IVI-3.4: API Style Guide

February 11, 2014 Edition
Revision 2.3

Important Information

The API Style Guide (IVI-3.4) is authored by the IVI Foundation member companies. For a vendor membership roster list, please visit the IVI Foundation web site at `www.ivifoundation.org`.

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

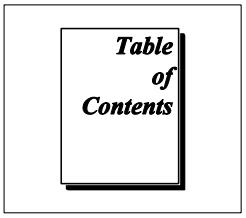
Warranty

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

Trademarks

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

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



API Style Guide

API Style Guide Revision History

This section is an overview of the revision history of the IVI-3.4 specification.

Table 1-1. Ivi-3.4 Revisions

Revision Number	Date of Revision	Revision Notes
Revision 1.0	March 2008	First approved version.
Revision 1.1	November 17, 2008	Describe style for Absolute Time
Revision 1.2	October 20, 2009	Add support for Instrument Capability Discovery.
Revision 2.0	June 9, 2010	Incorporated IVI.NET
Revision 2.1	March 6, 2013	Clarification on coupled attributes and configuration functions Added Section 16: Direct I/O
Revision 2.2	October 25, 2013	Minor change - Added a new section for Asynchronous I/O for IVI.NET drivers Editorial changes in Direct IO section
Revision 2.3	November 14, 2013	Change Section 5.1 to apply to IVI-COM only. Add Section 5.3 to describe IVI.NET versioning details.
Revision 2.3	December 09, 2013	Editorial change in Sections 16.1.3 and 16.1.4 to make the Session and System attributes read-only.
Revision 2.3	February 11, 2014	Change Section 5 to clarify that both policy files and side-by-side installation are a key part of making the IVI.NET versioning strategy work.

1. Overview of the API Style Guide

1.1 Introduction

This specification is primarily written to direct the efforts of working groups writing the *IVI-3.2 Inherent Capabilities Specification*, *IVI-3.3 Standard Cross-Class Capability Specification*, shared components, and instrument class specifications. Specifications should have a common format, similar presentation of content, and consistent design of capability groups. By providing a consistent style, IVI specific driver writers will find information presented in a familiar manner and thus avoid overlooking requirements. By adhering to this style, specification writers should avoid accidentally omitting information.

IVI specific drivers will invariably contain functions and attributes not described in the instrument class specification. Driver writers are expected to follow the style described here for those functions and attributes. Throughout the style guide names are shown that contain a class name. For IVI specific drivers, the same rules apply except the class name is replaced by the instrument specific prefix.

1.2 Overview

This specification applies to all IVI Instrument Class Specifications approved after this specification is approved. It provides rules and recommendations on organization, naming, choosing parameter types, required guidance to driver writers, and more.

- Existing instrument class specifications also provide insight into appropriate style The following instrument class specifications, however, were adopted before this specification was completed:
  IviScope
- IviDmm
- IviFgen
- IviDCPwr
- IviSwtch

Therefore these specifications may use a style different from the one in this specification.

1.3 References

Several other documents and specifications are related to this specification. These other related documents are as follows:

- VXI*plug&play* VPP-3.2 Instrument Driver Functional Body Description
- VXI*plug&play* VPP-3.4 Instrument Driver Programmatic Developer Interface Specification
- IVI-3.2: IVI Inherent Capabilities Specification
- IVI-3.3: Standard Cross-Class Capabilities Specification
- IEEE 488.2-1987 IEEE Standard Codes, Formats, Protocols, and Common Commands.

1.4 Definitions of Terms and Acronyms

This specification does not define additional terms or acronyms. Refer to *IVI–5: Glossary* for a description of terms used in this specification.

2. Approach to Designing Instrument Class Interfaces

Instrument Class working groups face many decisions while writing a specification. Some of these issues have been faced and dealt with by previous working groups and their wisdom can be applied to future work.

2.1 Development Process

Every specification goes through a process from its inception to approval to maintenance. This process is defined and controlled by the technical working group. Contact its chairman for details.

2.2 Scope of an Instrument Class Specification

One of the first questions a working group should answer is "What features are appropriate to include in this instrument class?" This answer defines the instrument class.

From this set of features, a few comprise the base capability. These features are commonly found in all instruments of this instrument class.

The remaining features are grouped into extension groups. Creating appropriate extension groups and allocating features to them is a difficult process which requires the insight of many experts who understand a variety of instruments in the instrument class. Placing too many features in one extension group may prevent a particular instrument driver from implementing that group. All the instruments in a class may provide a large variety of triggering methods though no one instrument implements all the methods. Each triggering method thus belongs in its own extension group. If one feature, however, fundamentally requires another feature then both belong in the same extension group.

Features found in only a small number of instrument models should not be included in the instrument class. No gain in interchangeability is achieved if only one or two instruments have a particular extension group. The user should access these specialized features through driver-specific means.

During early instrument class development, features may be added or removed. They may migrate between capability groups. As the specification matures, movement of features should decrease and finally cease.

2.3 Attributes

An instrument's state can generally be modeled with a vector of attributes. The instrument class specification defines a set of attributes within each capability group.

Each attribute can be set separately. Groups of attributes can also be set in a single configuration function call.

Attributes are classified as coupled if the legal range, or set of valid values, of one depends on the other. An attribute is classified as uncoupled if the legal range, or set of valid values, of the attribute does not depend on nor change the value of any another attributes.

2.3.1 Coupled Attributes

Coupled attributes affect each other. The legal settings for an attribute coupled to another attribute depend on the other attribute's setting. Instrument classes provide a configuration function which sets all the attributes which are coupled to each other. Thus the user can set all the coupled attributes at once and the driver and instrument can sort out the coupling.

Note that more than two attributes may be coupled, such as frequency, pulse width, rise time and fall time in a pulse generator. Also, a numeric parameter may be coupled with an enumerated parameter such as trigger source and trigger level where certain trigger sources may have different signal conditioning.

Coupled attributes generally reflect coupled instrument state variables. For example, the function and range in a digital multimeter are coupled. While 1e6 is often a valid range when the function is resistance, it is not often valid when the function is DC volts. So the IviDmm instrument class provides a function to set both attributes in one call.

The following principles should be used to determine if attributes that reflect instrument state variables are coupled.

- For class APIs, attributes are coupled if the corresponding instrument state variables are "commonly" coupled in instruments to which the class applies. (Refer to section 2.2, *Scope of an Instrument Class Specification*~~*Scope of an Instrument Class Specification*~~ for more discussion of what is appropriate in class API definitions.)
- For instrument specific APIs, attributes are coupled if the corresponding state variables are coupled in the instrument behavior.

Formatted: Font: Italic

2.3.2 Uncoupled Attributes

The legal settings of an uncoupled attribute do not depend upon any other instrument settings. Uncoupled attributes may appear in configuration functions.

For example, the trigger slope in a digital multimeter is independent of any other attribute.

2.4 Functions

The functions described by the API fall into three categories.

2.4.1 Configuration

A Configuration function sets one or more attributes that are coupled or related in some other way. Refer to section 2.3.1, *Coupled Attributes*~~*Coupled Attributes*~~ for more details.

Formatted: Font: Italic

Configuration functions are especially useful when dealing with coupled attributes. In this case, the configuration function includes a parameter for each of the coupled attributes set by the function. The driver resolves couplings to ensure that when the user specifies a legal state that the instrument reaches that state without error. Sometimes this resolution is done with the assistance of processing capabilities in the instrument itself. Other times the driver resolves any coupling conflicts itself before programming the instrument.

Configuration functions may be used to set coupled values that are not represented as settable attributes in the API. If two instrument or driver values are so connected that the act of setting one commonly invalidates or changes the value of the other, it may not be desirable to provide attributes that the user can set independently. Instead, a configuration function should be used to set coupled parameters. Read-only attributes should be used to read back the individual values of the coupled parameters.

2.4.2 Action

An Action routine makes the instrument perform an operation which depends on the present configuration. The operation generally does not involve changing any attribute's value. Triggering a measurement or starting a sweep are examples of action functions.

2.4.3 Retrieve Measurement Data

Two forms of a function which retrieve measurement date are commonly found in IVI drivers:

Fetch - The instrument returns already available measurement data. If the instrument is acquiring data, the driver waits until the acquisition is complete before returning data.

Read - The instrument initiates a new measurement and waits for the measurement to complete before returning data.

2.5 Relationship of Attributes to Configuration Functions

For IVI-C, the instrument class specification should define for each attribute exactly one high level configuration function. A possible exception is an attribute which can be changed in several capability groups. Often, several configuration functions control single, uncoupled attributes. While a user could use the Set Attribute function, these configuration functions provide better visibility into the functionality of the instrument class and instrument.

For IVI-COM and IVI.NET, an uncoupled attribute may not have a corresponding high level configuration function. The attribute can be configured by setting the corresponding property.

For IVI-C, IVI-COM, and IVI.NET, configuration function parameters that set an instrument state variable shall have a corresponding attribute. The following naming conventions shall be observed:

- For IVI-COM and IVI.NET, the name of the parameter should correspond to the name of the corresponding property. For example, if a configure function includes a parameter named “foo”, the corresponding property should be named `Foo`.
- For IVI-C the corresponding attribute should end with 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`.

3. Naming Conventions

Naming things and spelling those names can be a surprisingly contentious subject. As the English language has demonstrated, inconsistent spelling leads to frustration. The rules and recommendations here are intended to reduce a user's frustration level when discovering how the large variety of names used in IVI drivers are determined.

Avoid abbreviations. If a name contains so many words that it becomes objectionably long, typically more than 50 characters, abbreviations are allowed. If a word is abbreviated in one name, abbreviate that word everywhere and abbreviate it the same way in all names. Always abbreviate maximum as max and minimum as min.

Use acronyms only when they are more understandable than the complete word.

3.1 Instrument Class Names

An instrument class name may contain up to thirty-one total characters. The first three characters shall be "Ivi" using mixed case. The remaining characters are left to the discretion of the instrument class specification working group. The only constraint is that the resulting instrument class name cannot conflict with any existing instrument class name.

The size limit results from the constraints on the Class Driver Prefix attribute described in Section 5.5, *Class Driver Prefix* in *IVI-3.2: Inherent Capabilities Specification*. The string that this attribute returns contains a maximum of 32 bytes including the NUL byte.

IMPORTANT: Since this name is prolific throughout a instrument class specification, it should be as short as possible while leaving no ambiguity about the nature of the instruments in the class.

Two special forms of the instrument class name are used throughout this document where the instrument class name is substituted into a name.

<ClassName> means substitute the instrument class name with mixed case. For example, “IviDCPwr”. <InstrType> means substitute <ClassName> with the leading “Ivi” omitted. For example, “DCPwr”

<CLASS_NAME> means substitute the instrument class name in all upper-case characters. For example, “IVIDCPWR”. <INSTR_CLASS> means substitute <CLASS_NAME> with the leading “IVI” omitted. For example, “DCPWR”.

3.2 Capability Group Names

A capability group name may contain an arbitrary number of characters. The first characters shall be the instrument class name, <ClassName>. The capability group is described using complete English words. These words are appended to the instrument class name with no spaces between the words and the first character of each word is capitalized.

For example, `IviDCPwrSoftwareTrigger`. The prefix, IviDCPwr, is the instrument class name. The complete English words, Software and Trigger, describe the capability group.

3.3 Generic Function Names

A generic (e.g. independent of the API type) function name consists of one or more complete English words which describe the operation performed. Refer to existing instrument class specifications for commonly used naming patterns. The first word should be a verb or at least imply performing an action.

