IVI configuration store.

The mechanism for applying configurable initial settings may be used to allow users to configure the value of class-defined attributes, instrument specific attributes, or vendor specific attributes.

5.10.1.6 Interchangeability Checking

Interchangeability checking is *enabled* if the Interchange Check attribute is set to `VI_TRUE`. Interchangeability checking is *disabled* if the Interchange Check attribute is set to `VI_FALSE`.

If an IVI driver does not implement interchangeability checking, the driver shall return an error if the user attempts to enable interchangeability checking.

If interchangeability checking is enabled, an IVI class-compliant specific driver shall check for attributes that are not in a user-specified state. The driver shall implement this type of checking at either the full or minimal level as specified in Section 3.3.6, *Interchangeability Checking*.

IVI class-compliant specific drivers and IVI class drivers may implement other types of interchangeability checking.

5.10.1.7 Coercion Recording

Coercion recording is *enabled* if the Record Coercions attribute is set to `VI_TRUE`. Coercion recording is *disabled* if the Record Coercions attribute is set to `VI_FALSE`.

If an IVI specific driver does not implement coercion recording, the driver shall return an error if the user attempts to enable coercion recording.

If an IVI specific driver implements coercion recording, the driver shall create a record for each user-specified value that it coerces. The driver may impose a maximum on the size of the queue that holds the coercion records. If the queue overflows, the driver shall discard the oldest coercion record.

5.10.2 Interchangeability Features in Custom Drivers

IVI custom specific drivers may implement the following interchangeability features:

- applying configurable initial settings from the IVI configuration store
- coercion recording

All IVI custom specific drivers that apply configurable initial settings shall do so in the same manner as described in Section 5.10.1.5, *Applying Configurable Initial Settings from the IVI Configuration Store*.

All IVI custom specific drivers that implement coercion recording shall do so in the same manner as described in Section 5.10.1.7, *Coercion Recording*.

5.10.3 Range Checking

Range checking is *enabled* when the Range Check attribute is set to `VI_TRUE`. Range checking is *disabled* if the Range Check attribute is set to `VI_FALSE`.

If range checking is enabled, the IVI specific driver shall validate all parameters to the extent that it is feasible. IVI specific drivers shall perform range checking regardless of whether simulation is enabled.

The valid range of an instrument setting might depend on the interrelationship of many state variables in the instrument. The algorithm that the instrument uses to determine the valid range for the parameter might be so complex that it is unreasonable to replicate in the driver. In this case, the driver should at least check that the parameter falls within the absolute maximum and minimum allowable values.

All IVI specific drivers shall fully validate parameters in cases where the instrument does not handle errors in a reasonable manner.

When an IVI specific driver performs range checking on a `ViReal64` parameter or attribute that has a discrete set of legal values, the driver shall not include a guard band around any of the legal values.

5.10.4 Instrument Status Checking

Instrument status checking is *enabled* if the Query Instrument Status attribute is set to `VI_TRUE`. Instrument status checking is *disabled* if the Query Instrument Status attribute is set to `VI_FALSE`.

If the IVI specific driver can determine the status of the instrument through the instrument I/O interface, the driver shall implement code that determines the instrument status and return the Instrument Status error code if the instrument status indicates that an error has occurred. When instrument status checking is enabled, the driver shall invoke the status checking code in the following types of functions:

- All user-callable class-defined functions that perform instrument I/O, unless the class specification specifies otherwise.
- All user-callable instrument specific functions that perform instrument I/O, except for functions whose operations are included as part of a higher level function. Functions whose operations are included as part of a higher level function may invoke the status checking code.

The driver shall document which user-callable functions that perform I/O do not implement status checking.

If a driver can determine the instrument status without performing a separate query and response or other action that has a significant performance impact, the driver shall invoke the status checking code without regard to the setting of the Query Instrument Status attribute. If determining the instrument status has a significant performance impact, the driver shall not invoke the status checking code when instrument status checking is disabled. For the purpose of this rule, querying the instrument across a message-based interface has a significant performance impact.

If the instrument does not support the ability to query the instrument status, the setting of the Query Instrument Status attribute shall have no effect on the behavior of the instrument driver.

The setting of the Query Instrument Status attribute shall have no effect on the operation of the Error Query function.

Note: The driver shall never invoke the status checking code when simulation is enabled.

5.10.5 Simulation

Simulation is *enabled* if the Simulate attribute is set to `VI_TRUE`. Simulation is *disabled* if the Simulate attribute is set to `VI_FALSE`.

An IVI specific driver shall provide sufficient functionality when simulation is enabled such that it is usable in an application when the instrument is not present. When simulation is enabled, the IVI specific driver shall do the following:

- Refrain from performing I/O.
- Perform range checking when the Range Check attribute is set to `VI_TRUE`. The range checking that the IVI specific driver performs need not be as complete as when simulation is disabled.
- Perform the same parameter coercion that the driver performs when simulation is disabled.
- Return simulated data for output parameters. An IVI specific driver may provide configurable settings for the output data values and status return values for each function.

The Close function in the driver shall close the I/O session regardless of whether simulation is enabled or disabled.

If the specific driver is initialized with simulation disabled, the specific driver may return the Cannot Change Simulation State error if the user attempts to enable simulation prior to calling the Close function.

If the specific driver is initialized with simulation enabled, the specific driver shall return the Cannot Change Simulation State error if the user attempts to disable simulation prior to calling the Close function.

5.10.6 State Caching

State caching is *enabled* if the Cache attribute is set to `VI_TRUE`. State caching is *disabled* if the Cache attribute is set to `VI_FALSE`.

If state caching is disabled, the IVI specific driver shall perform I/O whenever a user program sets a hardware configuration attribute.

If state caching is enabled and the user sets a hardware configuration attribute, the IVI specific driver shall avoid performing I/O when all the following conditions are true:

- The driver caches the state of the attribute.
- The cache value for the attribute is valid.
- The cache value for the attribute is equal to the value that the user requests.

5.10.7 Multithread Safety

An IVI driver shall be multithread safe.

For IVI-C drivers, each user callable function, except Initialize, Close, Lock, and Unlock, shall acquire a multithread lock on a session and not release it until the function returns. For the Initialize function, IVI-C drivers do not acquire or release a multithread lock. For the Close function, IVI-C drivers may release the multithread lock before the return. Except for the Lock function, user callable functions in IVI-C drivers shall never return while still holding onto a lock. IVI-C drivers shall acquire and release multithread locks using the Session Management API in the C shared components. Class Driver Specifications may provide additional rules and exceptions for acquiring and releasing of a multithread lock.

For IVI.NET drivers, each user callable method except the two overloads of the Lock method, shall acquire a multithread lock and not release it until the method returns. The lock acquired within each method call shall be consistent with the mode of locking (per-instance, AppDomain-wide, or machine-wide) established in the call to the IVI.NET driver constructor. The mode of locking shall not change over the lifetime of an IVI.NET driver session. See Section 8 in *IVI-3.2: Inherent Capabilities Specification* for details on how the mode of locking is determined. See Section 4.3.11, *Multithread Locking* for an explanation of the three modes of IVI.NET driver locking. The lock shall also be the same lock used to implement the Lock functions exposed via the IIviDriverUtility interface. It is recommended that IVI.NET drivers use the LockManager class in the Ivi.Driver.dll assembly to implement both locking within each method call as well as locking via the IIviDriverUtility.Lock method.

5.10.8 Resource Locking

The IVI Foundation has not yet defined the requirements for Resource Locking.

5.10.9 Extensible Access to Instrument Features

All IVI specific drivers that use message-based I/O shall include functions that allow a user to send user-specified strings to the instrument and return results. Such functions shall not obviate the need for well-designed functions and attributes for robust access to instrument specific features.

IVI drivers that interface with non-message-based instruments may export functions that allow the user to directly communicate with the instrument.

5.11 Configuration of Inherent Features

All IVI specific drivers shall accept logical names as well as I/O resource descriptors for the Resource Name parameter in the Initialize function.

If the user passes a logical name to the Resource Name parameter of the Initialize function, the driver shall use the following precedence to assign values for each inherent attribute:

1. Value specified in the `OptionsString` parameter.
2. Value specified in the IVI configuration store.
3. Default value as defined in *IVI-3.2: Inherent Capabilities Specification.*

If the user passes an I/O resource descriptor to the Resource Name parameter of the Initialize function, the driver shall use the following precedence to assign values for each inherent attribute:

1. Value specified in the `OptionsString` parameter.
2. Default value as defined in *IVI-3.2: Inherent Capabilities Specification.*

5.12 IVI Error Handling

5.12.1 IVI-C and IVI-COM Error Handling

Each IVI-C driver function, IVI-COM driver method, and IVI-COM driver property shall return status information in the form of a 32-bit integer value that complies with the following rules:

- If no error or warning conditions occur, the value shall be zero.
- If an error occurs, the value shall be less than zero.
- If a warning occurs and no errors occur, value shall be greater than zero.

Each IVI driver function shall return status information in the form of a 32-bit integer. A status code of 0 for all IVI drivers shall mean successful completion.

For both the IVI-COM and IVI-C drivers, bit 31 is the sign bit. Status values greater than zero are reserved for completion codes. Status values less than zero are reserved for errors.

IVI-COM drivers shall adhere to the bit pattern format described in Table **5-1**. All IVI-C drivers shall adhere to the bit pattern format described in Table **5-2**. IVI-C Status Codes. Table **5-3**. Status Code Types and Ranges lists the reserved status code ranges returned by IVI driver components.

Table 5-1. IVI-COM Status Codes

	⇐ 15 bits ⇒	⇐ 4 bits ⇒	⇐ 12 bits ⇒
31	**30** … **16**	**15** … **12**	**11** … **0**

Bit 31: Severity

0 = success or warning

1 = error

Bits 30-16: facility code

0004 = FACILITY_ITF

Bits 15-12: Type of error (see Table **5-3**. Status Code Types and Ranges)

Bits 11-0: Identify a particular error within the specified type

Table 5-2. IVI-C Status Codes

		⇐ 14 ⇒	⇐ 4 bits ⇒	⇐ 12 ⇒
31	**30**	**29** … **16**	**15** … **12**	**11** … **0**

Bit 31: Success or failure

0 = success or warning

1 = error

Bit 30: Reserved (always 0)

Bits 29-16: Driver type or IO definition

3FFA = IVI drivers and components

3FFF = VISA

Bits 15-12: Type of error (see Table **5-3**. Status Code Types and Ranges)

Bits 11-0: Identify a particular error within the specified type

Each status code that an IVI driver returns shall have a unique integer value. To prevent status code value conflicts within the API of an IVI driver, the IVI Foundation has established status code ranges for different types of status codes. Bits 15-12 are used to identify the status code type. Table **5-3**. Status Codes Types and Ranges lists the status code types and the bit patterns that identify them. All IVI drivers shall comply with the ranges in Table **5-3**. Status Codes Types and Ranges.

Table 5-3. Status Code Types and Ranges

IVI-COM bits 15-12	**IVI-C bits 15-12**	**Type**
0x0 (0000)	0x0 (0000)	VISA errors and warnings, defined in VPP-4.x - VXI*plug&play* VISA Specifications.
0x7 (0111)	0x0 (0000)	Common errors and warnings defined in Section 11, *Common IVI-C and IVI-COM Error and Completion Codes*, in *IVI-3.2: Inherent Capabilities Specification.*
0x1 (0001)	0x1 (0001)	Errors and warnings defined by IVI shared components, and errors and warnings defined in *IVI-3.3: Standard Cross Class Capabilitie Specification.*
0x2 (0010)	0x2 (0010)	Errors and warnings defined in the individual IVI class specifications.
0x3 (0011)	0x3 (0011)	*reserved*
0x4 (0100)	0x4 (0100)	Errors and warnings defined by individual IVI specific drivers.
0x5 (0101)	0x5 (0101)	Errors and warnings defined by individual IVI-MSS role components.
0x6 (0110) 0x8 - 0xB (1000)-(1011)	0x6 (0110) 0x8 - 0xB (1000)-(1011)	Vendor specific errors and warnings.

0xC - 0xF (1100)-(1111)	0xC - 0xF (1100)-(1111)	*reserved*

The IVI Error Coordinator is responsible for subdividing the status code range for IVI shared components and standard cross-class capabilities. This ensures that the status codes for the IVI shared components and standard cross-class capabilities do not conflict with each other. Refer to the individual specifications for actual status code values. Contact the IVI Error Coordinator to obtain a list of the sub-ranges or to allocate a new sub-range.

Status code values defined in IVI class specifications are not unique from one IVI class to another.

Instrument specific status code values are not unique from one IVI specific driver to another.

Vendor specific status code values are not unique from one vendor to another.

5.12.1.1 Example Values

Table **5-4**. Error Code Value Examples lists the example values for the various status code types. Each example value represents an error and uses 0x001 for bits 11-0.

Table 5-4. Error Code Value Examples

IVI-COM	IVI-C	Component
`0x80040001`	`0xBFFF0001`	VISA error code.
`0x80047001`	`0xBFFA0001`	Common or inherent error code.
`0x80041001`	`0xBFFA1001`	Shared components or standard cross class capability error code.
`0x80042001`	`0xBFFA2001`	Class specification error code.
`0x80044001`	`0xBFFA4001`	IVI specific driver error code.
`0x80045001`	`0xBFFA5001`	IVI-MSS role component error code.
`0x80046001` `0x80048001 -` `0x8004B001`	`0xBFFA6001` `0x80048001 -` `0x8004B001`	Vendor specific error code.

5.12.1.2 Base Values

Table **5-5**. Error and Completion Code Base Values lists the base values for the status code ranges that IVI drivers may use.

Table 5-5. Error and Completion Code Base Values

Name	C Identifier	Actual C Value	Actual COM Value
Inherent Error Base	`IVI_INHERENT_ERROR_BASE`	`0xBFFA0000`	`0x80047000`
Inherent Warning Base	`IVI_INHERENT_WARN_BASE`	`0x3FFA0000`	`0x00047000`
LxiSync Error Base	`IVI_LXISYNC_ERROR_BASE`	`0xBFFA3000`	`0x80043000`
LxiSync Warning Base	`IVI_LXISYNC_WARN_BASE`	`0x3FFA3000`	`0x00043000`
Specific Error Base	`IVI_SPECIFIC_ERROR_BASE`	`0xBFFA4000`	`0x80044000`
Specific Warning Base	`IVI_SPECIFIC_WARN_BASE`	`0x3FFA4000`	`0x00044000`

Class Error Base	`IVI_CLASS_ERROR_BASE`	`0xBFFA2000`	`0x80042000`
Class Warning Base	`IVI_CLASS_WARN_BASE`	`0x3FFA2000`	`0x00042000`
Shared Component Error Base	`IVI_SHARED_COMPONENT_ERROR_BASE`	`0xBFFA1000`	`0x80041000`
Shared Component Warning Base	`IVI_SHARED_COMPONENT_WARN_BASE`	`0x3FFA1000`	`0x00041000`
Vendor Specific Error Base	`IVI_VENDOR_SPECIFIC_ERROR_BASE`	`0xBFFA6000` `0xBFFA8000 – 0xBFFAB000`	`0x80046000` `0x80048000 – 0x8004B000`
Vendor Specific Warning Base	`IVI_VENDOR_SPECIFIC_WARN_BASE`	`0x3FFA6000` `0x3FFA8000 – 0x3FFAB000`	`0x00046000` `0x00048000 – 0x0004B000`

- Inherent Error Base and Inherent Warning Base are the base values for common status codes that the IVI Foundation defines. Refer to Section 11, *Common IVI-C and IVI-COM Error and Completion Codes*, in *IVI-3.2: Inherent Capabilities Specification* for the list of these common status code values.
- LxiSync Error Base and LxiSync Warning Base are the base values for common status codes that the IVI Foundation defines as part of the LxiSync API. Refer to *IVI-3.15 IviLxiSync Specification* for more details.
- Specific Error Base and Specific Warning Base are the base values from which an IVI specific driver defines the status code values of the instrument specific errors and warnings.
- Class Error Base and Class Warning Base are the base values from which the IVI class specifications define status code values.
- Shared Component Error Base and Shared Component Warning Base are the base values for status codes used by shared components and standard cross class capabilities that the IVI Foundation defines. Refer to the error and completion codes section in *IVI-3.3: Standard Cross Class Capabilities Specification* and in the specifications for the shared components for the list of these status code values.
- Vendor Specific Error Base and Vendor Specific Warning Base are the base values from which a software supplier defines vendor specific status codes.

5.12.2 IVI.NET Error Handling

In general, standard .NET exception classes shall be used to return exceptions from IVI inherent, class compliant, or instrument specific APIs.

IVI.NET drivers may throw exceptions that are derived from inherent or class-compliant exceptions from inherent, class compliant or instrument specific interfaces.