For example, Configure Edge Trigger Source.

A generic function name shall be one or more complete words, with the first character of each word capitalized and the words separated by a space. When functions are referenced generically, use the generic function name in a Times font. Do not use the IVI-C, IVI-COM, or IVI.NET form of the name.

3.4 Generic Attribute Names

A generic (e.g. independent of the API type) attribute name consists of one or more complete English words which describe what is controlled. Be concise, but include enough words to avoid ambiguity.

Be consistent with word order. If a word appears in multiple attributes, put it first before any modifying words. Words which describe the core functionality come first, followed by words which qualify the previous word. A useful side effect of these rules becomes apparent when attributes are listed alphabetically. Related attribute name and values are listed together. For example in Waveform Size Min, waveform is the core functionality. Size qualifies waveform and Min qualifies size.

Random order frustrates. Having one attribute named Max Waveform Size and another Waveform Size Min would be extremely annoying.

For example, TV Trigger Signal Format.

A generic attribute name shall be one or more complete words, with the first character of each word capitalized and the words separated by a space. When attributes are referenced generically, use the generic attribute name in a Times font. Do not use the IVI-C, IVI-COM, or IVI.NET form of the name.

3.5 IVI.NET

This section describes the naming rules for IVI.NET source.

3.5.1 General Casing Rules

IVI.NET code elements use either Pascal casing or camel casing. For camel casing, the first letter of the name is lowercase. For Pascal casing, the first letter of the name is uppercase. Otherwise, the following rules apply to both:

- The first characters of each word, abbreviation, or acronym in the name is uppercase.
- Both characters of a two-letter acronym are uppercase.
- For an acronym of three characters or more, only the first character is uppercase.
- There is one exception for enumeration values. Refer to the section on Enumerations and Enumeration Values below.

3.5.2 Namespaces

All namespaces shall use Pascal casing.

The namespace for the IVI.NET inherent capabilities described in *IVI-3.2: Inherent Capabilities Specification*, and additional utility classes and interfaces described in *IVI-3.18: IVI.NET Utility Classes and Interfaces Specification*, shall be “Ivi.Driver”.

The namespace for an IVI instrument class shall be “Ivi.<ClassType>”. For example, “Ivi.Dmm”. The instrument class specification shall include the following sub-section in Section 2.

2.5 .NET Namespaces

The .NET namespace for the <ClassName> class is `Ivi.<ClassType>`.

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.

3.5.3 Interfaces

Interfaces shall use Pascal casing.

Interfaces described in *IVI-3.2: Inherent Capabilities Specification*, shall begin with “IIviDriver”. The root interface in the inherent capabilities hierarchy shall be named “IIviDriver”.

Class-compliant interfaces shall begin with “I<ClassName>”. The root interface in the class-compliant hierarchy shall be named “I<ClassName>”.

For example:

- interface IIviScope

Interfaces defined for other IVI components including those defined in, *IVI-3.18: IVI.NET Utility Intefaces and Classes Specification*, shall begin with “IIvi”.

Interfaces defined for IVI.NET specific drivers shall begin with “I”, followed by one or more words that describe the interface. If the IVI.NET specific driver has a root interface, it shall be named “I<ComponentIdentifier>”

For example:

- interface ITrigger
- interface IAgilent34410

There are two interfaces associated with a collection style repeated capability, the collection interface and the member interface. For instrument classes, the collection interface shall be I<ClassName><Rc>Collection, and the member interface shall be I<ClassName><Rc>, where <Rc> is the repeated capability name. For specific driver repeated capability collections, the collection interface shall be I<Rc>Collection, and the member interface shall be I<Rc>.

3.5.4 Classes and Structs

Classes and structs shall use Pascal casing.

Class and struct names may be prefixed with instrument class <ClassName> or specific driver <Component Identifier> for clarity, but need not be. The main IVI driver class shall be named <*ComponentIdentifier*>.

IVI defined .NET exceptions are not decorated with instrument class names.

For example:

- class Waveform<T>
- struct PrecisionTimeSpan

3.5.5 Enumerations and Enumeration Members

Enumeration type names shall use Pascal casing. In contrast to IVI-COM enumeration type names, IVI.NET enumeration names do not include the IVI instrument class name, component identifier, or the string “Enum”

Enumeration member names shall use Pascal casing. In contrast to IVI-COM enumeration type names, IVI.NET enumeration names do not include the IVI instrument class name or component identifier, or the name of the enumeration.

For example:

```
enum Slope
{
    Negative = 0
    Positive = 1,
}
```

Enumeration members that reference IEEE or ANSI-standard measurement units shall follow the capitalization rules associated with the unit of measure. If there is ambiguity, the form of the unit shall be the one used with numbers. For example, “dB in volts” would be “dBV,” rather than “Dbv.”

3.5.6 Method and Parameter Names

Method names shall use Pascal casing. Do not use the interface name or its corresponding interface property name in the method name. For example a method in IIviDmmTrigger would be Configure rather than ConfigureTrigger.

Parameter names shall use camel casing. Do not use the instrument class name in parameter names.

Use multiple words to make a parameter's usage obvious. Parameter names in a method prototype are a form of documentation.

For example:

```
Double Fetch(TimeSpan maxTime, out Boolean sampleOutOfRange);
```

If the parameter is also present in the IVI-C and IVI-COM interfaces, the same name should be used in the IVI.NET interface, with IVI.NET casing rules.

3.5.7 Properties

Property names shall use Pascal casing. Do not use the interface name or its corresponding interface property name in the property name. For example a property in IIviDmmTemperature would be TransducerType rather than TemperatureTransducerType.

For example:

```
PrecisionTimeSpan Delay { get; set; }
```

3.5.8 Interface Reference Properties

Properties that return references to other interfaces in the driver are called Interface Reference Properties. The names of the these properties shall be the trailing words of the type of interface the property points to. The “I” followed by the instrument class prefix or component identifier is omitted.

For example:

```
IIviDmmTrigger Trigger { get; }
```

3.6 IVI-C

This section describes the naming rules for IVI-C source.

3.6.1 Acronyms

Capitalize acronyms as if they are single words, e.g. CW, RTD, RADAR.

3.6.2 Functions

IVI function names shall consist of <ClassName>, an underscore character, and the function name with the spaces removed. The only underscore character is the one after the instrument class name.

For example, the function, Configure Edge Trigger Source becomes `IviScope_ConfigureEdgeTriggerSource`.

3.6.3 Parameter Names

Do not use the instrument class name in parameter names. Capitalize each word in a parameter's name regardless of the number of words in the name.

Use multiple words to make a parameter's usage obvious. Parameter names in a function prototype are a form of documentation.

Use the same parameter name in both IVI-COM method prototypes and IVI-C prototypes for the same function.

For example, InputImpedance.

3.6.4 Attributes

Since attributes names are macros they use only uppercase characters. The full attribute name begins with:

`<CLASS_NAME>_ATTR_`

The remainder of the IVI-C attribute name is the attribute name in all upper-case characters with spaces replaced by underscores.

For example, TV Trigger Signal Format becomes `IVISCOPE_VAL_TV_TRIGGER_SIGNAL_FORMAT`

3.6.5 Defined Values

Some properties and function parameters have defined values. These defined value names are macros, so they use only upper-case characters. A defined value name begins with:

`<CLASS_NAME>_VAL_`

The remainder of the name is also in all uppercase. If the remainder uses multiple words, they are separated by underscore characters.

For example, IVISCOPE_VAL_NTSC.

3.7 IVI-COM

This section describes the naming rules for IVI-COM source.

3.7.1 Acronyms

Capitalize acronyms as if they are single words, e.g. CW, RTD, RADAR.

3.7.2 Interface

IVI-COM interface naming standards are described in section 4.1.11, *Interface Requirements*, of *IVI-3.1: Architecture*, and subsections.

3.7.3 Methods

The hierarchy context plus the method name shall communicate the same content as the generic function name. The method name may omit some of the words in the function name if the interface pointer names provide the same information.

IVI-COM method prototypes are of the form:

```
HRESULT <MethodName> ( various parameters );
```

For example, the function, Configure Edge Trigger Source becomes `Trigger.Edge.Configure`.

3.7.4 Parameter Names

Do not use the instrument class name in parameter names. Capitalize each word in a parameter's name regardless of the number of words in the name.

Use multiple words to make a parameter's usage obvious. Parameter names in a function prototype are a form of documentation.

Use the same parameter name in both IVI-COM method prototypes and IVI-C prototypes for the same function.

For example, InputImpedance.

3.7.5 Properties

The hierarchy context plus the property name shall communicate the same content as the attribute name. The property name may omit some of the words in the attribute name if the interface pointer names provide the same information.

IVI-COM Property Names do not have any prefix. They are generally the same as the attribute name with the spaces removed except when some of the words are omitted.

For example, Class Driver Major Revision becomes `ClassDriverMajorRevision` and TV Trigger Signal Format becomes `Trigger.TV.SignalFormat`

Do not use the interface name or its corresponding interface property name in the property name. For example a property in IIviDmmTrigger would be Count rather than TriggerCount.

3.7.6 Enumeration Members

Defined value names are contained within an IVI- COM enumeration. An IVI-COM enumeration name is of the form:

```
<ClassName><descriptive words>Enum
```

Each word is capitalized with no spaces or underscores. The suffix, Enum, shall be included.

For example, IviScopeTriggerCouplingEnum

The defined values within the enumeration are of the form:

```
<ClassName><words from enumeration name><words>
```

The same words from the enumeration name shall appear in every value for a particular enumeration. The trailing word or words shall be picked to differentiate the values.

For example, IviScopeTriggerCouplingAC and IviScopeTriggerCouplingDC.

3.7.7 Interface Reference Properties

A special property is one which points to an interface. The names of the these properties shall be the trailing words of the type of interface the property points to. The instrument class name prefix, `I<ClassName>` is omitted.

For example, a property which points to a interface of type `IIviScopeReferenceLevel` is named `ReferenceLevel`.

4. Parameter Types

Functions and methods pass parameters which always have a type. Attributes and properties have a type. Using the right type in the right situation helps a user intuitively know a parameter's type.

Note that the use of ref (or C# “out”) parameters is discouraged in IVI.NET. The stylistic preference is to avoid ref parameters in favor of return values in IVI.NET, but they are allowed in situations that call for it.

The type of a parameter or an attribute may be given using a general term or a language specific type. The other types can be inferred using Table 5-6. *Compatible Data Types for IVI Drivers* in *IVI-3.1: Driver Architecture Specification*. Instrument class specifications often use just the ANSI C type.

4.1 Integers

Things which can be naturally counted should be integers. If a parameter can be thought of as a number of things, use an integer. The number of points in a trace or the number of triggers are naturally integers.

32-bit integer types should be the default choice for representing integers. In cases where instruments return large arrays of bytes or 16-bit integers, 8-bit or 16-bit integer types may be used. In cases where drivers need higher precision than 32-bit integers can provide, 64-bit integers may be used.

4.2 Reals

Any continuous value should be a real. If something can logically ever be a non-integer value, make it a real. Anything with physical units, such as voltage, power, frequency, length, angle, etc. should be a real.

64-bit floating point types should be the default choice for representing real numbers. In cases where instruments use large arrays of 32-bit floating point numbers, 32-bit floating point types may be used. For IVI.NET only, the Decimal type may be used when higher precision is needed.

4.2.1 Continuous Ranges and Discrete Values

Instrument classes should specify discrete legal values for a parameter only when intermediate values are not meaningful. In such cases, the instrument class shall specify that coercion is not allowed. IVI specific drivers report an error when a user specifies a value not in the set of legal values. Instrument classes should encourage IVI specific drivers to do the same where the IVI class-compliant specific driver takes a subset of the values defined by the instrument class.

In general, instrument classes should allow continuous ranges of values for real-valued parameters and attributes, even if some instruments in the instrument class implement only a discrete set of values for the setting. Some instruments that implement only a discrete set of values might accept a continuous range of values and coerce user-specified values to the discrete set. Other instruments might accept only the discrete set, in which case the IVI specific driver accepts a continuous range and coerces user-specified values to the discrete set accepted by the instrument. The instrument class should specify whether the value from the user should be coerced up, down, to the nearest arithmetic value, or to the nearest geometric value. If an instrument performs coercion in a manner different from what the instrument class specifies, the IVI specific driver coerces the value before sending it to the instrument.

Instrument classes should not specify a minimum or maximum value for a continuous range unless the value reflects an inherent limitation of the setting. An example of a setting with an inherent limitation is the vertical range on an oscilloscope, which cannot be a negative number.

4.2.2 Infinity and Not A Number

Use the IEEE 754 representations for positive infinity (PInf), negative infinity (NInf) and not a number (NaN). Instrument classes shall specify the exact circumstances when these values are returned. For example, a function could be required to return NaN when a measurement value could not be generated.

For IVI.NET the System.Double and System.Single floating point types provide properties to represent NaN, PInf, and NInf, and methods to test for each of those values. System.Decimal does not include these properties and methods, and should be avoided if they are needed.

For IVI-C and IVI-COM, a shared component, defined in *IVI-3.12: Floating Point Services Specification*, is available for driver writers to get these defined values. IVI drivers use the shared component to reduce the possibility of incompatibility with other implementations. For functions that can return one of these special values, the instrument class shall provide functions which compare any number against these special values. Don't require the user to make comparisons with NInf or PInf or Nan, but provide functions which do the comparison inside the driver.

4.3 Enumerations

An enumeration is a good choice for a parameter which has a natural association to several discrete possibilities and the possibilities have no numeric content. If adding or removing a possibility is difficult, the parameter should probably not be an enumeration, but rather an integer or real. For example, HIGH, MEDIUM, and LOW are poor choices as removing High would leave Medium and Low. Likewise, adding a choice creates a badly named value such as Medium High. In this case, the parameter should be a real or integer with appropriate coercion to a value the instrument can handle.

4.3.1 IVI.NET

Enumerations in IVI.NET are defined by declaring an enumeration type.

For example:

```
enum <EnumName>
{
    <EnumMember1> = 0,
    <EnumMember2> = 1
}
```

Where <EnumName> is an IVI.NET enumeration name, and <EnumMember1> and <EnumMember2> are enumeration values, generated using the rules in Section 3.5.5, *Enumerations and Enumeration Members*~~*Enumerations & Enumeration Values*~~.

Formatted: Font: Italic

4.3.2 IVI-C

IVI-C enumerations are implemented as 32-bit integers, and enumeration values are implemented as named 32-bit constants.

The names of the values associated with an enumeration shall have a prefix of the form

```
<CLASS NAME>_VAL_
```

The remainder of the enumeration value name should be easily associated with the parameter or attribute being set.

4.3.3 IVI-COM

Enumerations in IVI-COM are defined using an enumeration typedef in the IDL. Every typedef for an enumeration shall include these attributes:

- public - so the alias becomes part of the type library
- v1_enum - so the enumerated type is a 32-bit entry, rather than the 16-bit default.

The IDL for enumerations is of the form:

```
typedef [public, v1_enum, HELP("ENUM_<enum_name>")]
enum prefix<enum_name>Enum
{
    prefix<enum_name>Choice1            = 0,
    prefix<enum_name>Choice2            = 1,
} prefix<enum_name>Enum;
```

Where <enum_name> is the IVI-COM enumeration name generated using the rules in Section 3.7.6, *Enumeration ~~Enumeration Defined Values~~*.

Formatted: Font: Italic

4.4 Strings

Handling strings is generally cumbersome and they should be avoided especially for input parameters. Users get spelling and word order wrong very easily. Parsing an input string is generally harder than handling a numeric type and interchangeable parsers may be too much to expect. If only a finite choice of strings make sense, use an enumeration.

Input strings are necessary when something has a name, such as a channel or trace.

Output strings whose only use is to be read by the user are, however, a good choice. Strings elucidating an error code are quite useful. The string is generated by the driver or forwarded from the instrument and the user is never expected to programmatically parse it, only display it where a human can parse it.

	Input	Output
.NET	`System.String`	`out System.String System.Text.StringBuilder`
C	`ViConstString`	`ViChar[]`
COM	`BSTR`	`BSTR *`

4.4.1 IVI.NET

Memory allocation for the System.String type is handled by the .NET runtime. Since new memory is allocated every time a String variable is assigned, building a string in several steps is relatively inefficient. Strings can be built more efficiently by using the System.StringBuilder class.

4.4.2 IVI-C

In IVI-C, include a BufferSize parameter before the string parameter. If `<string>` is the name of the string parameter, the BufferSize parameter shall be `ViInt32 <string>BufferSize`. Since the string is null terminated, an actual size parameter is not required.

The IVI driver can expect the user to allocate at least an many characters for the string as the value of BufferSize. If the value of the string parameter is VI_NULL, the value of BufferSize may be zero.

Instrument class specifications shall also follow the rules in Section 3.1.21 Additional Compliance Rules for C Functions with ViChar Array Output Parameters in IVI 3.2: Inherent Capabilities Specification.

4.4.3 IVI-COM

The IVI driver allocates the memory for a BSTR so a size parameter is not needed.

4.5 Booleans

Many parameters which might otherwise be an enumeration but could only ever have two values are often best represented as Boolean. Specifications use “True” and “False” as Boolean values. Where Booleans naturally represent On/Off or Yes/No, “On” and “Yes” shall correspond to “True”, and “Off” and “No” shall correspond to “False”.

The instrument class specification shall include the following sub-section in Section 2.

2.4 Boolean Attribute and Parameter Values

This specification uses True and False as the values for Boolean attributes and parameters. The following table defines the identifiers that are used for True and False in the IVI.NET, IVI-COM, and IVI-C architectures.

Boolean Value	IVI.NET Identifier	IVI-COM Identifier	IVI-C Identifier
True	`true`	`VARIANT_TRUE`	`VI_TRUE`
False	`false`	`VARIANT_FALSE`	`VI_FALSE`

4.6 Arrays

When arrays are passed as parameters, the size of the array must also be passed. If the array is an in/out parameter, the input size must be passed on input, and the output size must be passed on output.

4.6.1 IVI.NET

IVI.NET interfaces shall pass arrays using .NET arrays. The size is embedded in the array so no additional parameters are needed.

In general, the driver allocates the memory for an IVI.NET output array, although there are some circumstances where the client may allocate array memory. For examples of the latter, refer to section 6, *IMemoryWaveform<T> Interface*, and section 8, *IMemorySpecturm<T> Interface* of *IVI-3.18: .NET Utility Interfaces and Classes*.

4.6.2 IVI-C

When passing an array into a function, each array shall have two parameters: a ViInt32 which is the size of the array and a pointer of appropriate type to the array. The size parameter immediately precedes the pointer in the parameter list.

When retrieving an array from a function, each array shall have three parameters: a ViInt32 which is the size of the array allocated by the caller, a pointer of appropriate type to the array, and finally a pointer to a ViInt32 which the function fills with the actual number of values returned.

Sometimes multiple arrays of exactly the same size are passed into or out of a function. In this case, pointers to these additional arrays shall immediately follow the first array pointer without any additional size parameters.

If the size of the array is the same for all applications, the size parameter may be omitted. The instrument class specification shall specify the size and provide a macro with its value.

Remember, the calling program is responsible for allocating any memory.

Use `Vi<type> <name>[]` for array parameters. The name of the parameter used to pass in the size of the array is `ViInt32 <name>ArraySize`. The name of the parameter used to pass back the actual number of elements put into the array is `ViInt32* <name>ActualSize`.

The function prototype for a routine which has an array as an input would have the form:

```
ViStatus <ClassName>_ConfigureArray(ViSession vi,
                                    ViInt32   AbcArraySize,
                                    ViReal64  Abc[]);
```

The function prototype for a routine which has one array as an output would have the form:

```
ViStatus <ClassName>_FetchArray(ViSession vi,
                                ViInt32   AbcArraySize,
                                ViReal64  Abc[],
                                ViInt32*  AbcActualSize);
```

The function prototype for a routine which has two arrays of the same size as outputs would have the form:

```
ViStatus <ClassName>_FetchArrays(ViSession vi,
                                 ViInt32   AbcArraySize,
                                 ViReal64  AbcDEF[],
                                 ViReal64  AbcXYZ[],
                                 ViInt32*  AbcActualSize);
```

4.6.3 IVI-COM

IVI-COM interfaces shall pass arrays using SAFEARRAY. The size is embedded in the structure so no additional parameters are needed.

4.7 Pointers

Do not pass pointers as values. Addresses are used only when passing

- a parameter by reference,
- an array,
- a string,
- an IVI-COM interface pointer.

For IVI.NET, addresses should not be used in public APIs. In interop situations the System.IntPtr type may be used.

4.8 Sessions

A valid session is created when the Initialize function in an IVI driver is called. Note that it cannot be assumed that sessions are addresses. Sessions are always 32-bit integers, even on 64-bit systems. For IVI-C they are defined as unsigned. For IVI-COM and IVI.NET, they are defined as signed.

4.8.1 IVI.NET

Generally, IVI.NET methods do not have an explicit session parameter. Driver sessions are implied by a driver object. I/O sessions or, in the case of IVI-COM wrappers on IVI-C drivers, underlying IVI-C sessions, may be passed as 32-bit integers if needed.

4.8.2 IVI-C

If one of the parameters of a function in an IVI-C driver is the driver's session, it should be the first parameter and its type shall be ViSession.

The functions init, InitWithOptions, and AttachToExistingCOMSession are all notable exceptions. Specific drivers, particularly wrappers, may have similar type exceptions.

4.8.3 IVI-COM

Generally, IVI-COM methods do not have an explicit session parameter. Driver sessions are implied by a driver object. I/O sessions or, in the case of IVI-COM wrappers on IVI-C drivers, underlying IVI-C sessions, may be passed as 32-bit integers if needed.

4.9 Return Values

4.9.1 IVI.NET

Methods that return data that can be conveniently represented by a single .NET type should return that data as the method's return value. If the data is best represented in the form of multiple types, consider combining the data into a struct and returning the struct. Out and ref parameters may be used, but should be considered a last resort.

In some cases, the results of a method do not need to be returned by the method and can be saved for later retrieval using a property or a different method.

4.9.2 IVI-COM

Many methods have a single out parameter. This parameter should also be declared as the retval for the function. That is, the attributes for the parameter are [out, retval]. Declaring a retval parameter allows the user to set a variable to the value of that parameter using an assignment operator. Declaring a parameter to be a retval also avoids the problem described in Section 6.2, *Use of SAFEARRAY as a Property*~~*Use of SAFEARRAY as a Property*~~.