If an existing .NET class is not appropriate, IVI APIs may define a new exception class. In such cases, both an exception class and an error string shall be created for each distinct error. The new exception class shall derive directly or indirectly from System.Exception, and it shall not derive from System.ApplicationException or System.SystemException.

The exception class name shall be:
```
<Name>Exception_
```
where `<Name>` is a series of words, in Pascal casing, that describe the exception.

To represent warnings, the instrument class shall define a warning class with static GUID members for each type of warning that can be issued by the driver. The warning class shall be named as follows:
```
<ClassName>Warnings_
```

For general information about specific IVI.NET exceptions, refer to IVI 3.2, Inherent Capabilities, which defines a variety of useful exceptions for driver writers.

5.12.2.1 Remapping .NET Exceptions

Underlying software called by IVI.NET drivers might report errors (for example, C return codes) or throw exceptions. If the underlying software throws exceptions from which the driver cannot recover, drivers may allow the exceptions to propagate up the stack or may assign the caught exception to the InnerException property of a new exception and throw the new exception. If the underlying software reports errors without using exceptions, the driver shall throw a suitable exception. This section describes some specific cases.

5.12.2.1.1 .NET Runtime and Framework Exceptions

If an IVI.NET driver catches an exception from the .NET runtime or a .NET framework class, the driver may re-throw the exception as is.

5.12.2.1.2 I/O Timeout Exceptions

The Ivi.Driver.IOTimeoutException shall be thrown by the driver in cases where the driver is reporting an I/O timeout that the calling program can be reasonably expected to handle. If the underlying I/O software reported the timeout as an exception, the driver should assign the caught exception to the InnerException property.

5.12.2.1.3 Configuration Server Exceptions

If an IVI.NET driver detects an error from the Configuration Server that it cannot handle, it shall throw an Ivi.Driver.ConfigurationServerException.

If an IVI.NET driver catches an exception other than an Ivi.Driver.ConfigurationServerException from the Configuration Server that it cannot handle, it shall throw an Ivi.Driver.ConfigurationServerException, and it shall assign the caught exception to the InnerException property.

5.12.2.2 .NET Warnings

Warnings from IVI.NET components shall not be returned as return values or exceptions. Instead, each driver shall implement an event called Warning defined in IIviDriverOperation. If the calling program wishes to receive warnings, they will need to register this event handler.

Each warning shall be defined as a static property of type System.GUID and shall be a member of a class that contains only warnings.

5.13 Comparing Real Values

IVI drivers shall use fuzzy comparisons with approximately 14 decimal digits of precision when comparing floating point parameters or attribute values with any of the following types of values:

- Discrete legal values
- Minimum and maximum limits of continuous ranges
- Discrete values which the driver coerces to a continuous range of values

5.14 Allowed Data Types

IVI-C user-callable functions shall use the data types listed in Table 5-6. Compatible Data Types for IVI Drivers, with the following restrictions:

- IVI-C drivers shall not use signed 16-bit integers, except for the `TestResult` output parameter of the Self Test function. This exception provides compatibility with the VXI*plug&play* specifications.
- IVI-C drivers shall use arrays of 32-bit integer values to represent integer arrays, except when typical usage involves arrays of at least one million elements or a class specification requires the use of arrays of

16-bit or 8-bit integer values.

- IVI-C drivers shall use arrays of 64-bit floating point values to represent floating point arrays, except when typical usage involves arrays of at least one million elements or a class specification requires the use of arrays of 32-bit floating point values.
- To represent a scalar output parameter, IVI-C drivers shall use data types of the form `Vi<Type>*`, such as `ViInt32*` or `ViReal64*`. The form `Vi<Type>*` is equivalent to the form `ViP<Type>` used in Section 3.5.1, *Compatible Types*, in VXI*plug&play* specification *VPP-4.3.2: VISA Implementation Specification for Textual Languages*.
- IVI-C drivers shall use the type `ViSession` only for session handles.
- The IVI-C drivers shall use the data type declarations in `IviVisaType.h`. Refer to the Appendix C – *Contents of IviVisaType.h* File, for a listing of `IviVisaType.h`.

IVI-COM user-callable methods shall use the data types listed in Table 5-6. Compatible Data Type for IVI Drivers.

- IVI-COM drivers shall use arrays of 32-bit integer values to represent integer arrays, except when typical usage involves arrays of at least one million elements or a class specification requires the use of arrays of 16-bit or 8-bit integer values.
- IVI-COM drivers shall use arrays of 64-bit floating point values to represent floating point arrays, except when typical usage involves arrays of at least one million elements or a class specification requires the use of arrays of 32-bit floating point values.

IVI.NET user-callable methods are free to use any .NET data types, with the following restrictions:

- All public .NET data types shall be Common Language Specification (CLS) compliant, except that the following non-CLS-compliant types are allowed: System.SByte.
- IVI.NET drivers shall use the System namespace data types listed in Table 5-6. Compatible Data Type for IVI Drivers, rather than C# or VB.NET keywords for Boolean, integer, floating point, and string parameters.
- IVI.NET drivers should use arrays of System.Int32 values by default, but may use arrays of System.Byte, System.SByte, System.Int16, or System.Int64. If the array represents waveform or spectrum data, the IWaveform or ISpectrum types shall be supported. Refer to Section 5, *IWaveform<T> Interface*, and Section 7, *ISpectrum<T> Interface*, of *IVI-3.18: IVI.NET Utility Classes and Interfaces Specification* for a description of the IWaveform and ISpectrum types.
- IVI.NET drivers should use arrays of System.Double by default, but may use arrays of System.Single. If the array represents waveform or spectrum data, the IWaveform or ISpectrum types shall be supported. Refer to Section 5, *IWaveform<T> Interface*, and Section 7, *ISpectrum<T> Interface*, of *IVI-3.18: IVI.NET Utility Classes and Interfaces Specification* for a description of the IWaveform and ISpectrum types.

Table 5-6. Compatible Data Types for IVI Drivers

Type Description	C API Type Name	COM API Type Name	.NET API Type Name
Boolean value	`ViBoolean`	`VARIANT_BOOL`	`System.Boolean`
Signed 8-bit integer	`ViInt8`	`CHAR`	`System.SByte`
Array of signed 8-bit integer values	`ViInt8[]`	`SAFEARRAY(CHAR)`	`System.SByte[]`
Unsigned 8-bit integer	`ViByte`	`BYTE`	`System.Byte`
Array of unsigned 8-bit integer values	`ViByte[]`	`SAFEARRAY(BYTE)`	`System.Byte[]`
Signed 16-bit integer	`ViInt16`	`N/A`	`System.Int16`

Array of 16-bit integer values	`ViInt16[]`	`SAFEARRAY(SHORT)`	`System.Int16[]`
Signed 32-bit integer	`ViInt32`	`LONG`	`System.Int32`
Array of 32-bit integer values	`ViInt32[]`	`SAFEARRAY(LONG)`	`System.Int32[]`
Signed 64-bit integer	`ViInt64`	`__int64`	`System.Int64`
Array of 64-bit integer values	`ViInt64[]`	`SAFEARRAY(__int64)`	`System.Int64[]`
Signed Decimal			`System.Decimal`
64-bit floating point number	`ViReal64`	`DOUBLE`	`System.Double`
Array of 64-bit floating point values	`ViReal64[]`	`SAFEARRAY(DOUBLE)`	`System.Double[]`
Array of 32-bit floating point values	`ViReal32[]`	`SAFEARRAY(FLOAT)`	`System.Single[]`
Pointer to a C string	`ViString or ViChar[]`	`BSTR`	`System.String or System.Text. StringBuilder`
An IVI-C or VISA resource descriptor	`ViRsrc`	`BSTR`	`System.String`
An IVI-C or I/O library session handle	`ViSession`	`LONG`	`N/A`
An IVI or VISA return status type	`ViStatus`	`HRESULT`	`N/A`
A constant C string	`ViConstString`	`BSTR`	`System.String`
An attribute ID	`ViAttr`	`N/A`	`N/A`
Enumeration	`ViInt32`	`<Etype>`	`<Etype>`
Interface Reference	`N/A`	`<Itype>`	`<Itype>`

Some older programming environments and operating systems do not support 64-bit integers. For example:

- Microsoft Visual Basic 6.0 does not support 64-bit integers.
- Microsoft Visual C++ 6.0 does not support 64-bit integers as a valid automation type.
- Microsoft COM on Windows 2000 does not support 64-bit integer SAFEARRAYS.

Therefore, users in these environments cannot use IVI drivers with APIs that contain 64-bit integers.

5.14.1 Enumerations

For all types of IVI drivers, enumeration values shall be explicitly specified in the source code for the enumeration. One of the members shall be assigned a value of zero.

5.15 IVI-COM Requirements

This section contains requirements specific to the IVI-COM architecture. Other sections that contain requirements specific to IVI-COM drivers are the following:

- Section 5.5.2, *Requirements for IVI-C, IVI-COM, and* IVI.NET APIs
- Section 5.7, *Use of Shared Components*
- Section 5.10.7, *Multithread Safety*
- Section 5.12, *IVI Error Handling*
- Section 5.14, *Allowed Data Types*

Note: The word “class” used without qualification in this section refers to a COM class, not an IVI instrument class.

5.15.1 IVI-COM Driver Classes

IVI-COM specific instrument drivers shall consist of one or more COM classes.

Exactly one of these classes shall be creatable using the driver’s class factory. This class is called the *main class*.

- The name of the main class shall be *<ComponentIdentifier>*.
- The default interface for the main class shall be *I<ComponentIdentifier>*.
- The main class shall be registered properly as a COM class. Refer to Section 8.1, *IVI-COM Registry Requirements*, in *IVI-3.17: Installation Requirements Specification*, for more details.
- The DllRegisterServer entry point in the driver DLL shall register the main class by adding appropriate entries to the system registry.
- The DllUnregisterServer entry point in the driver DLL shall unregister the main class by removing appropriate entries from the system registry.
- The main class shall implement the standard IVI-COM inherent interfaces, the root IVI-COM class-compliant interface for each instrument class supported by the driver, and the root instrument specific interface. It should implement all other class-compliant and instrument specific interfaces except those implemented by collection classes.
- The driver shall not create other classes independent of the main class nor allow other classes to outlive the main class. This avoids memory leaks.

5.15.2 Standard COM Interfaces

All IVI-COM instrument drivers shall implement the standard COM interfaces ISupportErrorInfo and IProvideClassInfo2. IVI-COM instrument drivers shall return a valid interface pointer when a client application calls QueryInterface on these standard COM interfaces.

If an IVI-COM instrument driver is implemented with several COM classes, all of the classes shall implement ISupportErrorInfo, and the main class shall implement IProvideClassInfo2.

IVI-COM instrument drivers shall support COM errors for IProvideClassInfo2, and shall report such support from the InterfaceSupportsErrorInfo method of the ISupportErrorInfo interface.

5.15.3 IVI-COM Inherent Interfaces

The IVI-COM inherent interfaces are defined in *IVI-3.2: Inherent Capabilities Specification*.

All IVI-COM specific instrument drivers shall implement all of the inherent interfaces except IIviClassIdentity.

If the driver is implemented with several COM classes, only the main class shall implement the inherent interfaces.

The IVI Foundation distributes the IVI-COM driver interfaces as a type library packaged as the sole component in a DLL (`IviDriverTypeLib.dll`).

IVI-COM instrument drivers shall return a valid interface pointer when a client application calls QueryInterface on each of the inherent interfaces.

IVI-COM instrument drivers shall support COM errors for each of the inherent interfaces and shall report such support from the InterfaceSupportsErrorInfo method of the ISupportErrorInfo interface.

5.15.4 IVI-COM Class-Compliant Interfaces

The IVI-COM class-compliant interfaces are defined in the various IVI instrument class specifications.

An IVI-COM class-compliant specific instrument driver shall export all of the class-compliant interfaces defined by the corresponding IVI instrument class specification.

The IVI Foundation distributes the IVI-COM class-compliant interfaces as a series of type libraries, one per class. The type libraries are packaged as the sole component in a DLL (`<ClassName>TypeLib.dll`).

A call to QueryInterface on the main class shall succeed for all class-compliant interfaces, except for interfaces that implement repeated capabilities as collections.

For each COM class that a driver implements, the class shall support COM errors for each of the class-compliant interfaces that it implements, and the class shall report such support from the InterfaceSupportsErrorInfo method of the ISupportErrorInfo interface.

5.15.5 IVI-COM Instrument Specific Interfaces

Instrument specific interfaces shall conform to the standards for IVI-COM interfaces listed in *IVI-3.4: API Style Guide*. Instrument specific interfaces in the same driver shall be related to one another in a hierarchy constructed using interface reference properties, except for hidden interfaces. The root interface of the hierarchy shall be named I<*ComponentIdentifier*>.

Instrument specific interfaces should leverage the syntax of the class-compliant interfaces, where possible.

The type library packaged in the driver DLL for an IVI-COM driver shall include the instrument specific COM interfaces.

A call to QueryInterface on the main class shall succeed for all instrument specific interfaces, except for interfaces that implement repeated capabilities as collections.

For each COM class that a driver implements, the class shall support COM errors for each of the instrument specific interfaces that it implements, and the class shall report such support from the InterfaceSupportsErrorInfo method of the ISupportErrorInfo interface.

5.15.5.1 Instrument Specific Direct I/O API

Specific IVI-COM drivers for devices that use message-based communication shall include a System interface named *`I<ComponentIdentifier>System`*. This interface shall include methods named *`ReadString`*, *`WriteString`*, *`ReadBytes`*, and *`WriteBytes`* and a property named *`IOTimeout`* that allows a client program to get and set the timeout of the underlying I/O. It may optionally include a property named *`DirectIO`* or *`Session`* that provides access to the driver's underlying I/O.

Specific IVI-COM drivers for devices that use message based communication shall include an interface reference property named *`System`* that returns a reference to the *`I<ComponentIdentifier>System`* interface.

Refer to Section 16.3 in *IVI-3.4: API Style Guide*, for the exact syntax of the IVI-COM direct I/O API.

5.15.6 Help Strings

The IDL file for an IVI-COM driver shall contain the following help strings.

- A help string that is associated with the type library. Its format shall be "`IVI <Component Identifier> <Component Revision> Type Library`".
- A help string that is associated with the class itself. Its format shall be "`IVI <Component`

`Identifier> Instrument Driver".`

<`Component Identifier`> is the same as the string returned by the Component Identifier attribute, and <`Component Revision`> is the same as the string returned by the Component Revision attribute.

These two help strings appear prominently in various tools that browse for available COM classes in type libraries.

5.15.7 Threading

IVI-COM drivers shall be registered with the "both" threading model.

IVI-COM drivers shall be implemented to live in the multi-threaded apartment (MTA).

5.15.8 Interface Versioning

IVI-COM drivers shall implement standard IVI-COM interfaces exactly as published by the IVI Foundation. IVI-COM drivers shall not make any modifications to a standard IVI-COM interface and export it as a standard IVI-COM interface. Instrument specific interfaces shall conform to the versioning guidelines defined in Section 5.1, *IVI.NET and IVI-COM Interface Versioning*, in *IVI-3.4: API Style Guide*.

5.15.9 Backwards Compatibility

When an IVI-COM driver is initially released, the driver shall implement the most recently approved version for each standard IVI-COM interface that it exports. The driver may also implement support for older interface versions.

When a driver is modified to add support for a more recent version of a standard IVI-COM interface, the driver shall retain support for the older versions that it already implements for the interface. The driver may also implement support for older versions that it does not already implement for the interface.

When an IVI-COM driver is modified, it shall retain support for all versions of the instrument specific interfaces that it already implements.

5.15.10 Packaging

All IVI-COM specific drivers shall install the following files:

- Microsoft Windows Dynamic Link Library (`.dll`) with a type library.
- Help File (`.hlp`, `.pdf`, `.doc`, `.chm`, or other commonly used help file format)
- Readme Text File (`readme.txt`).

IVI-COM specific drivers may install the following files. These files are necessary to provide support for ANSI-C clients, and may be preferred by some users for C++ clients.

- Header File for type library (`.h`)
- COM GUID Definition File (`_i.c`)

The dynamic link library (`.dll`) filename shall begin with the value that the Component Identifier property returns. For example, if the Component Identifier property returns `Agilent34401a`, the name of the dynamic link library shall be `Agilent34401a.dll`.

Note: The 32-bit dynamic link library name for IVI-COM driver DLLs may be the same as the 64-bit dynamic link library name because COM does not use the PATH environment variable to instantiate COM classes. However, where the IVI-COM driver and a C wrapper are packaged in the same DLL, the C naming standards for DLLs shall be used. See Section 5.15.10.1, *C Wrappers Packaged With IVI-COM Drivers*, for C wrapper naming details.

Table **5-7** lists the DLL requirements when a supplier provides both 32-bit and 64-bit versions of a driver. Table **5-8** lists the DLL requirements when a supplier only provides a 32-bit or only a 64-bit driver. *Required* means the DLL shall be installed for the specified driver bitness and operating system. *Invalid* means the DLL shall not be installed for the specified driver bitness and operating system.

Table 5-7. Required install files when both 32-bit and 64-bit versions of the driver are provided

File type	32-bit driver on a 32-bit OS	32-bit and 64-bit driver on a 64-bit OS
32-bit DLL	Required	Required
64-bit DLL	Invalid	Required

Table 5-8. Required install files when only a 32-bit or a 64-bit version of the driver is provided

File type	Only 32-bit driver provided	Only 64-bit driver provided
32-bit DLL	Required	Invalid
64-bit DLL	Invalid	Required

Note: *IVI-3.17: Installation Requirements Specification* provides instrument driver suppliers with installation requirements for IVI drivers.

The help file shall use a documentation file format readily viewable by customers, such as Windows Help (`.hlp`), Portable Document Format (`.pdf`), compiled HTML (`.chm`), or Microsoft Word document (`.doc`). The filename shall be easily recognizable as associated with the driver.

The `readme.txt` file typically contains installation recommendations, such as those described in Section 2.5.1.5, *Recommendations for Users*, in *IVI-3.17: Installation Requirements Specification*, as well as other information that users may need to know before installing the driver. It may also contain other information a user may find useful before installing the driver.

If there are no special installation recommendations applicable, a statement to that effect shall be included in the `readme.txt` file.

If the header file is installed, the file shall have the same name as the dynamic link library file with the appropriate (`.h`) extension.

If the COM GUID definition file (`_i.c`) is installed, the file shall begin with the value that the Component Identifier property returns with "`_i`" appended before the (`.c`) extension. For example, if the Component Identifier property returns `Agilent34401a`, the name of the COM GUID shall be `Agilent34401a_i.c`.

If a supplier provides both 32-bit and 64-bit versions of a driver, the contents of the include files (`.h`) and the

COM GUID definition files (`_i.c`) shall be the same. This is to ensure that users can easily recompile their application from 32-bit to 64-bit, or vice versa.

The source files for IVI-COM drivers may also be installed. If the source files are installed, at least one of the source files shall have the same name as the dynamic link library with an appropriate extension. If the source files are installed, all necessary files for rebuilding the driver shall be installed.

If the DLL requires the presence of other DLLs, the IVI-COM specific driver may also install the additional DLLs.

If the IVI-COM driver installs multiple files of the same type, the additional files may use different filenames. Unless an additional file is shared between drivers for instruments from multiple manufacturers, the additional files should begin with the two-character abbreviation for the instrument manufacturer reserved in the VXI*plug&play* Alliance specification *VPP-9: Instrument Vendor Abbreviations*.

5.15.10.1 C Wrappers Packaged With IVI-COM Drivers

To help identify files as belonging to a particular driver, driver files and directories make use of a unique driver identifier. For IVI-C drivers, this is the driver prefix. For IVI-COM drivers, this is the component identifier. When creating and distributing an IVI-C wrapper on top of an IVI-COM driver, special consideration is needed for prefix and component identifier usage. Most of the supported ADEs can find IVI-C wrapper files more easily if all the driver files are consistently named using the IVI-C prefix. Therefore, an IVI-C wrapper packaged with an IVI-COM driver shall comply with the following rules:

- If a supplier provides both 32-bit and 64-bit versions of a driver, the driver always installs C wrappers for both versions of the driver or neither version of the driver.
- For each supported operating system bitness, the IVI driver installer shall create a single driver specific directory in the `<IVIStandardRootDir>\Drivers` directory for both the non-dispersed IVI-COM driver files and the non-dispersed IVI-C wrapper files. The directory shall be named using the IVI-C wrapper prefix.
- All installed driver files, except the .NET PIAs, shall be named using the IVI-C wrapper prefix rather than the IVI-COM component identifier. The .NET PIAs shall be named using the IVI-COM component identifier, as though no IVI-C wrapper were present
- The IVI-C wrapper shall be implemented in the IVI-COM driver DLL. The dynamic link library (.dll) file name shall follow the requirements specified in Section 5.16.14, *Packaging*, for IVI-C driver dynamic link library names.
- The IVI driver installer shall not install the IVI-COM `.h` and `_i.c` files. The IVI-C wrapper files provide the necessary include files for C users.
- The IVI driver installer shall create a single software module entry in the IVI configuration store for both the IVI-COM driver and the IVI-C wrapper. The Name, ModulePath, Prefix, and ProgID attributes, as well as the Published APIs collection, shall follow the requirements specified in Section 5.3, *Details on Software Module Entries in the IVI Configuration Store*, in *IVI-3.17: Installation Requirements Specification*.

The IVI-COM driver's component identifier and the IVI-C wrapper prefix need not be the same, though they may be.

Because files names are based on the IVI-C wrapper prefix, the file naming and installation requirements differ slightly from the IVI-COM driver packaging requirements specified in Section 5.15.10, *Packaging*. Table **5-9** shows example pathnames based on an IVI-COM driver for which the component identifier is "Agilent34401A" and the prefix is "Ag34401a". All pathnames are relative to the `<IVIStandardRootDir>`.

Table 5-9. Example File Names for COM Drivers Packaged with C Wrappers

File	PathName relative to `<IVIStandardRootDir>\`
.NET interop DLL	Bin\Primary Interop Assemblies\Agilent.Agilent34401A.Interop.dll
.NET interop help file	Bin\Primary Interop Assemblies\Agilent.Agilent34401A.Interop.xml
Driver DLL	Bin\Ag34401a.dll (32-bit dynamic link library) Bin\Ag34401a_64.dll (64-bit dynamic link library)
Driver directory	Drivers\Ag34401a
Driver help file	Drivers\Ag34401a\Ag34401a.chm
Driver help index	Drivers\Ag34401a\Ag34401a.chi
Driver .fp file	Drivers\Ag34401a\Ag34401a.fp
Driver .sub file	Drivers\Ag34401a\Ag34401a.sub
Readme file	Drivers\Ag34401a\Readme.txt
C header	Include\Ag34401a.h (for C wrapper only)
C import library	Lib\msc\Ag34401a.lib

Note: A driver supplier that previously distributed an IVI-COM driver without a C wrapper and then distributes the IVI-COM driver with a C wrapper needs to be aware of the following:

- If the name of the driver DLL changes, the driver installer shall check for the presence of both the old DLL name as well as the new DLL name to determine whether the driver already exists on the system (as specified in Section 5.1.2, *Detecting the Presence, Vendor, and Version of an IVI-COM or IVI-C Driver*, in *IVI-3.17: Installation Requirements Specification*).
- If the name of the Software Module in the configuration store changes, the driver installer shall update existing IVI driver session configuration entries that refer to the old Software Module to refer to the new Software Module.
- If the old driver installed the generated `.h` and `_i.c` files, then user applications that depend on those files will have their new builds broken. Since built binaries do not depend on these files, existing built binaries will not break.

An IVI-COM driver supplier that intends to distribute a C wrapper in a later release can avoid these issues. The first two issues can be avoided by choosing a component identifier that is also a valid IVI-C driver prefix, as defined in Section 5.16.4, *Prefixes*. The third issue can be avoided by not shipping the generated `.h` and `_i.c` files with the original release of the IVI-COM driver.

5.16 IVI-C Requirements

This section contains requirements specific to the IVI-C architecture. Other sections that contain requirements specific to IVI-C drivers are the following:

- Section 5.5.2, *Requirements for IVI-C, IVI-COM, and* IVI.NET APIs
- Section 5.7, *Use of Shared Components*
- Section 5.10.7, *Multithread Safety*
- Section 5.12, *IVI Error Handling*
- Section 5.14, *Allowed Data Types*

5.16.1 Separate Sessions for IVI-C Class and IVI-C Specific Drivers

When an IVI-C class driver loads an IVI-C class-compliant specific driver, the class driver and specific driver shall create separate sessions. The class driver Initialize function shall return the class driver session. The class driver shall return the specific driver session when the application program calls the Get Specific Driver C Handle function.

5.16.2 Function Prototypes

The general function prototype for all IVI-C drivers shall be in the form of the following:

```
ViStatus _VI_FUNC <function name> ( <parameter> [, <parameter>]) ;
```

The first element of an IVI-C instrument driver function prototype shall be the specification of the type of the value the function returns. This return value shall always be of type `ViStatus`. The `IviVisaType.h` include file defines `ViStatus`. A listing of `IviVisaType.h` can be found in the Appendix C – *Contents of IviVisaType.h* File.

The second element of an IVI-C instrument driver function prototype shall be the function qualifier, which provides information to ADEs about access options and conventions. The `_VI_FUNC` macro shall be used for the function qualifier. The `_VI_FUNC` macro is defined in the `IviVisaType.h` include file. The definition of the `_VI_FUNC` macro varies based on the target operating system and ADE.

The remaining elements of the function prototype shall be the function name, the parameter list, and the terminating semicolon. The function name shall begin with the driver prefix as defined in Section 5.16.4, *Prefixes*. The number of characters in a function name, excluding the prefix, shall not exceed 79 characters. Additional restrictions on function and parameter names are defined in *IVI-3.4: API Style Guide*.

All IVI-C driver functions that require a session handle to identify the instrument shall define the session parameter to be of type `ViSession` and specify it as the first parameter in the argument list.

IVI-C driver functions shall not use variable parameter lists.

IVI-C driver functions shall be limited to a maximum of 18 parameters.

5.16.3 Accessing Attributes

An IVI-C driver shall export the Set Attribute <type> and Get Attribute <type> functions to provide access to all non-private attributes of the driver.

5.16.4 Prefixes

Each IVI-C specific driver shall have a prefix that uniquely identifies the driver. The prefix shall begin with a two-character vendor code as defined in the VXI*plug&play* specification *VPP-9: Instrument Vendor Abbreviations*, followed by characters that uniquely identify the driver. The prefix shall be a maximum of 31 characters.

Each IVI-C class driver shall use the class name specified in the IVI class specification with which the driver complies, such as `IviScope`, as the class prefix.

All function names, attribute names, attribute value names, and parameter value names that an IVI-C driver exports shall use the driver's prefix, regardless of whether they are inherent, class-defined, instrument specific, or vendor specific. The names of all class-defined, instrument specific, and vendor specific status codes that an IVI-C driver can return shall begin with the driver's prefix. An IVI-C driver shall export the common status codes it can return using the names that *IVI-3.2: Inherent Capabilities Specification* and the IVI shared component specifications define.

The IVI Foundation specifications use the terms *`PREFIX`* and *`Prefix`* in formal syntax specifications. Wherever the literal string `PREFIX` appears in formal syntax, the actual driver prefix for the IVI-C driver shall be substituted using uppercase. Wherever the literal string *`Prefix`* appears in formal syntax, the actual driver prefix for the IVI-C driver shall be substituted using a consistent case sensitivity. It is recommended that all IVI-C specific drivers use lowercase prefixes when *`Prefix`* appears in formal syntax.

Examples of prefix usage are given below for the Tektronix VX4790 Arbitrary Waveform Generator:

`tkvx4790_init()` – Initialize function

`TKVX4790 ATTR_INSTRUMENT_FIRMWARE_REVISION` – Inherent attribute

`TKVX4790_VAL_WFM_SINE` – Value for use with the `TKVX4790_ATTR_FUNC_WAVEFORM` attribute

5.16.5 IVI-C Attribute IDs

Each attribute in an IVI-C driver shall have a unique integer ID. The include file for an IVI-C driver shall define attribute IDs using macros. The number of characters in an attribute macro name, including the prefix, shall not exceed 100 characters.

To prevent attribute ID conflicts within the API of an IVI-C driver, the IVI Foundation has established ID value ranges for different types of attributes. Table 5-10. Attribute ID Base Values for IVI-C Drivers lists these ranges. All IVI-C drivers shall comply with the ranges in Table 5-10. Attribute ID Base Values for IVI-C Drivers.

Table 5-10. Attribute ID Base Values for IVI-C Drivers

Attribute ID Base	Value
`IVI_ATTR_BASE`	`1000000`
`IVI_INHERENT_ATTR_BASE`	`IVI_ATTR_BASE + 50000`
`IVI_INSTR_SPECIFIC_ATTR_BASE`	`IVI_ATTR_BASE + 150000`
`IVI_CLASS_ATTR_BASE`	`IVI_ATTR_BASE + 250000`
`IVI_VENDOR_CLASS_EXT_ATTR_BASE`	`IVI_ATTR_BASE + 350000`
`IVI_VENDOR_INHERENT_EXT_ATTR_BASE`	`IVI_ATTR_BASE + 450000`
`IVI_MODULE_PRIVATE_ATTR_BASE`	`IVI_ATTR_BASE + 550000`
`IVI_LXISYNC_ATTR_BASE`	`IVI_ATTR_BASE + 950000`
`Reserved`	`IVI_ATTR_BASE + 1050000`

- `IVI_INHERENT_ATTR_BASE` is the base value from which the ID values for IVI inherent attributes are defined. Refer to Section 10, *IVI Inherent Attribute ID Definitions*, in *IVI-3.2: Inherent Capabilities Specification* for the list of ID values for inherent attributes.
- `IVI_INSTR_SPECIFIC_ATTR_BASE` is the base value from which an IVI-C specific driver defines the ID values of the instrument specific attributes that it exports, excluding any vendor specific attributes. Notice that the ID values for instrument specific attributes are not unique from one IVI-C driver to another.
- `IVI_CLASS_ATTR_BASE` is the base value from which the IVI class specifications define attribute ID values. Notice that the ID values for class-defined attributes are not unique from one IVI class to another.
- `IVI_VENDOR_CLASS_EXT_ATTR_BASE` is the base value from which an IVI driver supplier defines ID values for vendor specific attributes that a particular IVI-C custom class driver and set of corresponding IVI-C specific drivers export. This is useful if a software supplier wants to add vendor specific extensions to a class specification that the IVI Foundation defines. Notice that the attribute ID values are not unique from one vendor defined class to another.

- `IVI_VENDOR_INHERENT_EXT_ATTR_BASE` is the base value from which a software supplier defines ID values for vendor specific attributes that are inherent to all its IVI-C specific drivers and/or IVI-C class drivers. Notice that the ID values for vendor specific inherent attributes are not unique from one vendor to another.
- `IVI_MODULE_PRIVATE_ATTR_BASE` is the base value from which any IVI software module can define hidden attributes. Hidden attributes are attributes that the software module uses internally and does not export.
- `IVI_LXISYNC_ATTR_BASE` is the base value from which the IVI LxiSync API defines attributes. Refer to *IVI 3-15 IviLxiSync Specification* for details.

The following is an example declaration of the ID for an instrument specific attribute:

```
#define HP34401A_ATTR_MATH_OPERATION (IVI_INSTR_SPECIFIC_ATTR_BASE + 2)
```

The following is an example declaration of the ID for a class-defined attribute:

```
#define IVIDCPWR_ATTR_VOLTAGE_LEVEL  (IVI_CLASS_ATTR_BASE + 1)
```

Refer to Section 5.16.7, *Include File*, for rules on declaring attribute IDs for inherent and class-defined attributes in the include files for IVI-C class-compliant specific drivers.

5.16.6 IVI-C Status Codes

The include file for an IVI-C driver shall define status codes using macros. Section 5.12, *IVI Error Handling,,* describes the formats and ranges for status code values.

The following is an example declaration of an instrument specific error using the base identifiers described in Section 5.12.1.2, *Base Values*:

```
#define TKDS30XX_ERROR_OPTION_NOT_INSTALLED (IVI_SPECIFIC_ERROR_BASE + 2)
```

The following is an example declaration of a class-defined warning using the base identifiers described in Section 5.12.1.2, *Base Values*:

```
#define IVIDMM_WARN_OVER_RANGE    (IVI_CLASS_WARN_BASE + 1)
```

Refer to Section 5.16.7, *Include File*, for rules on declaring status codes for class-defined errors and warnings in the include files for IVI-C class-compliant specific drivers.

5.16.7 Include File

The include file for an IVI-C driver shall contain C prototypes for all functions that the driver exports.

The include file for an IVI-C driver shall contain C constant definitions for all attributes and attribute values that the driver exports.

The include file for an IVI-C driver shall contain C constant definitions for all status codes that the driver can return.

The include file for an IVI-C driver shall define constants as macros.

The include file for an IVI-C driver shall allow itself to be included multiple times in the same source file without generating compiler errors or warnings.

5.16.8 Function Panel File

The function panel file for an IVI-C driver shall comply with either version 4.1, 5.1, or 9.0 of the format specified in Section 6, *Function Panel File Format*, in the VXI*plug&play* specification *VPP-3.3: Instrument Driver Interactive Developer Interface Specification*. Note: IVI-C instrument drivers shall not use version 9.0 until after June 2007.