Formatted: Font: Italic

If a method has multiple out parameters, typically none of the parameters is declared as the retval.

5. Version Control

IVI provides interoperability of drivers from multiple vendors released at various times and without coordination of release schedules between the vendors. At the same time, IVI must reserve the ability to revise interfaces in the shared components.

Versioning Objective. In principle, IVI versioning is designed to make it possible to create an application that uses different vendors' drivers created with different versions of the shared components provided by the IVI Foundation without requiring the application to adopt a special internal architecture to accommodate the version changes. This objective is key to the core value proposition of IVI instrument drivers.

The IVI Foundation may publish revised versions of the instrument class specifications. As IVI drivers are written, the IVI Foundation may discover areas to improve the specifications including adding additional capability groups. These new versions shall be done in a manner which does not make existing applications inoperable.

5.1 IVI-COM Interface Versioning

The IVI Foundation shall create a new version of any standard IVI-COM interface that changes in any of the following ways:

- The interface name is changed.
- The interface inheritance is changed.
- A method or property is added or deleted, or the spelling of a method or property name is changed.
- The parameter list of a method or property or the data type of any parameter is changed.
- Any attribute of any element of the interface is changed.
- For enumerations,
    - An enumeration is added or deleted, or the spelling of an enumeration value name or tag is changed.
    - For IVI-COM, an enumeration member is added or deleted, or the integer value associated with an enumeration value name is changed.
    - Any IDL attribute of any element of an enumeration is changed.

Each interface shall have a base name that is version independent. The first version shall use this name without modification. For each subsequent version, the base name shall have an integer appended, starting with "2" and incrementing by 1. For example, the first version of the IviScope's trigger interface is named IIviScopeTrigger. The second published version of this interface would be named IIviScopeTrigger2, the third would be named IIviScopeTrigger3, and so on.

Whenever a new version of an IVI-COM interface is created, it shall be assigned a new IID.

5.2 IVI-C Interface Versioning

The IVI Foundation shall maintain backwards compatibility when modifying the IVI-C interface defined in a instrument class specification. The IVI Foundation may extend an IVI-C interface, but not modify the existing elements. Modifications that shall be avoided include:

- Changes to existing function prototypes
- For an existing attribute,
    - Changes to its data type
    - Changes to its association with a repeated capability
    - Removing read or write access
- Changes to existing attribute ID constant names and values
- Changes to existing value ID constant names and values

- Changes to existing error code constant names and values

The IVI-C interface may be extended by adding new functions, attributes, attribute values, read or write attribute access, and error codes.

5.3 IVI.NET Versioning

There are two primary .NET versioning strategies for .NET assemblies. Side-by-side installation allows multiple versions of a .NET assembly to be installed side-by-side (e.g. at the same time). Publisher policy files direct references from older versions of an assembly to a newer version of the assembly, and the newer version of the assembly must be backwards compatible with the older versions.

- The IVI.NET Shared Component assemblies will be versioned using a combination of side-by-side installation and policy files when at all possible.[1]
- It is a strong recommendation that IVI.NET drivers be versioned using the same strategy whenever possible as well.

5.3.1 IVI.NET Shared Components

As mentioned above, IVI versioning is designed to make it possible to create an application that uses different vendors' drivers created with different versions of the shared components without requiring the application to adopt a special internal architecture to accommodate the version changes.

Note that the versioning style described in this section does not cover all of the possible ways in which the IVI Shared Components could change from version to version, but it does describe most of the situations that are distinctive to IVI.NET.

5.3.1.1 Versioning with Policy Files

In order to meet the Versioning Objective documented above, the IVI.NET shared components will revise assemblies so that the new version of an assembly continues to provide support for older versions of the IVI interfaces (and other APIs such as exception and class APIs), and then also provide publisher policy files to redirect references from older versions of the assembly to the newer version.[2] (When the term "policy file" is used in this document without qualification, it refers to publisher policy files.)

Using side-by-side versioning without policy files for shared component versioning violates this principle.

- User code that references shared components data types would be exposed to different versions of the same shared component data types.
- An application that used multiple IVI drivers would not be able to simultaneously reference or use drivers that referenced different versions of the IVI.NET Shared Components without taking measures that violate the versioning principle (such as isolating the calls to drivers that use different versions of the shared components into separate DLLs).

---

[1] Side-by-side versioning without policy files is only absolutely required when the target .NET Framework of the assemblies change, and the change results in using a version of the .NET Common Language Runtime (CLR) that is not compatible with the previous version. The IVI Foundation will only make changes for this reason when the current target .NET Framework version becomes unsupported. Massive changes to the IVI APIs could also trigger such a change, but this would likely be interpreted as a completely new set of APIs, and we do not anticipate changes on this scale.

[2] The following are relevant observations about .NET, and are not within the control of the IVI Foundation:

- .NET requires exactly one publisher policy file for each old major/minor version when using policy files to version up. The old major and minor version numbers are part of the policy file name.
- Adding methods or properties to an interface will break components built against the old interface, because the new method/property will not be implemented by the component.

Using publisher policy files implies that the assemblies continue to provide the older versions of the interfaces along with new ones. If assemblies do not continue to support older versions, the versioning principle is also violated.

- If an application uses Driver A built with an older version of the shared components and Driver B built with a newer version of the shared components that revises an Interface that Driver A uses, Driver A would break when the shared components are loaded, because the older version of the interface would not be available.

5.3.1.2 Maintaining Software Configurations

Some programs that use IVI drivers are rigorously qualified with a given software configuration, and once qualified, are expected to build and run against that exact configuration. Installing publisher policy files that redirect assembly references to new versions of an assembly might violate this expectation.

To accommodate users who need to strictly control their software configuration, multiple versions of the IVI.NET Shared Components can be installed concurrently.

Ordinarily these versions will not be used at run time, since any run time reference to them will "policy up" to the newest installed version. However, if a client wishes to continue using an older version of an assembly, an application configuration file (probably most common) or machine configuration file may be created that maintains references to the older version. IVI.NET driver vendors should be prepared to support customers who need to use older versions of an assembly.

When developing code, references to the specific version required for the application can be added to a project, and will be used consistently after that point for editing and building the project, as long as that specific version is installed. Once the IVI Foundation has released a version of the IVI.NET Shared Components, it will be available on the web site indefinitely.

5.3.1.3 Versioning for Policy Files

Any changes to an assembly require that the assembly have a new version number and that the policy file(s) be updated to refer to the new version of the assembly.

In general, the focus on the consistent use of policy files for nearly all versioning tasks means that once an API is published, it needs to be available indefinitely - for as long as policy files are used to redirect references from older assembly versions to newer assembly versions. APIs include interfaces, classes, enumerations, and events.

5.3.1.4 Naming New Versions of .NET Types

Each .NET type declared in an IVI.NET Shared Components assembly shall have a base name that is version independent. The first version shall use this name without modification. For each subsequent version, the base name shall have an integer appended, starting with "2" and incrementing by 1.

For example, the first version of the Ivi.Scope trigger interface is named IIviScopeTrigger. The second published version of this interface would be named IIviScopeTrigger2, the third would be named IIviScopeTrigger3, and so on.

5.3.1.5 Versioning Enumerations

Enumerations shall not be deleted or renamed. New enumerations may be added.

Enumeration members shall not be deleted or renamed. The numeric value of an existing enumeration member shall not be changed, since it is the same as deleting the member with the old value and adding the member with the new value.

If an enumeration member must be deleted or renamed, or existing numeric values must be changed, a new enumeration shall be created. The new enumeration shall be named as described in section 5.3.1.4, *Naming New Versions of .NET Types*~~*Naming New Versions of .NET Types*~~. Since the intent is to allow users to migrate to the new enumeration with a minimum of change, enumeration members that are common to both the old and new versions of the enumeration shall have the same spelling and numeric values.

Formatted: Font: Italic

Enumeration members may be added if they are added in a way that does not cause the value of any existing members to change. For enumerations where numeric values are not specified, this means that new members shall only be added to the end of the enumeration.

EXAMPLE

If an enumeration named "TriggerSource" includes a trigger source called "TriggerLine0" that is no longer needed, the following versioning strategy is used:

- The "TriggerSource" enumeration is not modified.
- A new enumeration is created, named "TriggerSource2", that includes all of the old members except for "TriggerLine0".
- The new members match the old members in spelling and value. If the "TriggerSource" enumeration uses default values, values may need to specified for "TriggerSource2" if the removal of the "TriggerLine0" member leaves a gap in the values.

5.3.1.6 Versioning Interfaces

Interfaces shall not be deleted or renamed. New interfaces may be added.

Interface members shall not be added or deleted, and the signatures of existing members shall not be changed in any way, including:

- The return type of an existing member shall not be changed.
- If a member has parameters, parameters shall not be added or deleted, and
- parameter names and types shall not be changed.

If an interface member must be added, deleted, or changed in some way, a new interface shall be created. The new interface shall be named as described in section 5.3.1.4, *Naming New Versions of .NET Types*~~*Naming New Versions of .NET Types*~~. Since the intent is to allow users to migrate to the new interface with a minimum of change, interface members that are common to both the old and new versions of the interface shall have the same signatures.

Formatted: Font: Italic

If a new interface is created to version an older interface, it shall be created in one of two ways:

- The new interface is cloned from the older interface, and then modified within the constraints listed above. This technique always works.
- The new interface derives from the older interface. However, derivation has pitfalls - interface reference properties may need to return references to newer interfaces, for example, and members from the derived interface may not be deleted. To accommodate these situations, the following process is followed when deriving a new interface from an older one.
    - New members are added to the new interface.
    - Where a new method or property that matches an older method or property except for the return type, the new method or property uses the "new" modifier to hide the older one. (This addresses the issue with interface reference properties.)
    - Obsolete members are tagged with the "Obsolete" attribute in the older interface. The "Obsolete" attribute is constructed so that trying to build code that uses the member generates a build warning or error (at the discretion of the IVI.NET WG). Note that the member is still available and does not generate a runtime error for binaries built against the older version of the interface.

EXAMPLE

For example, assume an interface named “IWaveform2”. “ IWaveform2” is missing a property named “Center”, and the “Configure” method is missing the corresponding “center” parameter. In addition, an interface reference property called “Display” is modified to return a reference to the “IWaveformDisplay2” interface instead of the “IWaveformDisplay” interface. The following versioning strategy is used:

- The “IWaveform2” interface is not modified.
- A new interface is created, named “IWaveform3”, that includes the following members
  - All of the old members with the same signatures. Overloads to the Configure method that make no sense without a “center” parameter may be omitted..
  - A new overload of the “Configure” method with the “center” parameter.
  - The new “Center” property.
  - The modified “Display” property.
- Where new members match the old members, the signatures also match.

EXAMPLE - CLONING CODE

```
public interface IWaveform
{
   public void Configure(Int32[] data, Int32 span);
   // Other methods …
   public Int32[] Data { get; }
   public IWaveformDisplay Display { get; }
   public Int32 Span ( get; }
   // Other properties …
}

public interface IWaveform2
{
   public void Configure(Int32[] data, Int32 center, Int32 span);
   // Other methods …
   public Int32[] Data { get; }
   public Int32 Center ( get; }
   public IWaveformDisplay2 Display { get; }
   public Int32 Span ( get; }
   // Other properties …
}
```

EXAMPLE - DERIVATION CODE

```
public interface IWaveform
{
   [Obsolete, false]  // This generates an warning on build.
   public void Configure(Int32[] data, Int32 span);
   // Other methods …
   public Int32[] Data { get; }
   public IWaveformDisplay Display { get; }
   public Int32 Span ( get; }
   // Other properties …
}

public interface IWaveform2 : IWaveform
{
   public void Configure(Int32[] data, Int32 center, Int32 span);
   public Int32 Center ( get; }
   new public IWaveformDisplay2 Display { get; }  // Use new to hide the
                                                  // old Display property.
```

}

5.3.1.7 Versioning Classes

Classes shall not be deleted or renamed. New classes may be added.