The function panel file for an IVI-C driver shall comply with Sections 3.3 through 3.5.5 in the VXI*plug&play* specification *VPP-3.3: Instrument Driver Interactive Developer Interface Specification*, except for RULE 3.4 and RULE 3.9. The following are the IVI replacements for these rules, respectively:

- The visual representation of the return value of an IVI-C function shall be placed in the lower right hand corner of that function's function panel. The control shall be labeled "Status" unless it is a function that adheres to the Section 3.1.2.1, *Additional Compliance Rules for C Functions with ViChar Array Output Parameters*, in *IVI-3.2: Inherent Capabilities Specification.* In this case the control shall be named "Status or Required Size".
- If an IVI function panel contains controls whose values are defined constants, the definitions of these constants shall be in the driver's include file or in a nested include file.

5.16.9 Function Tree Organization

For all IVI inherent functions that an IVI-C driver exports, the function tree for the driver should follow the inherent function hierarchy, insofar as it is practical. The inherent function hierarchy is specified in Section 4.3, *C Inherent Capabilities*, in *IVI-3.2: Inherent Capabilities Specification*. For all class-defined functions that an IVI-C class driver or IVI-C class-compliant specific driver exports, the function tree for the driver should follow the class-defined function hierarchy, insofar as it is practical. The class-defined function hierarchy is specified in the IVI class specification.

All IVI-C drivers shall comply with the guidelines in Section 13.1, *C Function Hierarchy*, in *IVI-3.4: API Style Guide* on grouping functions into a hierarchy.

5.16.9.1 Extending the Function Tree for Instrument Specific Functions

An IVI-C specific driver shall augment the function tree with instrument specific functions according to the following rules:

- No instrument specific functions shall appear at the root level in the function tree.
- Instrument specific categories may appear at the root level.
- Instrument specific functions and categories may appear within a category specified in the inherent function hierarchy.
- For an IVIC classcompliant specific driver, instrument specific functions and categories may appear within a category specified in the classdefined function hierarchy.

5.16.10 Sub File

The sub file for an IVI-C driver shall comply with the format specified in Section 7, *Function Panel Sub File Format*, in the VXI*plug&play* specification *VPP-3.3: Instrument Driver Interactive Developer Interface Specification*.

5.16.11 Attribute Hierarchy

For all IVI inherent attributes that an IVI-C driver uses, the attribute hierarchy for the driver should follow the inherent attribute hierarchy, insofar as it is practical. The inherent attribute hierarchy is specified in Section 4.3, *C Inherent Capabilities*, in *IVI-3.2: Inherent Capabilities Specification*.

For all class-defined attributes that an IVI-C class or IVI-C class-compliant specific driver uses, the attribute hierarchy for the driver should follow the class-defined attribute hierarchy, insofar as it is practical. The class-defined attribute hierarchy is specified in the IVI class specification.

All IVI-C drivers shall comply with the guidelines in Section 13.2, *C Attribute Hierarchy*, in *IVI-3.4: API Style Guide* on grouping attributes into a hierarchy.

5.16.11.1 Extending the Attribute Hierarchy for Instrument Specific Attributes

An IVI-C specific driver shall augment the attribute hierarchy with instrument specific attributes according to the following rules:

- No instrument specific attributes shall appear at level 1.
- Instrument specific categories may appear at level 1.
- Instrument specific and vendor specific attributes or categories may appear within the inherent attribute hierarchy.
- For an IVI-C class-compliant specific driver, instrument specific attributes and categories may appear within a category specified in the class-defined attribute hierarchy.

5.16.12 Instrument Specific Direct I/O API

Specific IVI-C drivers for devices that use message-based communication shall include <*Prefix*>_ViRead and <*Prefix*>_ViWrite functions and an attribute named <*Prefix*>_ATTR_SYSTEM_IO_TIMEOUT. It may optionally include an attribute named <*Prefix*>_ATTR_SYSTEM_IO_SESSION that provides access to the driver's underlying I/O.

The function hierarchy of specific IVI-C drivers for devices that use message-based communication shall include a level 1 category named *System*. Functions related to direct I/O, including the read and write functions, shall be placed in this level of the hierarchy.

The attribute hierarchy of specific IVI-C drivers for devices that use message-based communication shall include a level 1 category named *System*. Attributes related to direct I/O, including the timeout and session attributes, shall be placed in this level of the hierarchy.

Refer to Section 16.2 in *IVI-3.4: API Style Guide*, for the exact syntax of the IVI-C direct I/O API.

5.16.13 Backwards Compatibility

The versioning guidelines defined in Section 5.2, *IVI-C Interface Versioning*, in *IVI-3.4: API Style Guide*, guarantee that the inherent and class-compliant capabilities of the driver are backwards compatible with previous versions of the driver.

When an IVI-C driver is initially released, the driver shall comply with the most recent version of the IVI Foundation specifications.

When a driver is modified to add support for a more recent version of an IVI Foundation specification, the driver shall comply with the most recent versions of all IVI Foundation specifications with which it claims compliance.

The instrument specific capabilities of IVI-C instrument drivers shall conform to the versioning guidelines defined in Section 5.2, *IVI-C Interface Versioning*, in *IVI-3.4: API Style Guide*. This ensures that the instrument specific capabilities of the driver are backwards compatible.

5.16.14 Packaging

All IVI-C specific drivers shall install the following files:

- Include File (`.h`)
- Microsoft Windows Dynamic Link Library (`.dll`) with a type library
- Microsoft-compatible DLL Import Library File (`.lib`)
- Function Panel File (`.fp`)

- Attribute Information File (`.sub`)
- Help File (`.hlp`, `.pdf`, `.doc`, `.chm`, or other commonly used help file format)
- Readme Text File (`readme.txt`).

The include file (`.h`), import library file (`.lib`), function panel file (`.fp`), and attribute information file (`.sub`) shall use the same filename except for the filename extension. The filename shall begin with the value that the Specific Driver Prefix attribute returns.

If a supplier provides both 32-bit and 64-bit versions of a driver, the contents of the include files (`.h`), function panel files (`.fp`), and attribute information files (`.sub`) shall be the same. This is to ensure that users can easily recompile their application from 32-bit to 64-bit, or vice versa.

If a supplier provides both 32-bit and 64-bit versions of a driver, the driver shall install two import library files, one for compiling 32-bit applications and one for compiling 64-bit applications. If a supplier only provides a 32-bit or only a 64-bit driver, the driver shall install only one import library. Table **5-11** lists the valid combinations of DLL and import libraries when a supplier provides both 32-bit and 64-bit versions of a driver. Table **5-12** lists the valid combination of DLL and import libraries when a supplier only provides a 32-bit or only a 64-bit driver. *Required* means the file shall be installed for the specified driver bitness and operating system. *Invalid* means the file shall not be installed for the specified driver bitness and operating system.

Table 5-11. Required install files when both 32-bit and 64-bit versions of the driver are provided

File type	32-bit driver on a 32-bit OS	32-bit and 64-bit driver on a 64-bit OS
32-bit DLL	Required	Required
64-bit DLL	Invalid	Required
32-bit Import Library File	Required	Required
64-bit Import Library File	Required	Required

Table 5-12. Required install files when only a 32-bit or a 64-bit version of the driver is provided

File type	Only 32-bit driver available	Only 64-bit driver available
32-bit DLL	Required	Invalid
64-bit DLL	Invalid	Required
32-bit Import Library File	Required	Invalid
64-bit Import Library File	Invalid	Required

The 32-bit dynamic link library (`.dll`) filename shall have the same root name as the include file, except that "`_32`" may be appended. For example, if the name of the include file is `ag34401a.h`, the name of the 32-bit dynamic link library shall be `ag34401a.dll` or `ag34401a_32.dll`. The latter form is consistent with the VXI*plug&play* specifications.

The 64-bit dynamic link library (`.dll`) filename shall have the same root name as the include file, except that "`_64`" shall be appended. For example, if the name of the include file is `ag34401a.h`, the name of the 64-bit dynamic link library shall be `ag34401a_64.dll`.

Note: The 32-bit dynamic link library name needs to be distinct from the 64-bit dynamic link library name because both files are in the system path on Windows 7 (64-bit), Windows 8 (64-bit), Windows 10 (64-bit) and Windows 11. Neither the PATH environment variable nor the operating system's interpretation of the PATH environment variable is sensitive to the bitness of the application.

The help file shall use a documentation file format readily viewable by customers, such as Windows Help (`.hlp`), Portable Document Format (`.pdf`), compiled HTML (`.chm`), or Microsoft Word document (`.doc`). The filename shall be easily recognizable as associated with the driver.

The `readme.txt` file typically contains installation recommendations, such as those described in Section 2.5.1.5, *Recommendations for Users*, in *IVI-3.17: Installation Requirements Specification*, as well as other information that users may need to know before installing the driver. It may also contain other information a user may find useful before installing the driver.

If there are no special installation recommendations applicable, a statement to that effect shall be included in the `readme.txt` file.

The source files for IVI-C drivers may also be installed. If the source files are installed, at least one of the source files shall have the same name as the include file with an appropriate extension. If the source files are installed, all necessary files for rebuilding the driver shall be installed.

A Borland-compatible DLL import library file may also be installed. If the Borland-compatible DLL import library file is installed, the file shall have the same name as the Microsoft-compatible DLL import library file.

If the DLL requires the presence of other DLLs, the IVI-C specific driver may also install the additional DLLs.

If the IVI-C driver installs multiple files of the same type, the additional files may use different filenames. Unless an additional file is shared between drivers for instruments from multiple manufacturers, the additional files should begin with the two-character abbreviation for the instrument manufacturer reserved in the VXI*plug&play* Alliance specification *VPP-9: Instrument Vendor Abbreviations*.

5.17 IVI.NET Requirements

Note: To ensure that IVI.NET driver quality is the highest possible, the registration of all IVI.NET drivers released before 9 June, 2011 requires the approval of the IVI Foundation IVI.NET working group. The IVI Foundation will treat all changes to IVI.NET material in the specifications and IVI.NET shared components as editorial changes until the registration of the first driver that implements that particular set of IVI.NET interfaces.

This section contains requirements specific to the IVI.NET architecture. Other sections that contain requirements specific to IVI.NET drivers are the following:

- Section 5.5.2, *Requirements for IVI-C, IVI-COM, and IVI.NET APIs*
- Section 5.7, *Use of Shared Components*
- Section 5.10.7, *Multithread Safety*
- Section 5.12, *IVI Error Handling*
- Section 5.14, *Allowed Data Types*

Note: The word “class” used without qualification in this section refers to a .NET class, not an IVI instrument class.

5.17.1 IVI.NET Driver Classes

All IVI.NET APIs shall comply with the .NET Common Language Specification (CLS). This provides a measure of language independence when creating and using .NET drivers.

IVI.NET specific instrument drivers shall consist of one or more .NET classes.

One class in the driver assembly, called the *main class*

- *S*hall implicitly implement IServiceProvider.
    - GetService shall return, at a minimum, a reference to the root IVI.NET class-compliant interface for each instrument class supported by the driver,
- Shall implicitly or explicitly implement IIviDriver. If the main class explicitly implements IIviDriver, it must comply with the following provisions to ensure IVI inherent capabilities are discoverable in common .NET programming tools:
    - The main class shall explicitly implement (i.e., privately implement) all members of IIviDriver.
    - Additionally, the main class shall directly implement public members with the same names and equivalent functionality as the IIviDriver members.
    - The specific driver shall implement classes or interfaces that derive from the interfaces returned by IIviDriver interface reference properties.
        - The main class shall return these classes or interfaces through properties equivalent to the IIviDriver interface reference properties.
    - The classes or interfaces that derive from the interfaces returned by IIviDriver interface reference properties shall directly implement public members with the same names and equivalent functionality as the members of the interfaces returned by IIviDriver interface reference properties.
    - Explicit implementation example:
        - `NIDmm` is the main driver class.
        - `NIDmmDriverUtility` includes everything in `IIviDriverUtility` and additional methods or properties.
        - `NIDmm` explicitly implements `IIviDriver`, including the `IIviDriverUtility` interface reference property.

```
public class NIDmm : IIviDriver, IIviDmm, IDisposable
{
   public NIDmmDriverUtility Utility {…}
   IIviDriverUtility IIviDriver.Utility {…}
   …
}

public class NIDmmDriverUtility : IIviDriverUtility{…}
```

    - Implicit implementation example:
        - `NIDmm` is the main driver class.
        - In this scenario there is no need for the `NIDmmDriverUtility` class.

- `NIDmm` implicitly implements `IIviDriver`, including the `IIviDriverUtility` interface reference property.

```
public class NIDmm : IIviDriver, IIviDmm, IDisposable
{
   public IIviDriverUtility Utility {…}
   …
 }
```

- Shall implement the driver's constructors,
- Shall implicitly implement IDisposable, and shall call Dispose when the driver object is destroyed.

The name of the main class shall be *<ComponentIdentifier>*. The main driver assembly and all dependant assemblies shall be installed in the Global Assembly Cache (GAC).

The main driver assembly shall be registered at installation so that it appears on the Microsoft Visual Studio list of .NET references.

5.17.2 IVI.NET Namespaces

The namespace for the IVI.NET inherent capabilities described in *IVI-3.2: Inherent Capabilities Specification*, and *IVI-3.18: IVI.NET Utility Intefaces and Classes Specifciation*, shall be "Ivi.Driver".

The namespace for an IVI instrument class shall be *Ivi.<ClassType>* For example, "Ivi.Dmm".

The namespace for any other IVI.NET component owned by the IVI Foundation shall start with "Ivi.". The next element of the namespace shall be the name of the component. For example, "Ivi.SessionFactory".

The namespace of IVI.NET instrument drivers shall be *<CompanyName>.<ComponentIdentifier>* or *<CompanyName>.<Technology>.<ComponentIdentifier>*. For example, "Agilent.Agilent34411" or "NationalInstruments.ModularInstruments.NIDmm". Values for *<Technology>* are determined by vendors, not by the IVI Foundation.

All namespaces shall use Pascal casing.

5.17.3 Standard .NET Error Reporting

All IVI.NET instrument drivers shall consistently use the standard .NET exception mechanism to report errors. Neither return values nor out parameters shall be used to return error information. Refer to Section 5.12.2, *IVI.NET Error Handling* for more details.

5.17.4 IVI.NET .NET Interfaces

The main class of each IVI.NET specific instrument driver shall implicitly implement IDisposable and IServiceProvider. Other interfaces may either be implicitly or explicitly implemented. If the main class explicitly implements IIviDriver, the implementation must follow the provisions specified in Section 5.17.1, *IVI.NET Driver Classes*.

5.17.5 IVI.NET Inherent Interfaces

The IVI.NET inherent interfaces are defined in *IVI-3.2: Inherent Capabilities Specification*. The IVI Foundation distributes the IVI.NET driver interfaces as an assembly packaged as the sole component in a DLL (`Ivi.Driver.dll`).

The main class of each IVI.NET specific instrument driver may either explicitly or implicitly implement IIviDriver and other IVI.NET Inherent Interfaces. If the main class explicitly implements IIviDriver, the implementation must follow the provisions specified in Section 5.17.1, *IVI.NET Driver Classes*.

The inherent Initialize method is not explicitly implemented in IVI.NET. Instead, the Initialize method shall be implemented as part of the specific driver's constructor(s).

5.17.6 IVI.NET Class-Compliant Interfaces

The IVI.NET class-compliant interfaces are defined in the various IVI instrument class specifications.

An IVI.NET class-compliant specific instrument driver shall implement all the class-compliant interfaces defined by the corresponding IVI instrument class specification.

The IVI Foundation distributes the IVI.NET class-compliant interfaces as a series of assemblies, one per class. The assemblies are packaged as the sole component in a DLL (`Ivi.<ClassType>.dll`).

5.17.7 IVI.NET Instrument Specific Classes and Interfaces

Instrument specific classes and interfaces shall conform to the standards for IVI.NET instrument specific classes and interfaces listed in *IVI-3.4: API Style Guide*. Instrument specific classes and interfaces in the same driver shall be related to one another in a hierarchy constructed using reference properties. The root of the hierarchy shall be the main class <*ComponentIdentifier*>.

Instrument specific classes and interfaces should leverage the syntax of the class-compliant interfaces, where possible.

The public instrument specific classes and interfaces are part of a specific driver, and they shall use the same namespace as the driver.

5.17.7.1 Instrument Specific Direct I/O API

Specific IVI.NET drivers for devices that use message-based communication shall include a System interface named *`I<ComponentIdentifier>System`*. This interface shall include methods named *`ReadString`*, *`WriteString`*, *`ReadBytes`*, and *`WriteBytes`* and a property named *`IOTimeout`* that allows a client program to get and set the timeout of the underlying I/O. It may optionally include a property named *`DirectIO`* or *`Session`* that provides access to the driver's underlying I/O.

Specific IVI.NET drivers for devices that use message-based communication shall include an interface reference property named *`System`* that returns a reference to the *`I<ComponentIdentifier>System`* interface.

Refer to Section 16.1 in *IVI-3.4: API StyleGuide*, for the exact syntax of the IVI.NET direct I/O API.

5.17.8 Repeated Capability Interfaces

IVI.NET interfaces that represent a single instance of a repeated capability (a collection member) shall derive from Ivi.Driver.IIviRepeatedCapabilityIdentification. Refer to Section 11, *Repeated Capability Collection Base Classes*, in *IVI-3.18: IVI.NET Utility Classes and Interfaces Specificatio*n,for details about this interface.

IVI.NET repeated capability collection interfaces shall derive from Ivi.Driver.IIviRepeatedCapabilityCollection. Refer to Section 11, *Repeated Capability Collection Base Classes*, in *IVI-3.18: IVI.NET Utility Classes and Interfaces Specification*,for details about this interface.

5.17.9 Assembly Level Attributes

.NET Assembly Attributes shall be included for:

- AssemblyTitle. AssemblyTitle shall be the file name of the assembly.

- AssemblyDescription. AssemblyDescription shall be free-form; it is required to exist, but it does not have a particular value.
- AssemblyCompany. AssemblyCompany shall be the driver vendor's company name.
- AssemblyCulture. AssemblyCulture shall be "" for assemblies that are not globalized. If localized, AssemblyCulture shall be the value of the culture to which the assembly is localized.
- AssemblyProduct. AssemblyProduct shall be `"IVI` *`<Component Identifier>`* `.NET Assembly"`.
- AssemblyVersion. AssemblyVersion shall be the same as the version resource values for the same item, as defined in section 5.19.
- AssemblyFileVersion. AssemblyFileVersion shall be the same as the version resource values for the same item, as defined in section 5.19.

.NET Assembly Attributes should be included (if applicable) for:

- AssemblyCopyright.
- AssemblyTrademark.

The .NET assembly IntelliSense help shall include comments for each IVI.NET class, struct, interface, enumeration type, enumeration value, exception, method, property, and parameter.

The comments may be inserted in source code as .xml comments, from which the compiler generates an XML IntelliSense file. Other ways of generating the IntelliSense file are permitted.

5.17.10 Interface Versioning

IVI.NET drivers shall implement standard IVI.NET interfaces exactly as published by the IVI Foundation. IVI.NET drivers shall not make any modifications to a standard IVI.NET interface and export it as a standard IVI.NET interface. Instrument specific classes and interfaces shall conform to the versioning guidelines defined in Section 5.1, *IVI.NET and IVI-COM Interface Versioning*, of *IVI-3.4: API Style Guide*.

5.17.11 Backwards Compatibility

When an IVI.NET driver is initially released, the driver shall implement standard IVI.NET interfaces contained in the most recent version of the standard IVI.NET assemblies.

When an IVI.NET driver is modified, it shall not modify classes and interfaces in such a way as to break backwards compatibility.

In order to facilitate backwards compatibility, policy and .config files shall be delivered, as appropriate, to point users of older driver versions to the newer one.

5.17.12 Packaging

All IVI.NET specific drivers shall install the following files:

- Microsoft Windows Dynamic Link Library (`.dll`).
- An XML IntelliSense file.
- Help File (`.pdf`, `.doc`, `.chm`, or other commonly used help file format).
- Readme Text File (`readme.txt`).

The dynamic link library (.dll) filename for the main driver executable shall be <*Namespace*>.<*FwkVerShortName*> followed by ".dll. For example, if the driver namespace is `Agilent.Ag34401A` and the driver was built using version 2.0.50727 of the .NET Framework, the name of the dynamic link library shall be `Agilent.Ag34401A.Fx20.dll`.

The help file shall use a documentation file format readily viewable by customers, such as Portable Document Format (`.pdf`), compiled HTML (`.chm`), or Microsoft Word document (`.doc`). The filename shall be easily recognizable as associated with the driver.

The `readme.txt` file typically contains installation recommendations, such as those described in Section 2.5.2.5, *Recommendations for Users*, in *IVI-3.17: Installation Requirements Specification*, as well as other information that users may need to know before installing the driver. It may also contain other information a user may find useful before installing the driver.

If there are no special installation recommendations applicable, a statement to that effect shall be included in the `readme.txt` file.

The source files for IVI.NET drivers may also be installed. If the source files are installed, they shall be installed with all of the instructions and files necessary to correctly build the driver. For example, instructions might need to document changes such as:

- References that need to be corrected to the target PC.
- Public/private signing modifications, either to eliminate signing or change the keys.

If the DLL requires the presence of other DLLs, the IVI.NET specific driver may also install the additional DLLs.

If the IVI.NET driver installs multiple files of the same type, created by the driver supplier, the additional files may use different filenames. The additional files should begin with the name of the driver supplier.

All .NET Assemblies distributed with an IVI driver shall be signed with the vendor’s public/private key pair.

The installation directories for IVI.NET drivers will vary by the supported .NET Framework and the driver version. For details on IVI.NET driver installation, refer to *IVI-3.17: Installation*.

5.17.13 Signing

The driver supplier shall create a key pair for its organization using the Strong Name tool. The driver supplier shall publish the public key and keep the private key secret and secure. The same key shall be used to sign all the IVI.NET drivers and primary interop assemblies supplied by a driver supplier.

Driver suppliers shall sign their drivers.

5.18 Wrapper Packaging

If a C or COM wrapper is packaged with the native driver, the wrapper and its type library shall be in the same DLL file as the native driver. Refer to Section 5.15.10, *Packaging*, for packaging requirements for IVI-COM drivers. Refer to Section 5.16.14, *Packaging*, for packaging requirements of IVI-C drivers. Refer to Section 5.3, *Details on Software Module Entries in the IVI Configuration Store*, in *IVI-3.17: Installation Requirements Specification*, for configuration store requirements for IVI drivers.

If a C or COM wrapper is packaged separately from the native driver, all wrapper file names shall be the prefix or component identifier of the native driver followed by “CWrapper” or “COMWrapper”. The following table contains examples of file name prefixes for different types of IVI drivers.

An IVI.NET driver or wrapper is always packaged in a separate DLL from an IVI-COM or IVI-C driver or wrapper. The reason is that .NET uses managed C++ to call C functions. If they were packaged together, C and COM drivers would be required to load the .NET CLR.

Table 5-13. Example File Name Prefixes for Different Types of IVI Drivers

File Name Prefix	Type of Interface in Driver
ag33120a	IVI-C driver (uses C prefix)
age1463a	IVI-C driver packaged with COM wrapper (uses C prefix)
Agilent34401A	IVI-COM driver (uses COM component identifier)
ag34420	IVI-COM driver packaged with C wrapper (uses C prefix)
Agilent.Ag34401A.Fx20	IVI.NET driver
ag34401aCWrapper	C wrapper packaged separately (uses C prefix)
ag33120aCOMWrapper	COM wrapper packaged separately (uses COM component identifier)
Agilent.Ag34401A.Fx20.Wrapper	IVI.NET wrapper packaged separately.
Agilent.Agilent34401.Interop	IVI.NET PIA for IVI-COM driver.

5.19 File Versioning

IVI driver DLLs shall contain a Windows version resource with the three entries: CompanyName, ProductName, and FileVersion.

CompanyName is a string that contains the value of the Specific Driver Vendor, Class Driver Vendor, or Component Vendor attribute that the IVI driver returns.

ProductName is a string that includes the value of the Specific Driver Prefix, Class Driver Prefix, or Component Identifier attribute that the IVI driver returns.

FileVersion is a string in the following format:

```
MajorVersion.MinorVersion.BuildVersion[.InternalVersion]
```

MajorVersion, MinorVersion, BuildVersion, and InternalVersion are decimal numbers greater than or equal to zero and less than or equal to 65535. The MajorVersion shall not be zero, except for a pre-release version of the initial release of a driver. Each number may contain leading zeros but may not exceed 5 digits. The number fields are separated by periods, with no embedded spaces. The maximum valid FileVersion is 65535.65535.65535.65535.

A FileVersion string shall contain at least a MajorVersion, MinorVersion, and BuildVersion. A driver shall be released with an incremented MajorVersion or MinorVersion number when any of the following conditions are true:

- The new version of the driver contains changes in its API syntax.
- The new version of the driver contains significant changes to its semantics.

A driver shall be released with at least an incremented BuildVersion number when any of the following conditions are true:

- A new bitness of the driver is provided.
- The driver is modified in a way that does not require a MajorVersion or MinorVersion update.

If a supplier provides both a 32-bit and a 64-bit version of a driver, the 32-bit and 64-bit DLLs shall have the same MajorVersion, MinorVersion, and BuildVersion. Notice that when a 64-bit version of the driver is initially made available, the 32-bit version must be updated at the same time.

For the purpose of determining the version of an IVI driver, only the MajorVersion, MinorVersion, and BuildVersion are used; the InternalVersion is not used. The InternalVersion is optional and reserved for use by driver suppliers.

When comparing two FileVersion values, integer comparisons are performed successively for each number, starting at the leftmost number. If the first two numbers are equal, the next two numbers are compared, and so on.

Examples:

The following are examples of valid FileVersion values:

1.0.0
2.00.00
01.02.03
11.22.33
5.0.1
3.14.103
4.0.1000.0
0001.1.1.00005

The following are examples of comparisons of FileVersion values for the purpose of comparing driver versions:

5.1.345.213 is greater than 4.3553.3244.234
5.1.345.213 is less than 5.1.346.213
5.2.13.26 is equal to 5.2.13.56
5.001.100.001 is equal to 5.1.00100.1
2.15.32. 1 is greater than 2.15.18. 1
1.1.1 is equal to 1.1.1.0

5.20 Installation Requirements

All IVI drivers shall be made available to users in a Microsoft Windows installation program. The installation program shall install all the required files and documentation for the driver. For detailed requirements, refer to *IVI-3.17: Installation Requirements Specification*.

IVI driver suppliers may also distribute installation programs for other operating systems.

5.21 Driver Introduction Documentation

Driver introduction documentation shall be a separate file named ___Introduction to <*prefix*>___.<*xxx*> and shall be installed with the driver in the driver directory. Driver introduction documentation may consist mostly of links to other documentation and may contain additional information beyond what is specified in this section.

Driver introduction documentation shall contain the following sections, each containing the specified information and using the section names as shown.

Driver Documentation

There shall be a section called Driver Documentation. It shall list the location, file names, and purpose of the various documentation files supplied with the driver, including the compliance document (if separate).

Driver Source Code and Examples

For drivers that supply source code, there shall be a section called Driver Source Code and Examples. It shall list the location(s) of the source code and examples for the driver, but not each specific source and example file. If a specific directory exists dedicated to examples and/or samples, it is sufficient to reference that directory and not all the subdirectories therein. The section shall include a reference to the location in the documentation that explains how to build the source.

For drivers that do not supply source code there shall be a section called Examples. It shall list the location(s) of the examples for the driver, but not each specific example file. If a specific directory exists dedicated to examples and/or samples, it is sufficient to reference that directory and not all the subdirectories therein.

Connecting to the Instrument

There shall be a section called Connecting to the Instrument. It shall explain, or reference documentation that explains how to get started with the driver (including opening a driver session) in different development environments.

Configuring Instrument Settings

There shall be a section called Configuring Instrument Settings. It shall explain how to use the driver to configure instrument settings, using both attributes/properties and high-level configuration functions. It shall explain, or reference documentation that explains, how to use attributes/properties in various development environments.

Configuring Driver Settings

There shall be a section called Configuring Driver Settings. It shall explain that the driver has capabilities, such as simulation, that the user can configure either programmatically using attributes/properties or statically using a configuration utility.

Direct I/O

For drivers that provide Direct I/O functions, there shall be section called Direct I/O. It shall state the names of the functions which the user can use to communicate directly with the instrument.

Instrument Command Coverage

For drivers for message-based instruments, there shall be a section called Instrument Command Coverage. It shall provide a reference to the location in the documentation that lists the instrument commands that the driver implements and characterizes the instrument commands the driver does not implement.

Known Issues

There may be a section called Known Issues that lists the bugs and limitations that were known at the time the driver was released. If a driver supplier has a web page that lists bugs and limitations found after the driver was released, this section should provide a link to it.

Contacting Support

There shall be a section called Contacting Support. It shall specify how users can contact the driver supplier to report problems or ask questions relating to the driver.

Driver suppliers are encouraged to include links within each section that provide more detailed information.

The set of development environments that the driver supplier references in the document is at the discretion of the driver supplier.

5.21.1 Example Driver Introduction Documentation Files

This section contains example driver introduction documents for three different types of drivers: an IVI-C specific driver, an IVI-COM specific driver, and an IVI.NET specific driver.
The example driver introduction documents refer to trademarked product names. It is up to the driver supplier to determine how they will cite trademarks. No trademark citations are included in these examples.

Sample Driver Introduction Document for an IVI-C specific driver

Driver Documentation

The <*prefix*>.txt file can be found in the <*Program Files*>\IVI Foundation\IVI\Drivers\<*prefix*> directory. It contains notes about the driver's level of compliance with the IVI specifications and lists the optional IVI features supported in the driver.
The <*prefix*>.chm file can be found in the <*Program Files*>\IVI Foundation\IVI\Drivers\<*prefix*> directory. It contains descriptions of all functions, parameters and their valid values or ranges.

The <*prefix*>AttributeInfo.html file can be found in the <*Program Files*>\IVI Foundation\IVI\Drivers\<*prefix*> directory. It contains names and descriptions of all attributes and their valid values.

Driver Source Code and Examples

The C driver source code and example(s) can be found in the <*Program Files*>\IVI Foundation\IVI\Drivers\<*prefix*> directory. For instructions on rebuilding a driver, refer to the following Knowledgebase documents: Rebuilding an IVI Specific Driver in NI LabWindows/CVI and Rebuilding an IVI Specific Driver in Microsoft Visual Studio.

The NI LabVIEW wrapper VIs and examples can be found in the <*LabVIEW*>\instr.lib\<*prefix*> directory.

Examples of using MATLAB® and Instrument Control Toolbox with IVI-C drivers can be found in the online documentation.

Connecting to the Instrument

The IVI resources page (http://ivifoundation.org/resources/default.aspx) has documents and videos that explain how to get started with an IVI-C driver in different development environments:

IVI Getting Started Guide for LabVIEW
IVI Getting Started Guide for LabWindows/CVI
IVI Getting Started Guide for Microsoft Visual C++
IVI Getting started Guide for MATLAB

Configuring Instrument Settings

An IVI instrument driver implements each readable or writable setting on the instrument, such as the vertical voltage range on an oscilloscope, as an attribute.

IVI instrument drivers export high-level functions that allow you to set the value of multiple attributes in one call. This can be useful when it is necessary to send settings to an instrument in a particular order.

IVI instrument drivers also allow you to modify and get a value of individual attributes. You do this by calling <*Prefix*>_GetAttributeVi<*data type*> and <*Prefix*>_SetAttributeVi<*data type*> functions. To use these functions correctly you need to know the date type, valid values, and ID of the attribute you want to access. How you do so depends on the development environment you are using.

NI LabWindows/CVI Users

Open the <*prefix*>.fp file, expand the Configuration class, and select one of the Set/GetAttribute functions. From the function panel window, click on the **Attribute ID** control to display a dialog box containing a hierarchical list of the available attributes.

If you do not see the attribute you want, click on the **All Data Types** option in the Data Type pane of the dialog box.

NI LabVIEW Users

Use a Property Node to access the specific properties of a driver. A property node can be found in the Connectivity>>ActiveX functions palette.

Microsoft Visual Studio Users

Refer to the <*prefix*>AttributeInfo.html file in the <*Program Files*>\IVI Foundation\IVI\Drivers\<*prefix*> directory.

MATLAB Users

Refer here for examples and more information on using IVI-C drivers with MATLAB and Instrument Control Toolbox.

Configuring Driver Settings

IVI instrument drivers implement inherent capabilities such as simulation, range checking, state caching, coercion recording, interchangeability checking, and instrument status checking. A user can enable/disable these features either programmatically using an attribute or statically using a configuration utility.

Direct I/O

IVI instrument drivers for message-based instruments export <*Prefix*>_viRead and <*Prefix*>_viWrite functions, which enable you to perform direct I/O with the instrument**.**

Instrument Command Coverage

IVI instrument drivers for message-based instruments typically implement the full functionality of the instrument available via the commands and queries with a few exceptions. The <*prefix*>.chm file lists the instrument commands that the driver implements for each function and parameter. The same information can also be found in the <*prefix*>.fp file.
Some commands and queries are not suitable for an instrument driver. The commands from the following nodes are NOT implemented in this driver:

- *<DIAGnostic>*
- *<FORMat (may be used internally but not exposed to users)>*
- *<SYSTem:COMMunicate>*
- *< Service or Factory Calibration functionality>*
- *<Undocumented SCPI (factory use only)>*
- *<Other features not normally accessed through the programmatic interface, for example:*
  - *DISPlay*
  - *HARDCopy*
  - *MEMory:STATe*
  - *CURSor>*

Driver users can send any commands to a message-based instrument using the driver's Direct I/O functions.