Class members shall not be deleted, and the signatures of existing members shall not be changed in any way, including:

- The return type of an existing member shall not be changed.
- If a member has parameters, parameters shall not be added or deleted, and
- parameter names and types shall not be changed.
- For members derived from interfaces, implementation shall not be changed from explicit to implicit or vice versa.

If a class member must be deleted, or changed in some way, a new class shall be created. The new class shall be named as described in section 5.3.1.4, *Naming New Versions of .NET Types*~~*Naming New Versions of .NET Types*~~. Since the intent is to allow users to migrate to the new class with a minimum of change, class members that are common to both the old and new versions of the class shall have the same signatures.

Formatted: Font: Italic

New class members may be added to existing classes.

In general, the range of behavioral changes that don't affect the class API is fairly broad, and the decision to implement a new class or not in response to a particular behavioral change is left to the discretion of the IVI.NET Working Group.

The only significant difference between versioning interfaces and classes is the way that new members are treated. Therefore, the techniques used to version classes are nearly the same as those used to version interfaces, with the exception that if the only change to a class is to add new members, the new members may be added to the existing class.

Exceptions are just a specialization of a class, and are versioned like classes.

5.3.1.8 Other Considerations

All API changes shall be considered major or minor changes. The decision is left to the discretion of the IVI.NET Working Group.

Behavioral changes may be considered major, minor, or build changes. The decision is left to the discretion of the IVI.NET Working Group.

XML comments may be changed freely. The changes are considered build changes.

When any of the IVI.NET Shared Component assemblies is changed, all of the assembly versions shall be changed to the identical version. If a new version of the IVI.NET Shared Components installer includes a mix of major, minor, and build level revisions, the highest version needed to correctly version the individual assemblies shall be the version used for all of the assemblies. Note that this implies that the policy file(s) for each assembly must be updated every time any of the assemblies change.

5.3.2 IVI.NET Shared Components Installer

The installer version major/minor version shall be the same as the assembly versions. In some cases where the only changes are to the installer, the installer build number may be greater than the assembly build numbers.

5.3.3 IVI.NET Specific Drivers

It is recommended that IVI.NET specific drivers use the versioning style for the IVI.NET Shared Components, except that one of the restrictions on interface versioning may be loosened. In particular, interface members may be added to an interface without creating a new version of the interface if the vendor does not support any other interface implementations outside of the assembly.

6. IVI-COM IDL Style

One of the responsibilities of a working group developing an instrument class specification is to create the IVI-COM IDL for the instrument class. An IVI-COM specific driver writer also produces COM IDL. These style considerations apply to both.

6.1 Use of out vs. in,out

Microsoft Visual Basic has a defect which causes a memory leak if a parameter is declared to be only [out]. It does not free the memory allocated by the server. To work around this defect, no parameters in IVI IDL shall every be declared as just [out]. They shall always be either [out, retval] or [in, out].

If a parameter is declared as `[in, out]` when it truly is just an output, the instrument class specification shall state so.

This defect is known to exist up through version 6.0. If the defect is repaired in future revisions of Visual Basic, this requirement may be amended.

6.2 Use of SAFEARRAY as a Property

Properties shall never be a SAFEARRAY of any type since Microsoft Visual Basic does not gracefully handle SAFEARRAYS as properties.

6.3 Help Strings

The IDL file for an instrument class specification shall contain the following help string.

"<*ClassName*> <*Revision*> Type Library"

Where <*Revision*> is the revision of the instrument class specification.

This help string appears prominently in various tools that browse for available type libraries. The IDL file shall contain help context IDs and help strings for every interface, method, property, and enumeration.

7. Controlling Automatic Setting Attributes

All of the requirements in this section and its sub-sections apply to class specifications approved after January 1, 2010, and to instrument specific interfaces for drivers released after January 1, 2010. If an instrument specific driver mirrors a class API that does not observe the requirements described in this section, the instrument specific API may mirror the class API.

Instruments and drivers contain algorithms which automatically adjust settings based on other settings or characteristics of input signals. Typically, these algorithms can be enabled and disabled. For example, a DMM can adjust its voltage range based on the amplitude of the input signal. The setting which enables or disables the algorithm is separate from the setting which represents the actual value. For example, even with autoranging on, the DMM range has an actual value. Another example is automatic coupling of frequency span to resolution bandwidth, video bandwidth, and sweep time in a spectrum analyzer.

Automatic settings shall be reflected in IVI drivers using two attributes – the primary attribute and the automatic setting attribute. These attributes shall conform to the following description:

- The primary attribute corresponds to an instrument setting.
    - The primary attribute is read/write.
    - The primary attribute may be manually set to one of a variety of values, which in turn sets the instrument to the corresponding value.
    - If the primary attribute is manually set, the automatic setting attribute is set to Off or False.
    - When read, the primary attribute always returns the actual value from the instrument. The actual value may be set by the API or, if the automatic setting attribute is On or Once, determined by the instrument's automatic algorithm.
- The automatic setting attribute determines whether or not the instrument sets the value of the primary attribute automatically.
    - The automatic setting attribute is read/write.
    - If the automatic setting attribute includes On (True) and Off (False) choices, it is a 2-state automatic setting attribute, and is implemented as a Boolean
    - If the automatic setting attribute also include a value that indicates that the instrument should set the value of the first attribute automatically one time only ("Once"),, it is a 3-state automatic setting attribute, and is implemented as type Ivi.Driver.Auto.
    - If the automatic setting is to be turned off, the instrument shall revert to the manual setting mode and retain the current value for the primary attribute. If the instrument does not support this behavior, an exception should be thrown (IVI.NET) or an error returned (IVI-C and IVI-COM).
    - The name of the automatic setting property shall be constructed by concatenating the name of the primary attribute with the string "Auto". For example, if the name of the primary property is "Resolution Bandwidth", the name of the corresponding automatic setting attribute would be "Resolution Bandwidth Auto". If placing Auto first is a recognized domain convention (for example, Auto Range or Auto Zero), then the Auto may be placed before the primary attribute name.
- The semantics are the same across the IVI.NET, IVI-C, and IVI-COM APIs.

An instrument class specification shall not use special, out-of-range values for primary attributes. For example, for a primary attribute called Range, do not define a special value of –1 for Range that turns autoranging on. Instead use two attributes: Range and Auto Range, as described above.

7.1 Functions and Automatic Settings

7.1.1 IVI.NET

For methods that include the primary attribute as a parameter, IVI.NET shall provide overloads that allow the function to set either the primary attribute or the automatic setting parameter (but not both). Functions that include the primary parameter shall set the automatic setting attribute to Off or False.

For example:

```
void ConfigureRange(Ivi.Driver.Auto autoRange);
void ConfigureRange(Double range);
```

The user may call the overload with the auto parameter and specify the automatic setting is to be turned off. In this case, the instrument shall revert to the manual setting mode and retain the current value. If the instrument does not support this behavior, an exception should be thrown (IVI.NET) or an error returned (IVI-C and IVI-COM).

Functions may need to provide overloads for more than one automatic setting. In this case, there needs to be an overload for each permutation of automatic setting parameters. If there is one automatic setting, there are two overloads. If there are two automatic settings, there are four overloads. If there are three automatic settings, there are nine overloads. The overload approach works well for 1-3 automatic settings, and would be bulky but workable for four. For instances of functions with four or more automatic settings, other approaches to overloads may be considered.

7.1.2 IVI-C & IVI-COM

For methods that include the primary attribute as a parameter, IVI-C and IVI-COM shall include parameters for both the primary attribute and the automatic setting attribute.

For example:

```
ViStatus IviDmm_ConfigureRange(ViSession Vi,
                               ViReal64 Range
                               ViInt32 AutoRange);
HRESULT ConfigureRange([in] DOUBLE Range
                       [in] IviDmmRangeAutoEnum AutoRange);
```

If the auto parameter is set to False or Auto Off, the function uses the value manually passed in by the primary parameter. If the auto parameter is set to True, Auto On or Auto Once, the primary parameter is ignored and the instrument is set to automatically determine the value of the primary attribute.

8. Time Representation

8.1 Absolute Time

Instruments sometimes provide time stamps for measured data. In these cases it is necessary for drivers to include a means of setting and retrieving the absolute time. Refer to Section 6, *Absolute Time* of *IVI-3.3: Standard Cross Class Capabilities* for a description of functions which are used in instrument classes for this purpose.

These methods may be needed in a driver's instrument specific API if, for example, the instrument class(es) implemented by the driver does not include them, or if the driver is a custom driver. In such cases, the placement of these methods within the hierarchy of an IVI driver is at the discretion of the driver developer, but these methods should be located with other instrument specific, system-related attributes and methods in the instrument specific portion of the driver hierarchy unless there is a compelling reason to locate them elsewhere. If methods that provide this functionality are needed in an instrument specific API, the method signatures shall match the signatures in the section of IVI-3.3 referenced in the preceding paragraph.

8.2 General Time Parameters

For IVI.NET, all time parameters shall be PrecisionTimeSpan or PrecisionDateTime. The PrecisionTimeSpan and PrecisionDateTime classes are defined in *IVI-3.18: IVI.NET Utility Classes and Interfaces*.

For IVI-C and IVI-COM, all time parameters, except time out parameters shall be real and have units of seconds. This rule is consistent with Section 4.2, *Reals~~Reals~~*. Timeout parameters, however, shall be an integer with units of milliseconds.

Formatted: Font: Italic

8.3 TimeOut Parameters

A timeout parameter indicates the maximum amount of time the instrument or driver should wait for an event before returning control to the client program. There are two special cases requiring special defined values:

- The driver should return immediately.
- The driver should wait indefinitely.

Defined values are placed in tables in the section that defines the method.

8.3.1 IVI.NET TimeOut Parameters

IVI.NET timeout parameters shall be named `maxTime`, and shall be of type TimeSpan. Note that units are implicit in the definition of the TimeSpan class, and do not need to be specified.

If `maxTime` is `TimeSpan.Zero`, the driver should return immediately. If `maxTime` is `TimeSpan.MaxValue`, the driver should wait indefinitely.

Place a table of this form in the method section.

Defined Values for the maxTime Parameter (.NET)

Name	*Description*		
		Language	*Identifier*
Zero	The method returns immediately. If no valid measurement value exists, the method returns an error.		
		.NET	`TimeSpan.Zero`
MaxValue	The method waits indefinitely for the measurement to complete.		
		.NET	`TimeSpan.MaxValue`

8.3.2 IVI-C and IVI-COM TimeOut Parameters

IVI-C and IVI-COM, timeout parameters shall be named MaxTimeMilliseconds, and shall be 32-bit integers. Units shall be milliseconds.

When an instrument class specification defines a value for timeout parameter, its IVI-C name shall be of the form `<CLASS_NAME>_VAL_MAX_TIME_<WORD>` and its IVI-COM name shall be of the form `<ClassName>MaxTime<Word>`. The two most common values are “Immediate” and “Infinite”.