Known Issues

<None>

Contact Support

If you have feedback or need help using this driver, contact < *appropriate support contact information* >.

Trademarks

<Optionally add trademark statements here>

Sample Driver Introduction Document for an IVI-COM specific driver

Driver Documentation

The readme.txt file can be found in the <*Program Files*>\IVI Foundation\IVI\Drivers\<*ComponentID*> directory. It contains notes about installation, known issues, and the driver's revision history.

The <*ComponentID*>.chm file can be found in the <*Program Files*>\IVI Foundation\IVI\Drivers\<*ComponentID*> directory. It contains

- A getting started example program
- General information about using the driver
- Reference information for all methods and properties in the IVI-COM driver
- Reference information for all functions and attributes in the IVI-C wrapper
- Information about using the driver in a variety of development environments including Visual Studio, LabVIEW, and MATLAB
- IVI compliance information

In addition to the .chm file, the driver may install integrated help for Visual Studio 2005/2008 (this may increase the install time by several minutes).

Driver Source Code and Examples

The IVI-COM driver source code can be found in the <*Program Files*>\IVI Foundation\IVI\Drivers\<*ComponentID*>*\Source* directory. For instructions on rebuilding the driver, refer to the "Building the Driver Source Code" help topic.
Driver example(s) can be found in the <*Program Files*>\IVI Foundation\IVI\Drivers\<*prefix*>\Examples directory. For instructions on rebuilding a driver, refer to the "Driver Examples" help topic.

Connecting to the Instrument

The driver help topic "Driver Examples" documents program examples for Visual Studio, LabVIEW, and MATLAB. Each of these examples illustrates how to connect to an instrument in the respective development environment.
The IVI resources page (http://ivifoundation.org/resources/default.aspx) has documents and videos that explain how to get started with an IVI-COM driver in different development environments:

IVI Getting Started Guide for LabVIEW
IVI Getting Started Guide for LabWindows/CVI
IVI Getting Started Guide for Microsoft Visual C++
IVI Getting started Guide for MATLAB
IVI Getting Started Guide for Agilent VEE

Configuring Instrument Settings

The <*ComponentID*> instrument driver application programming interface (API) includes methods and properties for setting instrument state variables, as well as methods for controlling the instrument and reading results from the instrument. These are documented in the <*ComponentID*> *IVI-COM Driver* > *Reference* help topic.

There are two driver API hierarchies that client programs may use to control the instrument. The first is the instrument specific hierarchy. This hierarchy can be used to access all of the functionality of the instrument. This hierarchy is available to programs that use one of the driver constructors to instantiate the driver. For more information on using the driver constructors, refer to the information on *Direct Driver Instantiation* in the *<ComponentID> IVI-COM Driver > Initializing the IVI-COM Driver* help topic. For more information on using the instrument specific hierarchy, refer to the *<ComponentID> IVI-COM Driver > Reference > I<ComponentID>* help topic.

The second hierarchy is the class compliant hierarchy. This hierarchy can be used to access the IVI class API for the *<ClassName>* class. This hierarchy is available to programs that use the IVI-COM class factory to instantiate the driver. For more information on using the class factory, refer to the information on *COM Session Factory* in the *<ComponentID> IVI-COM Driver > Initializing the IVI-COM Driver* help topic. For more information on using the class compliant hierarchy, refer to the *<ComponentID> IVI-COM Driver > Reference > IIvi<ClassName>* help topic.

Configuring Driver Settings

IVI instrument drivers implement inherent capabilities including properties that control driver behavior, utility methods, and identifying information. For more information on using the inherent capabilities, refer to the *<ComponentID> IVI-COM Driver > Reference > IIviDriver* help topic.

Properties that control driver behavior such as simulation, range checking, and instrument status checking can be enabled/disabled when initializing the driver or by using configuration information in the IVI Configuration Store. For more information, refer to the *Getting Started > Configuring the Driver* help topic.

Direct I/O

IVI instrument drivers for message-based instruments include *Read*, *ReadBytes*, *Write*, and *WriteBytes* methods, as well as a reference to the underlying I/O, *DirectIO*, which enable you to perform I/O directly with the instrument. For more information on using these methods and properties, refer to the *<ComponentID> IVI-COM Driver > Reference > I<ComponentID> > System* help topic.

Instrument Command Coverage

IVI instrument drivers for message-based instruments typically implement the full functionality of the instrument available via the commands and queries with a few exceptions. The help reference topics for each method and property list the instrument command(s) that the driver implements for each function and parameter.

Some commands and queries are not suitable for an instrument driver. The following commands are NOT implemented in this driver:

- All commands in the *<DIAGnostic>* subsystem
- All commands in the *<CALibrate>* subsystem
- All commands in the *<FORMat>* subsystem
- All commands in the *<DISPlay>* subsystem
- All commands in the *<SYSTem:COMMunicate>*subsystem
- Undocumented SCPI commands
- Specific commands
    - *HARDCopy*
    - *MEMory:STATe*
    - *CURSor>*

Driver users can send any commands to a message-based instrument using the driver's Direct I/O functions.

Known Issues

The readme.txt file can be found in the <*Program Files*>\IVI Foundation\IVI\Drivers\<*ComponentID*> directory. It contains information about known issues.

Contact Support

If you have feedback or need help using this driver, contact <*appropriate support contact information*>.

Trademarks

<Optionally add trademark statements here>

Sample Driver Introduction Document for an IVI.NET specific driver

Driver Documentation

The readme.txt file can be found in the <*Program Files*>\IVI Foundation\IVI\Microsoft.NET\Framework64\v2.0.50727\<*DriverDirName*> directory. It contains notes about installation, known issues, and the driver's revision history.
The <*DriverName*>.chm file can be found in the <*Program Files*>\IVI Foundation\IVI\Microsoft.NET\Framework64\v2.0.50727\<*DriverDirName*> directory. It contains

- A getting started example program
- General information about using the driver
- Reference information for all methods and properties in the IVI.NET driver
- Information about using the driver in a variety of development environments including Visual Studio, LabVIEW, and MATLAB
- IVI compliance information

In addition to the .chm file, the driver may install integrated help for Visual Studio 2005/2008 (this may increase the install time by several minutes), and may separately install integrated help for Visual Studio 2010.

Driver Source Code and Examples

The IVI.NET driver source code can be found in the <*Program Files*>\IVI Foundation\IVI\Microsoft.NET\Framework64\v2.0.50727\<*DriverDirName*>*\Source* directory. For instructions on rebuilding the driver, refer to the "Building the Driver Source Code" help topic.

Driver example(s) can be found in the <*Program Files*>\IVI Foundation\IVI\Microsoft.NET\Framework64\v2.0.50727\<*DriverDirName*>\Examples directory. For instructions on rebuilding a driver, refer to the "Driver Examples" help topic.

Connecting to the Instrument

The driver help topic "Driver Examples" documents program examples for Visual Studio, LabVIEW, and MATLAB. Each of these examples illustrates how to connect to an instrument in the respective development environment.

Configuring Instrument Settings

The <*DriverDirName*> instrument driver application programming interface (API) includes methods and properties for setting instrument state variables, as well as methods for controlling the instrument and reading results from the instrument. These are documented in the *Driver API Reference* help topic.

There are two driver API hierarchies that client programs may use to control the instrument. The first is the instrument specific hierarchy. This hierarchy can be used to access all of the functionality of the instrument. This hierarchy is available to programs that use one of the driver constructors to instantiate the driver. For more information on using the driver constructors, refer to the information on *Using the Driver's Constructors* in the *Using the Driver Effectively > The .NET API > Instantiating the Driver* help topic. For more information on using the instrument specific hierarchy, refer to the *Driver API Reference > Driver Hierarchy > I<ComponentID>* help topic.

The second hierarchy is the class compliant hierarchy. This hierarchy can be used to access the IVI class API for the *<ClassName>* class. This hierarchy is available to programs that use an IVI.NET Create(…) class factory method to instantiate the driver. For more information on using Create(…), refer to the information on *Using the IVI Factory Methods* in the *Using the Driver Effectively > The .NET API > Instantiating the Driver* help topic. For more information on using the class compliant hierarchy, refer to the *Driver API Reference > Driver Hierarchy > IIvi<ClassName>* help topic.

Configuring Driver Settings

IVI instrument drivers implement inherent capabilities including properties that control driver behavior, utility methods, and identifying information. For more information on using the inherent capabilities, refer to the *Driver API Reference > Driver Hierarchy > IIviDriver* help topic.

Properties that control driver behavior such as simulation, range checking, and instrument status checking can be enabled/disabled when initializing the driver or by using configuration information in the IVI Configuration Store. For more information, refer to the *Using the Driver Effectively > The .NET API > Instantiating the Driver* help topic.

Direct I/O

IVI instrument drivers for message-based instruments include *Read*, *ReadBytes*, *Write*, and *WriteBytes* methods, as well as a reference to the underlying I/O, *DirectIO*, which enable you to perform I/O directly with the instrument. For more information on using these methods and properties, refer to the *Driver API Reference > Driver Hierarchy > I<ComponentID> > System* help topic.

Instrument Command Coverage

IVI instrument drivers for message-based instruments typically implement the full functionality of the instrument available via the commands and queries with a few exceptions. The help reference topics for each method and property list the instrument command(s) that the driver implements for each function and parameter.

Some commands and queries are not suitable for an instrument driver. The following commands are NOT implemented in this driver:

- All commands in the *<DIAGnostic>* subsystem
- All commands in the *<CALibrate>* subsystem
- All commands in the *<FORMat>* subsystem
- All commands in the *<DISPlay>* subsystem
- All commands in the *<SYSTem:COMMunicate>*subsystem
- Undocumented SCPI commands
- Specific commands:
    - *HARDCopy*
    - *MEMory:STATe*
    - *CURSor>*

Driver users can send any commands to a message-based instrument using the driver's Direct I/O functions.

Known Issues

The readme.txt file can be found in the <*Program Files*>\IVI Foundation\IVI\Microsoft.NET\Framework64\v2.0.50727\<*DriverDirName*> directory. It contains information about known issues.

Contact Support

If you have feedback or need help using this driver, contact <*appropriate support contact information*>.

Trademarks

<Optionally add trademark statements here>

5.22 Help Documentation

Help documentation shall be installed with the driver. Help documentation may contain additional information beyond what is specified in this section.

For each IVI-C driver function and IVI-COM or IVI.NET driver method, an IVI driver shall provide help documentation for the following:

- The function prototype
- A description of the function usage
- For each parameter, a description of its usage and valid values
- Return value and status codes
- For instruments that have an ASCII command set such as SCPI, the commands used in the function or method

For each IVI-C driver attribute and IVI-COM or IVI.NET driver property, an IVI driver shall provide help documentation for the following:

- A description of the attribute usage
- The data type
- Read/write access
- Valid values
- For IVI-C and IVI-COM, return value and status codes, and for IVI.NET, exceptions
- For instruments that have an ASCII command set such as SCPI, the commands used to get or set the attribute

Common status codes or exceptions for functions and attributes may be presented in a standard location instead of documented for each function and attribute.

The help documentation for IVI-C drivers may present status codes for attributes in the Set Attribute and Get Attribute functions.

Each IVI driver shall provide help documentation on known ADE restrictions, such as minimum versions or feature requirements of ADEs.

The driver documentation for the identifier parameters in class compliant interfaces shall explain that users who want to achieve interchangeability should use virtual identifiers and that virtual identifiers should be sufficiently specific to the test system such that they are unlikely to conflict with physical identifiers.

5.22.1 Copyright Notice

Each IVI driver shall include the following text in a visible location in the help documentation.

```
Content from the IVI specifications reproduced with permission from the IVI
Foundation.

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.
```

Documentation published before July 1, 2011, that reproduces material from the IVI specifications shall have until January 1, 2016, to add the required citation.

5.23 Compliance Documentation

Each IVI driver shall include documentation defining its level of compliance with the IVI specifications and identifying the optional IVI features supported in the driver. The compliance information shall be installed with the driver. It shall be prominently displayed in the Windows help file or other readable help document. Compliance documentation may contain additional information beyond what is specified in this section. If 32-bit and 64-bit versions of the same driver exist, the contents of the compliance documents for them shall be the same.

The compliance documentation shall contain the following sections, each containing the specified information and using the section and item names as shown.

Compliance Category Section

The Compliance Category section of the compliance document informs the user of the type of the driver and the API type that it exports. An IVI-COM or IVI.NET specific driver that complies with multiple class specifications shall contain a separate Compliance Category section for each class specification with which it complies. All IVI drivers shall include the following items in the Compliance Category section.

Compliance Category Name

The Compliance Category Name item shall specify the type of IVI driver as detailed in Section 2.2, *Types of IVI Drivers*. The name of the driver shall be formatted as follows.

```
IVI[-C, -COM, .NET, -COM/C, -COM/.NET, -C/.NET, -COM/C/.NET] [One of the
IVI classes, Custom][Class, Specific] Instrument Driver
```

Square brackets (`[]`) indicate a set of terms from which one shall be selected. Each part of the name is defined in the following paragraphs:

All IVI-C drivers shall use “IVI-C” in the compliance name. All IVI-COM drivers shall use “IVI-COM” in the compliance name. All IVI.NET drivers shall use “IVI.NET” in the compliance name. All IVI drivers that implement multiple API types shall use “IVI” followed by the implemented APIs, separated by the ‘/’ character, in the compliance name.

To specify an IVI class, all IVI class-compliant specific drivers and IVI class drivers shall use the name of the IVI class specification. All IVI custom specific drivers shall use “Custom” in the compliance name.

All IVI class drivers shall use “Class” in the compliance name. All IVI specific drivers shall use “Specific” in the compliance name.

The following are examples of compliance names.

```
IVI-C IviDmm Class Instrument Driver
IVI-C IviDmm Specific Instrument Driver
IVI-COM IviScope Specific Instrument Driver
IVI.NET IviScope Specific Instrument Driver
IVI-COM/C Custom Specific Instrument Driver
IVI-COM Custom Specific Instrument Driver
```

Class Specification Version

The Class Specification Version item specifies the version of the IVI class specification in accordance with which the driver was developed or updated. This item is not present for IVI custom drivers.

IVI Generation

The IVI Generation represents a list of specification versions that, taken together, constitute a citable and distinguishable set.

The IVI Generation item specifies a set of minimum IVI specification versions with which a specific driver must comply in order to claim compliance with a specific IVI Generation. The IVI Foundation web site contains a list of the IVI Generations and the minimum specification versions required to claim compliance with each.

The IVI Generation is designated in the form of "IVI-<*year*>", where <*year*> is the year following the year in which the specification changes that constitute the new generation were approved. The following are examples of existing IVI Generations: IVI-2003, IVI-2014. This item is not required for IVI-2003 drivers.

Class Capability Groups

For IVI specific drivers, the Class Capability Groups item shall contain a tabular list of all the Class Group Capabilities. If a class-compliant specific driver implements a capability group for one or more instrument models that the driver supports, the documentation shall indicate support for that capability group. If a class-compliant driver does not implement a capability group in the driver, the documentation shall indicate the driver does not support that capability group. This item is not present for IVI custom specific drivers or IVI class drivers.

Certification Statement

All IVI drivers shall include a statement certifying that they comply with all applicable requirements of the IVI specifications at the time this compliance document was submitted prepared. The following text shall be used:

```
<provider's name> has evaluated and tested this driver to verify
that it meets all applicable requirements of the IVI specifications
at the time this compliance document was submitted to the IVI
Foundation and agrees to abide by the dispute arbitration provisions
in Section 8 of IVI-1.2: Operating Procedures, if the IVI Foundation
finds this driver to be non-conformant.
```

Optional Features Section

For IVI specific drivers, the Optional Features section of the compliance document informs the user of any optional IVI features included in the IVI driver. IVI Class drivers do not have an Optional Features section. IVI specific drivers shall include the following items in the Optional Features section.

Interchangeability Checking

The Interchangeability Checking item shall specify whether the specific driver supports interchangeability checking. If a specific driver implements minimal or full interchangeability checking, as described in Section 3.3.6, *Interchangeability Checking*, the documentation shall specify that the driver supports interchangeability checking. Otherwise, the documentation shall specify that the driver does not support interchangeability checking. This item is not present for IVI custom specific drivers.

State Caching

The State Caching item shall specify whether the driver supports state caching. If a specific driver implements state caching for one or more attribute, as described in Section 5.10.6, *State Caching*, the documentation shall specify that the driver supports state caching. Otherwise, the documentation shall specify that the driver does not support state caching.

Coercion Recording

The Coercion Recording item shall specify whether the driver supports coercion recording. If a specific driver implements coercion recording, as described in Section 5.10.1.7, *Coercion Recording*, the documentation shall specify that the driver supports coercion recording. Otherwise, the documentation shall specify that the driver does not support coercion recording.

Driver Identification Section

The Driver Identification section informs the user of the identity of the driver. All IVI drivers shall include the following items in this section.

Driver Revision

The Driver Revision item shall contain the value of the Specific Driver Revision, Class Driver Revision, or Component Revision attribute that the IVI driver returns.

Driver Vendor

The Driver Vendor item shall contain the value of the Specific Driver Vendor, Class Driver Vendor, or Component Vendor attribute that the IVI driver returns.

Description

The Description item shall contain value of the Specific Driver Description, Class Driver Description, or Component Description attribute that the IVI driver returns.

Prefix/Component Identifier

The Prefix/Component Identifier item shall contain the value of the Specific Driver Prefix, Class Driver Prefix, or the Component Identifier attribute that the IVI driver returns.

Hardware Information Section

The Hardware Information section informs the user about the hardware supported by the instrument driver. All IVI specific drivers shall include the following items in this section:

Instrument Manufacturer

The Instrument Manufacturer item shall contain the name(s) of the instrument manufacturer.

Supported Instrument Models

The Supported Instrument Models item shall contain the value of the Supported Instrument Models attribute that the IVI driver returns.

Supported Bus Interfaces

The Supported Bus Interfaces item shall contain an itemized list of the bus interfaces that the IVI driver supports.

<*nn*>-bit Software Information Section

The 32-bit Software Information section and 64-bit Software Information section inform the user about additional software required by the instrument driver. The 32-bit Software Information section shall include information relevant to the 32-bit IVI driver. The 64-bit Software Information section shall include information relevant to the 64-bit IVI driver. IVI drivers that support only 32-bit operating systems shall include a 32-bit Software Information section. IVI drivers that support 64-bit operating systems shall include both a 32-bit Software Information section and a 64-bit Software Information section.

For each software information section, IVI drivers shall include the following:

Supported Operating Systems

The Supported Operating Systems item shall contain a list of supported operating systems that the IVI driver was known to work on at the time of release.

Unsupported Operating Systems

The Unsupported Operating Systems item shall contain a list of IVI Foundation targeted operating systems on which the IVI driver is not supported at the time of release. If a driver does not support one or more of the following operating systems, this item shall list those operating systems.

- Windows 7 (32-bit)
- Windows 7 (64-bit)
- Windows 8 (32-bit)
- Windows 8 (64-bit)
- Windows 10 (32-bit)
- Windows 10 (64-bit)
- Windows 11

This item may be absent if there are no operating systems to list.

.NET Minimum Runtime Version

The minimum required .NET runtime version shall be specified for all .NET drivers, including wrappers.

.NET Target Framework Version

The specific version of the .NET Framework against which the driver was compiled shall be specified for all .NET drivers, including wrappers.

Drivers published after Jan.1 2019 must include this information. Drivers published before this date may omit the information.

Support Software Required

The Support Software Required item shall contain a list of the support libraries that the IVI driver requires but that are not provided by the IVI Foundation or the operating system. Restrictions, such as the minimum version number, should be included.

Source Code Availability

The Source Code Availability item shall specify if instrument driver source code is available to end-users and the conditions under which it is distributed. If the instrument driver is just a thin layer on top of support libraries, this item shall contain a statement indicating as such.

Unit Testing Section

The Testing section informs the user about the unit testing performed on the instrument driver. All IVI specific drivers shall include the following items in this section:

Test Setup(s)

The Test Setup(s) item shall specify a list of test setups on which you ran the complete set of unit tests, as specified in Section 5.2.2.1.1, *Unit Test Procedure*. If you performed the complete set of unit tests on a large number of setups, you can express multiple setups with one specification by listing multiple values for the various setup elements, as long as you test on all valid combinations implied by the multiple values listed.

Instrument Model and Firmware Revision

The Instrument Model and Firmware Revision item shall specify the instrument models and their firmware revisions on which you performed the complete set of unit tests. If you performed the complete set of unit tests using multiple instrument models and/or multiple firmware revisions, you may document that information as shown in the following example:

```
Instrument Model (Firmware Revision):   Agilent 34410A (2.1, 2.21),
                                        Agilent 33411A (2.39)
```

Bus Interface

The Bus Interface item shall specify the bus interface through which the instrument was connected to the computer when you performed the complete set of unit tests. If you performed the complete set of unit tests with multiple bus interfaces, you may document that information as shown in the following example:

```
Bus Interface:   GPIB, USB, LAN
```

Operating System and Service Pack

The Operating System and Service Pack item shall specify the operating system(s) and corresponding service pack(s) on which you performed the complete set of unit tests. If you performed the complete set of unit tests with multiple operating systems and/or corresponding service packs, you may document that information as shown in the following example:

```
Operating System (Service Pack): Windows 7 (no SP, SP1)
```

OS Bitness and Application Bitness

The OS Bitness and Application Bitness item shall specify the bitness of the operating system and the bitness of the application which you used to perform the complete set of unit tests. If you performed the complete set of unit tests on multiple bitnesses, you may document that information as shown in the following example:

```
OS Bitness/Application Bitness:  32-bit/32-bit, 64-bit/32-bit, 64-bit/64-
bit
```

VISA Vendor and Version

The VISA Vendor and Version item shall specify the vendor and version of the VISA implementation you used to perform the complete set of unit tests if the driver requires VISA. If you performed the complete set of unit tests using multiple VISA implementations, you may document that information as shown in the following example:

```
VISA Vendor and Version:   NI-VISA 5.2, TekVISA 3.3.4
```

IVI Shared Components Version

The IVI Shared Components Version item shall specify the version of the IVI Shared components you used to perform the complete set of unit tests. If you performed the complete set of unit tests using multiple versions of the IVI Shared Components, you may document that information as shown in the following example:

```
IVI Shared Components Version:   2.0, 2.2.1
```

To clarify the implication of listing multiple values for multiple items, consider the following partial example:

```
Operating System (Service Pack):         Windows 8, Windows 7 (SP1)
OS Bitness/Application Bitness:          32-bit/32-bit, 64-bit/32-bit
```

This indicates that you performed the complete set of unit tests using the following three combinations:

- Windows 8 32-bit, with a 32-bit application
- Windows 7 (SP1) 64-bit, with a 32-bit application
- Windows 7 (SP1) 32-bit, with a 32-bit application

since IVI does not support 64-bit applications.

Driver Installation Testing Section

The Driver Installation Testing section informs the user about the driver installation testing performed on the instrument driver. All IVI specific drivers shall include the following items in this section:

Operating Systems and Service Packs

The Operating Systems and Service Packs item shall specify a list of operating systems and service packs on which you performed the driver installation testing.

OS Bitness

The OS Bitness item shall specify the bitness(es) of the operating systems on which you performed the driver installation testing.

Driver Buildability Section

The Driver Buildability section informs the user about the driver buildability testing performed on the instrument driver. All IVI specific drivers that include source code shall include the following items in this section:

Operating Systems and Service Packs

The Operating Systems and Service Packs item shall specify a list of operating systems and service packs on which you performed the driver buildability testing.

OS Bitness

The OS Bitness item shall specify the bitness(es) of the operating systems on which you performed the driver buildability testing.

Driver Test Failures Section

The Driver Test Failures section informs the user about the failures that the driver testing revealed on the instrument driver that were not fixed before release. All IVI specific drivers shall include the following items in this section:

Known Issues

The Known Issues item shall specify a list of known issues that reflect all test failures that were not fixed. If there are no known issues, indicate None.

Additional Compliance Information Section

The Additional Compliance Information section informs the user about additional information that relates to compliance but is not included in other defined items. For example, an IVI custom driver may include notification of another IVI class-compliant driver available for the same hardware. This section is not present if no additional compliance information exists.

5.23.1 Example Compliance Text Files

This section contains example compliance documents for three different types of drivers: an IVI-COM class-compliant specific driver, an IVI-C custom specific driver, and an IVI-C class driver. IVI.NET compliance documents are similar to IVI-COM, with the addition of the .NET Minimum Runtime Version.

Sample Compliance Document for an IVI-COM class-compliant specific driver

```
IVI Compliance Category:
IVI-COM IviFgen Specific Instrument Driver
Class Specification Version:  4.00
IVI Generation: IVI-2014
Group Capabilities Supported:


Base             =  Supported
StdFunc          =  Supported
ModulateAM       =  Supported
ModulateFM       =  Supported
ArbWfm           =  Supported
ArbFrequency     =  Supported
ArbSeq           =  Not Supported
Trigger          =  Supported
InternalTrigger  =  Supported
SoftwareTrigger  =  Supported
Burst            =  Supported


Optional Features:
Interchangeability Checking = True
State Caching = False
Coercion Recording = True


Driver Identification:
Driver Revision:              2.1.0
Driver Vendor:                VTI Instruments
Component Identifier:         ag33220a
Description:                  Function/Arbitrary Waveform Generator


Hardware Information:
Instrument Manufacturer:      Agilent Technologies
Supported Instrument Models:  33220A
Supported Bus Interfaces:     USB, LAN, GPIB
```

32-bit Software Information:

```
Supported Operating Systems:  Windows 7 (32-bit), Windows 7 (64-bit), Windows 8
(32-bit), Windows 8 (64-bit), Windows 10 (32-bit), Windows 10 (64-bit), Windows
11
Unsupported Operating Systems: N/A
Support Software Required:    VISA-COM
Source Code Availability:     Source code included with driver.
```

64-bit Software Information:

```
Supported Operating Systems:  Windows 7 (64-bit), Windows 8 (64-bit), Windows 10
(64-bit), Windows 11
Unsupported Operating Systems: N/A
Support Software Required:    VISA-COM
Source Code Availability:     Source code included with driver.
```

Unit Testing:

```
Test Setup 1:

Instrument Model (Firmware Revision):   33220A (2.07)
Bus Interface:                          GPIB, USB, LAN
Operating System (Service Pack):        Windows 7 (SP1)
OS Bitness/Application Bitness:         32-bit/32-bit, 64-bit/32-bit
VISA Vendor and Version:                Agilent VISA-COM (IO Libraries 16.3)
IVI Shared Components Version:          2.2.1

Test Setup 2:

Instrument Model (Firmware Revision):   33220A (1.08)
Bus Interface:                          GPIB
Operating System (Service Pack):        Windows 8
OS Bitness/Application Bitness:         32-bit/32-bit
VISA Vendor and Version:                Agilent VISA-COM (IO Libraries 16.3)
IVI Shared Components Version:          2.2.1
```

Driver Installation Testing:

```
Operating System (Service Pack):        Windows 8, Windows 7 (SP1)
OS Bitness:                             32-bit, 64-bit
```

Driver Buildability:

```
Operating System (Service Pack):        Windows 8, Windows 7 (SP1)
OS Bitness:                             32-bit, 64-bit
```

Driver Test Failures:

```
Known Issues:                           None
```

Sample Compliance Document for an IVI.NET class-compliant specific driver

IVI Compliance Category:

```
IVI.NET IviFgen Specific Instrument Driver
Class Specification Version:  4.00
IVI Generation: IVI-2014
Group Capabilities Supported:
```

```
Base             =  Supported
StdFunc          =  Supported
ModulateAM       =  Supported
ModulateFM       =  Supported
ArbWfm           =  Supported
ArbFrequency     =  Supported
ArbSeq           =  Not Supported
Trigger          =  Supported
InternalTrigger  =  Supported
SoftwareTrigger  =  Supported
Burst            =  Supported
```

Optional Features:

```
Interchangeability Checking = True
State Caching = False
Coercion Recording = True
```

Driver Identification:

```
Driver Revision:             2.1.0
Driver Vendor:               VTI Instruments
Component Identifier:        Agilent33220a
Description:                 Function/Arbitrary Waveform Generator
```

Hardware Information:

```
Instrument Manufacturer:     Agilent Technologies
Supported Instrument Models:  33220A
Supported Bus Interfaces:    USB, LAN, GPIB
```

32-bit Software Information:

```
Supported Operating Systems:  Windows 7 (32-bit), Windows 7 (64-bit), Windows 8
(32-bit), Windows 8 (64-bit), Windows 10 (32-bit), Windows 10 (64-bit), Windows
11
Unsupported Operating Systems: N/A
.NET Minimum Runtime Version:  2.0.50727.4927
Source Code Availability:     Source code included with driver.
```

64-bit Software Information:

```
Supported Operating Systems:  Windows 7 (64-bit), Windows 8 (64-bit), Windows 10
(64-bit), Windows 11
Unsupported Operating Systems: N/A
.NET Minimum Runtime Version:  2.0.50727.4927
Source Code Availability:     Source code included with driver.
```

Unit Testing:

```
Test Setup:

Instrument Model (Firmware Revision):   33220A (2.07)
Bus Interface:                          GPIB, USB, LAN
Operating System (Service Pack):        Windows 7 (no SP, SP1)
OS Bitness/Application Bitness:         32-bit/32-bit, 64-bit/64-bit
VISA Vendor and Version:                Agilent VISA (IO Libraries 16.3)
IVI Shared Components Version:          2.2.1
```

Driver Installation Testing:

```
Operating System (Service Pack):        Windows 7 (SP1)
OS Bitness:                             32-bit, 64-bit
```

Driver Buildability:

```
Operating System (Service Pack):        Windows 7 (SP1)
OS Bitness:                             32-bit, 64-bit
```

Driver Test Failures:

```
Known Issues:                           None
```

Sample Compliance Document for an IVI-C custom specific driver

IVI Compliance Category:
```
IVI-C Custom Specific Instrument Driver
```

Optional Features:
```
Interchangeability Checking = False
State Caching = True
Coercion Recording = True
```

Driver Identification:
```
Driver Revision:             1.1.0
Driver Vendor:               National Instruments
Prefix:                      AG81100
Description:                 Pulse/Pattern Generator
```

Hardware Information:
```
Instrument Manufacturer:      Agilent Technologies
Supported Instrument Models:  81101A, 81104A, 81110A, 81130A
Supported Bus Interfaces:     GPIB
```

32-bit Software Information:
```
Supported Operating Systems:  Windows 7 (32-bit), Windows 7 (64-bit), Windows 8
(32-bit), Windows 8 (64-bit), Windows 10 (32-bit), Windows 10 (64-bit), Windows
11
Unsupported Operating Systems: N/A
Support Software Required:    NI-VISA ver 5.0 or later
                              NI IVI Compliance Package ver 4.2 or later
Source Code Availability:     Source code available.
```

64-bit Software Information:
```
Supported Operating Systems:  Windows 7 (64-bit), Windows 8 (64-bit), Windows 10
(64-bit), Windows 11
Unsupported Operating Systems: N/A
Support Software Required:    NI-VISA ver 5.0 or later
                              NI IVI Compliance Package ver 4.2 or later
Source Code Availability:     Source code available.
```

Unit Testing:

```
Test Setup 1:

Instrument Model (Firmware Revision):   81104A (1.04), 81130A (1.12)
Bus Interface:                          GPIB
Operating System (Service Pack):        Windows 8, Windows 7 (SP1)
OS Bitness/Application Bitness:         32-bit/32-bit, 64-bit/32-bit
VISA Vendor and Version:                NI-VISA 5.2
```

```
IVI Shared Components Version:          2.2.1
```

Driver Installation Testing:

```
Operating System (Service Pack):        Windows 8, Windows 7 (SP1)
OS Bitness:                             32-bit, 64-bit
```

Driver Buildability:

```
Operating System (Service Pack):        Windows 7 (SP1)
OS Bitness:                             32-bit, 64-bit
```

Driver Test Failures:

```
Known Issues:                           None
```

Sample Compliance Document for an IVI-C Class Driver

IVI Compliance Category:

```
IVI-C IviPwrMeter Class Instrument Driver
Class Specification version:  1.00
```

Driver Identification:

```
Driver Revision:              1.0.0
Driver Vendor:                National Instruments
Prefix:                       IviPwrMeter
Description:                  Class driver for IviPwrMeter
```

32-bit Software Information:

```
Supported Operating Systems:  Windows 7 (32-bit), Windows 7 (64-bit), Windows 8
(32-bit), Windows 8 (64-bit), Windows 10 (32-bit), Windows 10 (64-bit), Windows
11
Unsupported Operating Systems: N/A
Support Software Required:    NI IVI Compliance Package ver 4.2 or later
Source Code Availability:     Source code available under separate license.
```

64-bit Software Information:

```
Supported Operating Systems:  Windows 7 (64-bit), Windows 8 (64-bit), Windows 10
(64-bit), Windows 11
Unsupported Operating Systems: N/A
Support Software Required:    NI IVI Compliance Package ver 4.2 or later
Source Code Availability:     Source code available under separate license.
```

5.24 Compliance for Custom Drivers

It is possible to create an IVI custom specific driver for an instrument that fits within an instrument class. Driver suppliers may create such IVI custom specific drivers to meet the needs of a special market niche. The Additional Compliance Information section of the compliance document for these IVI custom specific drivers shall include a statement informing users how to obtain an IVI class-compliant specific driver for the instrument.