Place a table of this form in the function section. Note that this table should not be combined with the IVI.NET table, as the parameter names are different.

Defined Values for the MaxTimeMilliseconds Parameter (C and COM)

Name	*Description*		
		Language	*Identifier*
Max Time Immediate	The function returns immediately. If the operation had not already completed, the function returns an error.		
		C	`<CLASS_NAME>_VAL_MAX_TIME_IMMEDIATE`
		COM	`<ClassName>MaxTimeImmediate`
Max Time Infinite	The function waits indefinitely for the operation to complete.		
		C	`<CLASS_NAME>_VAL_MAX_TIME_INIFINITE`
		COM	`<ClassName>MaxTimeInfinite`

Place a table of this form in the Function Parameter Value Definitions section.

Value Name	*Language*	*Identifier*	*Actual Value*
Max Time Immediate	C	<CLASS_NAME>_VAL_MAX_TIME_IMMEDIATE	0
	COM	<ClassName>MaxTimeImmediate	0
Max Time Infinite	C	<CLASS_NAME>_VAL_MAX_TIME_INIFINITE	0xFFFFFFFFUL
	COM	<ClassName>MaxTimeInfinite	0xFFFFFFFFUL

9. Units

Most likely every parameter or attribute which is real, and even some integers, has a unit associated with it. The instrument class specification shall specify what that unit is. Instrument classes may provide mechanisms to change the unit of a value.

Avoid units with a multiplier. Stick with the base unit. The unit meter is preferred over kilometer.

The unit should, in general, be an SI primary unit. IEEE 488.2 Table 7-1 *<suffix unit> Elements* provides a list of commonly used units.

10. Disable

The instrument class specification shall include a statement about how IVI specific drivers implement the Disable function for that instrument class. The statement shall appear in Section 3.1.1. Refer to Section 6.4, *Disable* in *IVI-3.2: Inherent Capabilities Specification* for a general description of the function.

11. Completion Codes and Error Messages

IVI-3.2: Inherent Capabilities Specification describes IVI-C and IVI-COM completion codes, IVI.NET exceptions, and error messages which are applicable to most of the error conditions in IVI drivers and other components. They cannot, however, cover all cases. Instrument class specification writers shall use the codes and exceptions in IVI-3.2 as appropriate. When those codes or exceptions are not sufficient, the instrument class may define additional codes and/or exceptions. Along with each code or exception, the instrument class specification shall define an error message which the IVI driver returns in Error Message, in IVI-C, in ErrorInfo in IVI-COM, or in the exception message in IVI.NET.

11.1 IVI.NET

The IVI.NET name for an exception shall end with "Exception". The name shall begin with a series of words, in Pascal casing. This portion of the name shall use exactly the same words as the error-specific portion of the IVI-C and IVI-COM error constant name. All IVI.NET exceptions shall derive from System.Exception.

In many cases, IVI.NET drivers will be able to use exceptions defined by the .NET Framework, and in general this is encouraged. In these cases, the IVI specification or specific driver may still need to define an error message that reflects the specific error condition

IVI.NET warnings are assigned a GUID. If warnings are defined for a class, the class shall implement a static class with a static, read only property of type GUID for each warning. The property name for a warning shall consist of exactly the same words as the warning-specific portion of the IVI-C and IVI-COM warning constant name.

11.2 IVI-C

The C constant name for an error shall begin with:

```
<CLASS_NAME>_ERROR_
```

The C constant name for a warning shall begin with:

```
<CLASS_NAME>_WARN_
```

The remainder of the name shall be a series of words, all in uppercase, separated by underscore characters. This portion of the name shall exactly match the equivalent portion of the IVI-COM constant name.

11.3 IVI-COM

The IVI-COM constant name for an error shall begin with:

```
E_<CLASS_NAME>_
```

The IVI-COM constant name for a warning shall begin with:

```
S_<CLASS_NAME>_
```

The remainder of the name shall be a series of words, all in uppercase, separated by underscore characters. This portion of the name shall exactly match the equivalent portion of the IVI-C constant name.

12. Repeated Capabilities

Many instruments have capabilities which are duplicated. For example, an oscilloscope might have several channels with identical functionality. A power supply may have several outputs.

Several styles are available to provide consistent access to the repeated capabilities.

- Parameter Style
- Collection Style (IVI-COM and IVI.NET)
- Selector Style

12.1 Parameter Style

For a given repeated capability, class APIs may add a parameter to every function or attribute which uses that repeated capability, The parameter is a string parameter that "selects" the name of the repeated capability to be used.

Most of the current class specifications with repeated capabilities use this technique for the IVI-C API. However, for IVI-COM and IVI.NET, this technique is only used in the IviFgen specification.

12.1.1 IVI-C

In IVI-C, attributes are accessed using the Get and Set Attribute functions defined in *IVI-3.2: Inherent Capabilties*. Each of these functions has one parameter named AttributeID to specify which attribute is being accessed and one parameter named RepCapIdentifier which is used only if the attribute is specific to a repeated capability.

This is the preferred technique for IVI-C.

12.1.2 IVI-COM

The IDL used to describe IVI-COM APIs allows the use of parameterized properties. For IVI-COM, properties that are specific to a repeated capability have a parameter that specifies the repeated capability. While the IDL actually reflects the fact that the implementation is done by creating set (put) and get methods, the two methods are tied together by propget and propput attributes.

12.1.3 IVI.NET

IVI.NET syntax does not allow the use of parameterized properties. Instead, Set and Get Attribute methods must be created for each property, and each method must have the repeated capability as a parameter.

12.2 Collection Style (IVI-COM and IVI.NET)

In IVI-COM and IVI.NET, repeated capabilities can be organized into collections. Repeated capabilities can then be selected and manipulated via the collection.

It is not possible to use this approach with IVI-C. Most of the current class specifications with repeated capabilities use this technique for the IVI-COM and IVI.NET APIs.

This is the preferred technique for IVI-COM and IVI.NET, as it is more explicit than selector style.

12.2.1 Base Interfaces (IVI.NET)

In IVI.NET, all repeated capability collections shall derive from IIviRepeatedCapabilityCollection, which includes the Count property and the Item Operators.

In IVI.NET, all classes that represent instances of a repeated capability shall derive from IIviRepeatedCapabilityIdentification, which includes the Name property.

12.3 Selector Style

For a given repeated capability, class APIs may provide a string property or a function with a single string parameter which "selects" a currently active capability by designating the name of the repeated capability to be used. The selected repeated capability remains active until another one is selected.

Do not use the selector technique when the API contains nested repeated capabilities. A nested repeated capability occurs when the user must specify at least two items in a hierarchy to gain access to a particular capability. For example, an API with a window repeated capability where each window has a trace repeated capability contains a nested repeated capability.

This technique is not the first choice for representing repeated capabilities in IVI-C, IVI-COM, or IVI.NET. In some circumstances it is useful – for example, when trying to represent a large number of repeated capability instances in a single selector, or when there is only one instance of the repeated capability for most instruments of a class, but class designers want to accommodate more than one instance. If this technique is used, it should be used consistently across the IVI-C, IVI-COM, and IVI.NET APIs.

12.4 Using the Techniques

The following table shows the attributes and functions that are used to implement each of the techniques described above, in each of the supported IVI APIs.[3] Remember that collections cannot be used in IVI-C.

Technique	API Type	IVI.NET	IVI-C	IVI-COM
Parameter Style	Attributes	<RC> Count	<RC> Count	<RC> Count <RC> Name (Index)
	Functions	Get <RC> Name	Get <RC> Name	
Collection Style	Attributes	<RC> Item <RC> Count <RC> Name	N/A	<RC> Item <RC> Count <RC> Name(Index)
Selector Style	Attributes	<RC> Count Active <RC>	<RC> Count Active <RC>	<RC> Count Active <RC> <RC> Name
	Functions	Get <RC> Name Set Active <RC>	Get <RC> Name	

[3] The table uses the notation <RC> to indicate a repeated capability name. . The instrument class specification specifies the actual name of the repeated capability. The plural form is shown as <RC>s. The instrument class specification uses the proper English plural; it does not blindly add an s. In cases where the repeated capability name should be lowercase, <rc> is used.

12.4.1 Specifying Repeated Capability Attributes and Functions

Most repeated capabilities use parameter style for IVI-C, and collection style for IVI-COM and IVI.NET. This is the most common way to implement repeated capabilities in IVI class specifications. Repeated capabilities should be implemented this way unless there is a compelling reason not to.

A smaller number of repeated capabilities use selector style for IVI-C, IVI-COM, and IVI.NET.[4]

Refer to Section 3, *Repeated Capability Group* in *IVI 3.3: Standard Cross-Class Capabilities Specification*, for a complete description of the attributes and functions needed to implement these two ways of specifying repeated capabilities.

One repeated capability, the IviFgen Channel, uses parameter style for IVI-C, IVI-COM, and IVI.NET. This is not recommended for future classes.

[4] One repeated capability, the IviUpconverter Analog Modulation Source, uses a limited variation that replaces the Active <RC> attribute and Set Active <RC> function with AM Source, FM Source, and PM Source, which allow the client to select one or more sources as the modulating signal.

13. Hierarchies

Instrument class specifications and the resulting IVI drivers present many functions and attributes to the user. In an effort to reduce the perceived complexity, these items are organized into hierarchies. Creating hierarchies with different styles add unneeded confusion to the user. Instrument class specification writers and IVI specific driver developers should follow these guidelines when designing hierarchies.

13.1 C Function Hierarchy

IVI-C drivers include a function panel file that contains and a function hierarchy of the exported functions. A function hierarchy assists the user in finding functions by organizing the functions into categories.

The function panel file format is specified in Section 6, Function Panel File Format, of the VXIplug&play specification VPP-3.3: Instrument Driver Interactive Developer Interface Specification.

Instrument class specifications provide guidelines for organizing functions defined by the instrument class specification. IVI-C specific drivers should follow the hierarchy for the instrument class defined functions and extend the hierarchy to include instrument specific functions.

General principles to follow when creating a function hierarchy:

1. The Initialize, Initialize With Options, and Close functions should appear at the top level of the function tree hierarchy.

2. Functions should be organized into class types according to usage. Common categories include *Application*, *Configuration*, *Measurement*, and *Action/Status*.

    - *Application* functions are created from a lower-level set of IVI driver functions. Typically, these are provided with IVI specific drivers, but are not defined within an instrument class specification.

    - *Configuration* functions change the state of instrument settings.

    - *Action/Status* functions include two types of functions. Action functions, such as Initiate and Abort, cause the instrument to initiate or terminate test and measurement operations. These operations can include arming the triggering system or generating a stimulus. These functions are different from the configuration functions because they do not change the instrument settings, but only order the instrument to carry out an action based on its current configuration. Status functions obtain the current status of the instrument or the status of pending operations.

    - *Measurement* functions capture single-point and multi-point measurement data. Functions that initiate and transfer data to and from the instrument appear within the *Measurement* category. If a *Measurements* category exists for a class, low-level action functions should not appear under a separate *Action/Status* category. Instead, low-level action and measurement functions, such as Initiate, Fetch, Abort, and Sent Software Trigger should appear in a *Low-Level Measurements* sub-category below the *Measurements* category. If the instrument operation includes more than scalar measurements, this category might be more appropriately named. For example, the IviScope specification defines a *Waveform Acquisition* category that includes both waveform and measurement functions.