Appendix A – Example: Applying Virtual Identifier Mappings

This appendix presents an example of how an IVI specific driver applies virtual repeated capability identifier mappings in a repeated capability selector. The example is in the form of a procedure that an IVI specific driver might follow when the user passes a virtual repeated capability selector to a driver function. The procedure reflects material from Section 4.4, *Repeated Capability Selectors*, and the requirements in Section 5.9.2, *Applying Virtual Identifier Mappings*. In particular, the procedure uses the syntax terminology defined in Section 4.4.7, *Formal Syntax for Repeated Capability Selectors*.

The procedure assumes the following selector string and virtual identifier mappings:

```
"MyDisplay:MyWindow:[MyTrace1-MyTrace3,MyTraceList,tr16,tr17]"

MyDisplay = disp3
MyWindow = win1
MyTrace1 = tr0
MyTrace2 = tr1
MyTrace3 = tr2
MyTraceList = tr11-tr13
tr16 = tr15
```

1. If the selector is passed as a parameter to a function which operates on a multilevel hierarchy of repeated capabilities, the IVI specific driver parses the selector into path segments. Using the example selector string, an IVI specific driver that supports nested repeated capabilities parses the selector string into the following three path segments:

   ```
   MyDisplay
   MyWindow
   [MyTrace1-MyTrace3,MyTraceList,tr16,tr17]
   ```

   If the number of path segments exceeds the number of levels in the repeated capability hierarchy on which the function operates, the driver may report an error and exit the procedure.

   If the IVI specific driver does not use nested repeated capabilities or the selector is passed to a COM collection, the driver does not parse the selector into path segments. In that case, if the selector contains colon operators, the driver may report an error and exit the procedure.

2. If the driver allows multiple instances of the repeated capability to be accessed at once, the driver parses the selector or the appropriate path segments into list elements. In the example, a driver that allows multiple instances to be accessed at the third level of the repeated capability hierarchy parses the third path segment into the following four list elements.

   ```
   MyTrace1-MyTrace3
   MyTraceList
   tr16
   tr17
   ```

3. The driver parses ranges into repeated capability tokens. Any remaining unparsed items are considered to be repeated capability tokens. In the example, the following are repeated capability tokens:

   ```
   MyDisplay
   MyWindow
   MyTrace1
   MyTrace3
   MyTraceList
   tr16
   tr17
   ```

4. For each repeated capability token, the driver checks whether the token matches a virtual identifier the

user defined in the IVI configuration store. If so, the driver replaces the token with the string to which the user mapped the virtual identifier in the IVI configuration store.

In the example, all repeated capability tokens except `tr17` match virtual identifiers. After the mappings, the resultant selector string is the following:

```
"disp3:win1:[tr0-tr2,tr11-tr13,tr15,tr17]"
```

If a token matches a virtual identifier, the driver replaces the token with the mapped string regardless of whether the token is also a valid physical identifier for an instance of the repeated capability. Notice that `tr16` is both a valid physical identifier and a virtual identifier that maps to `tr15`. As required, `tr16` is mapped to `tr15`.

The driver performs the replacement operation only on the virtual identifiers that explicitly appear in the original selector string. For example, if `MyTrace1` were mapped to `MyTrace2`, the driver would not replace `MyTrace2` with its mapped string. The resultant selector string would be the following:

```
"disp3:win1:[MyTrace2-tr2,tr11-tr13,tr15,tr17]"
```

Ultimately, the driver would report an error on this selector string because `MyTrace2` is not a valid physical identifier.

5. The driver then parses the resultant selector into path segments, lists, ranges, and tokens again.
6. The driver verifies that each token is valid physical name for the repeated capability.
7. The driver verifies that lists and ranges are valid according to the syntax specified in Section 4.4.2, *Representing a Set of Instances*.
8. If all tokens, lists, and ranges are valid, the result is a valid physical repeated capability selector.
9. In this example, the resultant selector is the following:

```
"disp3:win1:[tr0-tr2,tr11-tr13,tr15,tr17]"
```

The selector is a valid physical selector that resolves to the following set of physical identifiers:

```
disp3:win1:tr0
disp3:win1:tr2
disp3:win1:tr11
disp3:win1:tr13
disp3:win1:tr15
disp3:win1:tr17
```

Appendix B – Example: IVI Conformance Tests

The following table documents a suggested checklist of tests and checks that driver vendors may find useful for assuring compliance with various IVI specifications. The list is not comprehensive: driver vendors should evaluate each development project to determine the best method for assuring compliance. The list also contains tests that test other driver characteristics areas such as ADE usability that are not covered by IVI specifications.

Test Step	Actions
Document Development and Test Setup(s)	Document: Hardware Used • Instrument Model(s) • Firmware Version • Options Installed • Accessories Used • Communication Bus Software Used, Including Version Information • IVI Shared Components • Support Components (VISA) • Developer Specific Support Libraries • ADE Operating System • Operating System(s), Service Pack(s), and Bitness(es) • CPU Description
Inherent Capabilities Checks	Check or Test: API Compliance • Inherent API Attributes Values Data Types Read/Write Access Functions Function Prototypes

Test Step	Actions
Class Compliant Capabilities Checks	Check or Test: API Compliance • Class Compliant API Attributes Values Data Types Read/Write Access Base Class Extension Cross dependencies in extensions Functions Prototypes Base Class Extension Cross dependencies in extensions Behavior • Attribute Coercion Direction • Parameters (When Ignored) • Disables Unused Extensions per IVI3.1, section 3.3.4 • Other Behavioral Models • Simulation
Architecture Checks	Check or Test: Simulation Applying User-Defined Settings Usage of Shared Components Status Checking – How specific do we need to be? Resource Locking Error Handling Interchange Checking Multithreading
Installer Tests	Check: Presence of Proper Files Common Components • Registry Entry • Configuration Server Entry Install Modes • Silent Install • Dialog Install • Standard Install • Custom Install • Uninstall

Test Step	Actions
Instrument Specific Capabilities Checks	Check or Test: API Compliance • Instrument Specific Compliant API Attributes Values Data Types Read/Write Access Functions Prototypes Check: Adherence to IVI 3.4 • Naming Conventions • Parameter Types • Help Strings Other 3.4 Topics • Behavior • State Caching
Unit Tests	Test: Function & Attributes Perform as Intended • Nominal Values • Boundary Conditions • Proper Coercion of Values • Errors are Logged and Handled • Proper Error Codes are Returned • Handling of Illegal Values Parameters (When Ignored)
System Tests	Test: Driver Attributes/Functions in a Realistic Application Program For Couplings and Dependencies For Memory Leaks Driver Can be Called From a Class Interface
ADE Tests	Test: Syntactic Performance in ADE If the driver provides an interface not specified by the IVI Foundation, it should be tested in an appropriate ADE. Supplied example programs
Help Tests	Check: • Style • Topic Content • Cross References • Table of Contents Hyperlinks • Spelling • Context Sensitivity
Performance Tests	Test: Data Throughput Function/Attribute Execution Time Memory Usage
Interoperability Tests	Check: Driver Performance With Other IVI Drivers, participate in the IVI Foundation Interoperability Forum.

Appendix C – Contents of IviVisaType.h File

This file is provided as a reference and may not have the same date or version as the actual file installed on the system.

```
/****************************************************************************
 * IviVisaType.h
 *
 * Copyright (c) Interchangeable Virtual Instruments Foundation 2006 - 2016.
 * All Rights Reserved.
 *
 ****************************************************************************/
#ifndef IVI_VISA_TYPE_H
#define IVI_VISA_TYPE_H


/* This defines the include guard of visatype.h for backward compatibility
 * reasons. Please ensure that changes in this ifndef block are reflected
 * in visatype.h when necessary.
 */
#ifndef __VISATYPE_HEADER__
#define __VISATYPE_HEADER__


#if defined(_WIN64)
#define _VI_FAR
#define _VI_FUNC            __fastcall
#define _VI_FUNCC           __fastcall
#define _VI_FUNCH           __fastcall
#define _VI_SIGNED          signed
#elif (defined(WIN32) || defined(_WIN32) || defined(__WIN32__) || defined(__NT__)) &&
!defined(_NI_mswin16_)
#define _VI_FAR
#define _VI_FUNC            __stdcall
#define _VI_FUNCC           __cdecl
#define _VI_FUNCH           __stdcall
#define _VI_SIGNED          signed
#elif defined(_CVI_) && defined(_NI_i386_)
#define _VI_FAR
#define _VI_FUNC            _pascal
#define _VI_FUNCC
#define _VI_FUNCH           _pascal
#define _VI_SIGNED          signed
#elif (defined(_WINDOWS) || defined(_Windows)) && !defined(_NI_mswin16_)
#define _VI_FAR             _far
#define _VI_FUNC            _far _pascal _export
#define _VI_FUNCC           _far _cdecl  _export
#define _VI_FUNCH           _far _pascal
```

```
#define _VI_SIGNED          signed
#elif (defined(hpux) || defined(__hpux)) && (defined(__cplusplus) || defined(__cplusplus__))
#define _VI_FAR
#define _VI_FUNC
#define _VI_FUNCC
#define _VI_FUNCH
#define _VI_SIGNED
#else
#define _VI_FAR
#define _VI_FUNC
#define _VI_FUNCC
#define _VI_FUNCH
#define _VI_SIGNED          signed
#endif

#define _VI_ERROR           (-2147483647L-1)  /* 0x80000000 */
#define _VI_PTR             _VI_FAR *

/*- VISA Types --------------------------------------------------------------*/

#ifndef _VI_INT64_UINT64_DEFINED
#if defined(_WIN64) || ((defined(WIN32) || defined(_WIN32) || defined(__WIN32__) || defined(__NT__)) &&
!defined(_NI_mswin16_))
#if (defined(_MSC_VER) && (_MSC_VER >= 1200)) || (defined(_CVI_) && (_CVI_ >= 700)) ||
(defined(__BORLANDC__) && (__BORLANDC__ >= 0x0520)) || defined(__LCC__) || (defined(__GNUC__) &&
(__GNUC__ >= 3)) || (defined(__clang__) && (__clang_major__ >= 3))
typedef unsigned   __int64  ViUInt64;
typedef _VI_SIGNED __int64  ViInt64;
#define _VI_INT64_UINT64_DEFINED
#if defined(_WIN64)
#define _VISA_ENV_IS_64_BIT
#else
/* This is a 32-bit OS, not a 64-bit OS */
#endif
#endif
#elif defined(__GNUC__) && (__GNUC__ >= 3)
#include <limits.h>
#include <sys/types.h>
typedef u_int64_t           ViUInt64;
typedef int64_t             ViInt64;
#define _VI_INT64_UINT64_DEFINED
#if defined(LONG_MAX) && (LONG_MAX > 0x7FFFFFFFL)
#define _VISA_ENV_IS_64_BIT
#else
/* This is a 32-bit OS, not a 64-bit OS */
#endif
```

```
#else
/* This platform does not support 64-bit types */
#endif
#endif

#if defined(_VI_INT64_UINT64_DEFINED)
typedef ViUInt64    _VI_PTR ViPUInt64;
typedef ViUInt64    _VI_PTR ViAUInt64;
typedef ViInt64     _VI_PTR ViPInt64;
typedef ViInt64     _VI_PTR ViAInt64;
#endif

#if defined(LONG_MAX) && (LONG_MAX > 0x7FFFFFFFL)
typedef unsigned int        ViUInt32;
typedef _VI_SIGNED int      ViInt32;
#else
typedef unsigned long       ViUInt32;
typedef _VI_SIGNED long     ViInt32;
#endif

typedef ViUInt32    _VI_PTR ViPUInt32;
typedef ViUInt32    _VI_PTR ViAUInt32;
typedef ViInt32     _VI_PTR ViPInt32;
typedef ViInt32     _VI_PTR ViAInt32;

typedef unsigned short      ViUInt16;
typedef ViUInt16    _VI_PTR ViPUInt16;
typedef ViUInt16    _VI_PTR ViAUInt16;

typedef _VI_SIGNED short    ViInt16;
typedef ViInt16     _VI_PTR ViPInt16;
typedef ViInt16     _VI_PTR ViAInt16;

typedef unsigned char       ViUInt8;
typedef ViUInt8     _VI_PTR ViPUInt8;
typedef ViUInt8     _VI_PTR ViAUInt8;

typedef _VI_SIGNED char     ViInt8;
typedef ViInt8      _VI_PTR ViPInt8;
typedef ViInt8      _VI_PTR ViAInt8;

typedef char                ViChar;
typedef ViChar      _VI_PTR ViPChar;
typedef ViChar      _VI_PTR ViAChar;
typedef unsigned char       ViByte;
typedef ViByte      _VI_PTR ViPByte;
typedef ViByte      _VI_PTR ViAByte;

typedef void        _VI_PTR ViAddr;
typedef ViAddr      _VI_PTR ViPAddr;
typedef ViAddr      _VI_PTR ViAAddr;

typedef float               ViReal32;
typedef ViReal32    _VI_PTR ViPReal32;
typedef ViReal32    _VI_PTR ViAReal32;

typedef double              ViReal64;
typedef ViReal64    _VI_PTR ViPReal64;
typedef ViReal64    _VI_PTR ViAReal64;

typedef ViPByte             ViBuf;
typedef ViPByte             ViPBuf;
typedef ViPByte     _VI_PTR ViABuf;

typedef ViPChar             ViString;
typedef ViPChar             ViPString;
typedef ViPChar     _VI_PTR ViAString;

typedef ViString            ViRsrc;
typedef ViString            ViPRsrc;
typedef ViString    _VI_PTR ViARsrc;

typedef ViUInt16            ViBoolean;
typedef ViBoolean   _VI_PTR ViPBoolean;
typedef ViBoolean   _VI_PTR ViABoolean;

typedef ViInt32             ViStatus;
typedef ViStatus    _VI_PTR ViPStatus;
typedef ViStatus    _VI_PTR ViAStatus;

typedef ViUInt32            ViVersion;
typedef ViVersion   _VI_PTR ViPVersion;
typedef ViVersion   _VI_PTR ViAVersion;

typedef ViUInt32            ViObject;
typedef ViObject    _VI_PTR ViPObject;
typedef ViObject    _VI_PTR ViAObject;
typedef ViObject            ViSession;
typedef ViSession   _VI_PTR ViPSession;
typedef ViSession   _VI_PTR ViASession;

typedef ViUInt32             ViAttr;

#ifndef _VI_CONST_STRING_DEFINED
typedef const ViChar * ViConstString;
#define _VI_CONST_STRING_DEFINED
#endif

/*- Completion and Error Codes ----------------------------------------------*/

#define VI_SUCCESS          (0L)

/*- Other VISA Definitions --------------------------------------------------*/

#define VI_NULL             (0)

#define VI_TRUE             (1)
#define VI_FALSE            (0)

/*- Backward Compatibility Macros -------------------------------------------*/

#define VISAFN              _VI_FUNC
#define ViPtr               _VI_PTR

#endif /* __VISATYPE_HEADER__ */

/* This defines the include guard of vpptype.h for backward compatibility
 * reasons. Please ensure that changes in this ifndef block are reflected
 * in vpptype.h when necessary.
 */
#ifndef __VPPTYPE_HEADER__
#define __VPPTYPE_HEADER__

/*- Completion and Error Codes ----------------------------------------------*/

#define VI_WARN_NSUP_ID_QUERY     (          0x3FFC0101L)
#define VI_WARN_NSUP_RESET        (          0x3FFC0102L)
#define VI_WARN_NSUP_SELF_TEST    (          0x3FFC0103L)
#define VI_WARN_NSUP_ERROR_QUERY  (          0x3FFC0104L)
#define VI_WARN_NSUP_REV_QUERY    (          0x3FFC0105L)
```

```
#define VI_ERROR_PARAMETER1       (_VI_ERROR+0x3FFC0001L)

#define VI_ERROR_PARAMETER2       (_VI_ERROR+0x3FFC0002L)

#define VI_ERROR_PARAMETER3       (_VI_ERROR+0x3FFC0003L)

#define VI_ERROR_PARAMETER4       (_VI_ERROR+0x3FFC0004L)

#define VI_ERROR_PARAMETER5       (_VI_ERROR+0x3FFC0005L)

#define VI_ERROR_PARAMETER6       (_VI_ERROR+0x3FFC0006L)

#define VI_ERROR_PARAMETER7       (_VI_ERROR+0x3FFC0007L)

#define VI_ERROR_PARAMETER8       (_VI_ERROR+0x3FFC0008L)

#define VI_ERROR_FAIL_ID_QUERY    (_VI_ERROR+0x3FFC0011L)

#define VI_ERROR_INV_RESPONSE     (_VI_ERROR+0x3FFC0012L)


/*- Additional Definitions --------------------------------------------------*/


#define VI_ON               (1)

#define VI_OFF              (0)


#endif /* __VPPTYPE_HEADER__ */


#endif /* IVI_VISA_TYPE_H */
```