    - *Utility* functions perform operations that are auxiliary to the operation of the instrument. These utility functions include inherent functions specified by IVI 3.2: Inherent

Capabilities Specifications. Other functions that appear under the *Utility* category include functions for reading/writing to the instrument and calibration functions.

3. Within a given category, place the functions in the natural order in which they will be used. Functions that should be called first appear higher in the hierarchy than those that are called later. For example, Initialize appears before Close.

4. If a category contains many functions, divide the category into sub-categories. For example, the Configuration category could contain a sub-category called Trigger Configuration.

5. All attribute accessor functions should appear in a *Set/Get Attribute* sub-category of the *Configuration* category.

13.1.1 Sample Function Hierarchy

A sample function hierarchy appears in **Table 13-1**. IVI Class and IVI specific drivers should follow this hierarchy as closely as possible.

Table 13-1 Prefix Function Hierarchy

Name or Class	Function Name
Initialize	*<ClassName>*_init
Initialize with Options	*<ClassName>*_InitWithOptions
Configuration…	
<Configure sub-categories and functions>	
Set/Get Attribute…	
Set Attribute…	*<ClassName>*_SetAttribute<type>
Get Attribute…	*<ClassName>*_GetAttribute<type>
Measurement…	
Read <Measurement>	*<ClassName>*_Read<Measurement>
<Measurement sub-categories and functions>	
Low-Level Measurement…	
Initiate <Measurement>	*<ClassName>*_Initiate<Measurement>
<Measurement> Status	*<ClassName>*_<Measurement>Status
Fetch <Measurement>	*<ClassName>*_Fetch<Measurement>
Abort	*<ClassName>*_Abort
Utility…	
Reset	*<ClassName>*_reset
ResetWithDefaults	*<ClassName>*_ResetWithDefaults
Disable	*<ClassName>*_Disable
Self-Test	*<ClassName>*_self_test
Revision Query	*<ClassName>*_revision_query
Error-Query	*<ClassName>*_error_query
Error Message	*<ClassName>*_error_message
Get Next Coercion Record	*<ClassName>*_GetNextCoercionRecord

Table 13-1 Prefix Function Hierarchy

Name or Class	Function Name
Get Next Interchange Warning	*<ClassName>*_GetNextInterchangeWarning
Clear Interchange Warnings	*<ClassName>*_ClearInterchangeWarnings
Reset Interchange Check	*<ClassName>*_ResetInterchangeCheck
Invalidate All Attributes	*<ClassName>*_InvalidateAllAttributes
Error Info…	
Get Error	*<ClassName>*_GetError
Clear Error	*<ClassName>*_ClearError
Locking…	
Lock Session	*<ClassName>*_LockSession
Unlock Session	*<ClassName>*_UnlockSession
Close	*<ClassName>*_close

13.2 C Attribute Hierarchy

IVI-C drivers include a sub file that contains and an attribute hierarchy of the exported attributes. An attribute hierarchy assists the user in finding attributes by organizing attributes into logical groups.

The sub file format is specified in Section 7, Function Panel Sub File Format, of the VXIplug&play specification VPP-3.3: Instrument Driver Interactive Developer Interface Specification.

Instrument class specifications provide guidelines for organizing attributes defined by the instrument class specification. IVI-C specific drivers should follow the hierarchy for the instrument class defined functions and extend the hierarchy to include instrument specific functions.

General principles to follow when creating an attribute hierarchy:

1. The Inherent IVI Attributes category appears before instrument class-defined attribute categories.

2. Place attributes in a natural order in which they will be set. Attributes that should be set first appear higher in the hierarchy than those that are set later. For example, in the IviDmm class, Range should always be set before Resolution. Therefore, the Range attribute appears before Resolution in the hierarchy.

3. Attributes should be grouped by extension capability groups.

4. Attributes specific to a particular trigger type should be grouped together. For example, in the IviScope class, each trigger type appears as a level 2 category. Each trigger category includes the attributes unique to that trigger type.

5. Attributes that are necessary to fully specify the state of the instrument appear higher in the hierarchy than those that are optional or included in an extension capability group. For example, for the IviDmm class-specification, the Function and Range attributes are at level 2 while the Thermocouple Type attribute, which is used only when taking temperature measurements, appears at level 3.

13.2.1 Sample Attribute Hierarchy

A sample attribute hierarchy appears in **Table 13-2**. Insofar as it is practical, IVI Class and IVI specific drivers should follow this hierarchy.

Table 13-2. <*ClassName*> C Attributes Hierarchy

Category or Generic Attribute Name	C Defined Constant
Inherent IVI Attributes	
User Options	
Range Check	*<CLASS_NAME>*_ATTR_RANGE_CHECK
Query Instrument Status	*<CLASS_NAME>*_ATTR_QUERY_INSTRUMENT_STATUS
Cache	*<CLASS_NAME>*_ATTR_CACHE
Simulate	*<CLASS_NAME>*_ATTR_SIMULATE
Record Value Coercions	*<CLASS_NAME>*_ATTR_RECORD_COERCIONS
Interchange Check	*<CLASS_NAME>*_ATTR_INTERCHANGE_CHECK
Class Driver Identification	
Class Driver Description	*<CLASS_NAME>*_ATTR_CLASS_DRIVER_DESCRIPTION
Class Driver Prefix	*<CLASS_NAME>*_ATTR_CLASS_DRIVER_PREFIX
Class Driver Vendor	*<CLASS_NAME>*_ATTR_CLASS_DRIVER_VENDOR
Class Driver Revision	*<CLASS_NAME>*_ATTR_CLASS_DRIVER_REVISION
Class Driver Class Spec Major Version	*<CLASS_NAME>*_ATTR_ CLASS_DRIVER_CLASS_SPEC_MAJOR_VERSION
Class Driver Class Spec Minor Version	*<CLASS_NAME>*_ATTR_ CLASS_DRIVER_CLASS_SPEC_MINOR_VERSION
Driver Identification	
Specific Driver Description	*<CLASS_NAME>*_ATTR_SPECIFIC_DRIVER_DESCRIPTIO N
Specific Driver Prefix	*<CLASS_NAME>*_ATTR_SPECIFIC_DRIVER_PREFIX
Specific Driver Locator	*<CLASS_NAME>*_ATTR_SPECIFIC_DRIVER_LOCATOR
Specific Driver Vendor	*<CLASS_NAME>*_ATTR_SPECIFIC_DRIVER_VENDOR
Specific Driver Revision	*<CLASS_NAME>*_ATTR_SPECIFIC_DRIVER_REVISION
Specific Driver Class Spec Major Version	*<CLASS_NAME>*_ATTR_ SPECIFIC_DRIVER_CLASS_SPEC_MAJOR_VERSION
Specific Driver Class Spec Minor Version	*<CLASS_NAME>*_ATTR_ SPECIFIC_DRIVER_CLASS_SPEC_MINOR_VERSION
Driver Capabilities	
Supported Instrument Models	*<CLASS_NAME>*_ATTR_SUPPORTED_INSTRUMENT_MODEL S
Class Group Capabilities	*<CLASS_NAME>*_ATTR_GROUP_CAPABILITIES
Instrument Identification	

Table 13-2. <*ClassName*> C Attributes Hierarchy

Category or Generic Attribute Name	C Defined Constant
Instrument Manufacturer	<*CLASS_NAME*>_ATTR_INSTRUMENT_MANUFACTURER
Instrument Model	<*CLASS_NAME*>_ATTR_INSTRUMENT_MODEL
Instrument Firmware Revision	<*CLASS_NAME*>_ATTR_INSTRUMENT_FIRMWARE_REVISION
Advanced Session Information	
Logical Name	<*CLASS_NAME*>_ATTR_LOGICAL_NAME
I/O Resource Descriptor	<*CLASS_NAME*>_ATTR_IO_RESOURCE_DESCRIPTOR
Driver Setup	<*CLASS_NAME*>_ATTR_DRIVER_SETUP
Basic Operation	
<Base Capability Attributes and sub-categories>	
Trigger	
<Trigger attributes and sub-categories>	
<*Other Capability Groups/Operations*>	
<Attributes and sub-categories>	

13.3 IVI-COM and IVI.NET Interface Hierarchy

In defining some IVI-COM and IVI.NET hierarchies, these general principles were discovered.

1. Use nouns for interface names, not verbs. These nouns may be instrument subsystem names. Extension group names may also be good choices.

2. Create large interfaces only when the items are all read-only or seldom used. A user's level of discomfort seems to be around fifteen. Never exceed 30 as this many items requires scrolling.

3. Create interfaces with at least three items. One interface with one or two items is acceptable, but rare.

4. Separate multiple, mutually exclusive options into separate sub-interfaces. For example, with scope triggers only one of the several trigger options can be used at any one time so they were split into their own sub-interfaces.

5. Whenever possible, use collection style to represent repeated capabilities. If a design seems to indicate that selector style or parameter style are a better choice, try re-designing to allow the use of collection style.

6. Put only the items that are truly repeated in the repeated capability interface, the interface with the singular name, to minimize redundancy.

7. Avoid adding other methods and properties to the collections interfaces (the interface with the plural name). Any additional methods and properties beyond the standard three should apply to all the items in the collection.

8. Separate or combine optional elements depending on factors such as the number of methods and properties, the depth of hierarchy, etc. Examples are multi-point trigger on DMM, which is separated, and AMInternal in the Fgen, which is not separated.

9. Represent cross-class capabilities in the same way.

10. Keep in mind the potential for future evolution of the design and potential ways that specific instruments might extend the instrument class capabilities when designing the hierarchy.

11. Keep logical siblings at the same depth in the hierarchy. Either put the elements in one interface or in sibling interfaces.

12. Hide seldom used, read-only item lower in the hierarchy, particularly if it serves to highlight more commonly used properties and methods.

13. Avoid adding properties and methods to the root. Instrument class specifications should generally add only interface reference properties to the root. Some instruments classes, however, have a small number of methods and properties which are so fundamental to the operation of the instrument class that the only logical place in the hierarchy is at the root. Range, function, and resolution for the Dmm are examples.

14. Maintain parallelism between like elements with respect to the level in the hierarchy and interface names.

These principles may lead to conflicting results. Instrument class writers should use them, but with caution and judgment. These principles have no known order or priority.

14. Synchronization

Users must often synchronize a program's execution with the operation of instruments in a system. Instrument class specifications can help users by supplying methods which assist in synchronization. Using similar notation, behavior, and parameters is a matter of style and not functionality.

When instrument class specification writers identify an operation which can take an appreciable amount of time to complete, they should consider adding two functions to the capability group associated with the operation.

14.1 Non-blocking

This function should be of the form:

```
Is<operation complete> (ViBoolean* Done)
```

The words in <operation complete> are related to the operation being performed. For example, Settled.

The function returns rapidly. The parameter, Done, is false if the operation is still being performed and it is true if the operation has completed.

14.2 Blocking

For IVI-C this function should be of the form:

```
ViStatus WaitUntil<operation complete> (
                                        ViSession Vi,
                                        ViInt32 MaxTimeMilliseconds)
```

For IVI-COM, this function should be of the form:

```
HRESULT WaitUntil<operation complete> (
                                       [in]long MaxTimeMilliseconds)
```

For IVI.NET, this function should be of the form:

```
void WaitUntil<operation complete> (PrecisionTimeSpan maxTime)
```

The words in <operation complete> are identical to those in the non-blocking function. If needed, a Boolean output parameter (C/COM) or return value (.NET) may be added to indicate whether the operation timed out.

Refer to section 8.3, *TimeOut Parameters~~Time Out Parameters~~* for information on how to document and implement the MaxTimeMilliseconds and maxTime parameters.

Formatted: Font: Italic

15. Out-Of-Range Conditions

Functions that return floating point numbers may indicate that the number is not in the expected range for the measurement by returning NaN to indicate that the value is out-of-range. Whenever a driver includes a function that uses NaN to indicate out-of-range, it shall also provide a function to test values for out-of-range.

For waveforms and spectrums, the IWaveform and ISpectrum interfaces have special properties that indicate whether or not there is an out-of-range value in the waveform or array. Refer to section 4.3.4, *ContainsOutOfRangeElement*, in *IVI-3.18: IVI.NET Utility Classes and Interfaces Specification* for details.

16. Direct I/O

Direct I/O for IVI specific drivers consists of two elements:

- Basic read and write functions that are limited to sending whole commands and reading whole responses from the connected device, along with a Timeout attribute to allow the calling program to set appropriate timeouts for those read and write functions, if necessary. These functions are required for drivers that use message-based communication.
- A reference to the driver's underlying I/O that allows the calling program to use the underlying I/O directly to perform I/O to the connected device. This reference takes different forms depending on the API of the driver and the API of the underlying I/O. Since it is not always possible to expose the API of the underlying I/O in this way, this reference is optional.

Though simple, the basic read and write functions are flexible enough to handle a wide variety of instrument commands and responses, but do not provide many of the capabilities of a full I/O library such as VISA or VISA.NET.

- A single instrument command cannot be spread over multiple write function calls, and a single instrument response cannot be read with multiple read function calls. This keeps the complexity of managing termination characters and END signals out of the driver API.
- The read and write functions do not format or parse data to be sent to the instrument. The burden of formatting and parsing data when using the standard driver read and write functions falls to the calling program.
- The read and write functions do not support asynchronous I/O.
- The standard driver API does not include functions or attributes that are specific to particular protocols such as GPIB, USB, or LAN.

Addressing these limitations in a driver standard would involve duplicating much of the functionality of a full-featured I/O library, which is not practical either for the IVI Foundation or for driver developers. Instead, drivers can expose the driver's underlying I/O by providing references to it in the driver's instrument specific interface. VISA C sessions can be exposed by providing the session ID. VISA COM and VISA.NET sessions can be exposed by providing a reference to the session in the form of an interface reference property.

References to other I/O libraries can also be exposed using similar techniques if VISA is not the underlying I/O.

Once the calling program has a reference to the underlying I/O, it can perform any I/O operation that is available to the driver. If the underlying I/O is VISA, all the capabilities listed above as missing from the basic read and write API are available to the calling program via the VISA reference.

When simulating, it is at the discretion of the driver supplier whether to return an error for Direct I/O operations.

16.1 Direct I/O Properties

This section gives a complete description of each Direct I/O attribute.

- Direct I/O (IVI-COM and IVI.NET)
- I/O Timeout
- Session
- System (IVI-COM and IVI.NET)

16.1.1 Direct I/O (IVI-COM and IVI.NET)

Data Type	Access
<I/O Interface>	RO

.NET Property Name

System.DirectIO

COM Property Name

System.DirectIO

C Attribute Name

N/A

The Direct I/O property is reserved for cases where the underlying form of I/O is implemented in COM or .NET. In an IVI-C driver, there is no practical way to expose the underlying I/O if it is implemented in COM or .NET.

Description

Returns a reference to the <I/O Interface> interface, which is a .NET or COM interface supported by the driver's underlying I/O. This interface should allow the client program to access broad functionality in the driver's underlying I/O.

.NET Exceptions

Section 12, *Common IVI.NET Exceptions and Warnings*, in *IVI-3.2 Inherent Capabilities Specification*, defines general exceptions that may be thrown, and warning events that may be raised, by this property.

Compliance Notes

1. This property is optional, but if a specific driver exposes the underlying I/O with an interface reference property, the property shall have the name DirectIO and must be an element of the I<DriverName>System interface.

2. If the underlying I/O does not expose a .NET interface, this property shall not be implemented in a specific IVI.NET driver.

3. If the underlying I/O does not expose a COM interface, this property shall not be implemented in a specific IVI-COM driver.

4. If the underlying I/O is VISA-COM, this property shall return a reference to the IFormattedIO488 interface used by the underlying VISA-COM object.

5. If the underlying I/O is VISA.NET, this property shall return a reference to the IMessageBasedFormattedIO interface used by the underlying VISA.NET object.

   When simulating, this property always returns an Operation Not Supported error (COM) or throws an Operation Not Supported exception (.NET).

16.1.2 I/O Timeout

Data Type	Access
`PrecisionTimeSpan` (.NET) `Long` (COM) `ViInt32` (C)	R/W

.NET Property Name

`System.IOTimeout`

COM Property Name

`System. IOTimeout`

C Property Name

`<prefix>_ATTR_SYSTEM_IO_TIMEOUT`

Description

The I/O timeout.

.NET Exceptions

Section 12, *Common IVI.NET Exceptions and Warnings*, in *IVI-3.2 Inherent Capabilities Specification,* defines general exceptions that may be thrown, and warning events that may be raised, by this property.

16.1.3 Session

Data Type	Access
`viSession (C)` `UInt32 (COM, .NET)`	RO

.NET Property Name

`System.Session`

COM Property Name

`System.Session`

C Property Name

`<prefix>_ATTR_SYSTEM_IO_SESSION`

Description

An integer that identifies the session in the underlying I/O that implements the I/O connection to the device. This session ID should allow the client program to access broad functionality in the driver's underlying I/O.

This property applies only to drivers that use C message-based communications to communicate with devices, and it is optional for such drivers.

.NET Exceptions

Section 12, *Common IVI.NET Exceptions and Warnings*, in *IVI-3.2 Inherent Capabilities Specification,* defines general exceptions that may be thrown, and warning events that may be raised, by this property.

Compliance Notes

1. This property is optional in an IVI-COM or IVI.NET specific driver, but if the driver exposes the underlying I/O with a session ID, the property shall have the name `Session` and must be defined in the `I<DriverName>System` interface.

2. This property is optional in an IVI-C specific driver, but if the driver exposes the underlying I/O with a session ID, the attribute shall have the name `<prefix>_ATTR_SYSTEM_IO_SESSION` and must be defined in the `System` level of the attribute hierarchy.

3. If the underlying I/O does not expose a session ID, this property shall not be implemented in a specific IVI.NET, IVI-COM, or IVI-C driver.

4. The value for <prefix>_ATTR_SYSTEM_IO_SESSION shall be IVI_INHERENT_ATTR_BASE + 322.

5. When simulating, this property returns zero. It should not be assumed that this is a valid session.

16.1.4 System (IVI-COM and IVI.NET)

Data Type	Access
`I<DriverName>System`	RO

.NET Property Name

`<root>.System`

where `<root>` is the root interface of the instrument specific interface hierarchy.

COM Property Name

`<root>.System`

where `<root>` is the root interface of the instrument specific interface hierarchy.

C Attribute Name

`N/A`

There is no System interface in IVI-C.

Description

An interface reference pointer to the `I<Class>System` instrument specific interface. This provides access to the `System` interface from the root level of the driver's main class.

.NET Exceptions

Section 12, *Common IVI.NET Exceptions and Warnings*, in *IVI-3.2 Inherent Capabilities Specification,* defines general exceptions that may be thrown, and warning events that may be raised, by this property.

Compliance Notes

1. This property is optional in an IVI-COM or IVI.NET specific driver, but if the driver defines the `I<DriverName>System` interface, the root instrument specific interface shall contain an interface reference property named `System` that returns a reference to the interface.

2. When simulating, this property always returns a reference to a valid system interface.

16.2 Methods

This section gives a complete description of each Direct I/O method/function.

- Read Bytes
- Read String (IVI-COM and IVI.NET)
- Write Bytes
- Write String (IVI-COM and IVI.NET)

16.2.1 Read Bytes

Description

Reads a complete response from the instrument.

.NET Method Prototype

```
Byte[] System.ReadBytes ();
```

COM Method Prototype

```
HRESULT ReadBytes([out, retval] SAFEARRAY(BYTE) *pBuffer);
```

C Function Prototype

```
ViStatus _VI_FUNC <prefix>_viRead (ViSession DriverSession, ViInt64 BufferSize,
                                   ViByte Buffer[], ViInt64* ReturnCount);
```

Parameters

Inputs	Description	Data Type
`DriverSession (C)`	Unique identifier for an IVI session.	`ViSession`
`BufferSize (C)`	The size of the array, in bytes, passed into the function to hold the results.	`viInt64`
`pBuffer (COM, retval)`	An array allocated by the method to hold the bytes returned by the device.	`SAFEARRAY (BYTE)`
`Buffer (C)`	An array allocated by the calling program and passed into the function to hold the bytes returned by the device.	`ViByte[]`
`ReturnCount (C)`	The number of bytes in the array that were filled in by the function.	`viInt64*`

<Return Value> (.NET)	An array of bytes returned by the device.	Byte[]

.NET Exceptions

Section 12, *Common IVI.NET Exceptions and Warnings*, in *IVI-3.2 Inherent Capabilities Specification*, defines general exceptions that may be thrown, and warning events that may be raised, by this property.

Compliance Notes

1. This method reads a complete response from the connected device. It reads data from the device until it encounters an END indicator or termination character. If the device sends an END indicator or termination character that the underlying I/O cannot detect, the method times out.

2. For IVI-C, if the number of bytes read exceeds the capacity of the array, the extra bytes are discarded.

16.2.2 Read String (IVI-COM and IVI.NET)

Description

Reads a complete response from the instrument and returns it as a string.

.NET Method Prototype

```
String System.ReadString();
```

COM Method Prototype

```
HRESULT ReadString([out, retval] BSTR *pBuffer);
```

C Function Prototype

N/A

Parameters

Inputs	Description	Data Type
pBuffer (COM, retval)	The array allocated by the method to hold the data read from the instrument.	BSTR
<Return Value> (.NET)	A string returned by the device.	String

.NET Exceptions

Section 12, *Common IVI.NET Exceptions and Warnings*, in *IVI-3.2 Inherent Capabilities Specification*, defines general exceptions that may be thrown, and warning events that may be raised, by this property.

Compliance Notes

1. This method reads a complete response from the connected device. It reads data from the device until it encounters an END indicator or termination character. If the device sends an END indicator or termination character that the underlying I/O cannot detect, the method times out.

2. The string received from the instrument is converted to a .NET (Unicode) string before it is returned to the calling program.

16.2.3 Write Bytes

Description

Write an array of bytes to the device.

.NET Method Prototype

```
System.WriteBytes(Byte[] buffer);
```

COM Method Prototype

```
HRESULT WriteBytes([in] SAFEARRAY(BYTE) *buffer,
                   [out, retval] longlong *ReturnCount);
```

C Function Prototype

```
ViStatus _VI_FUNC <prefix>_viWrite (ViSession DriverSession, ViByte Buffer[],
                                    ViInt64 Count, ViInt64* ReturnCount);
```

Parameters

Inputs	Description	Data Type
`DriverSession (C)`	Unique identifier for an IVI driver session.	`ViSession`
`Buffer`	The array of bytes to be written to the device.	`ViByte[]`
`Count (C)`	The number of bytes to write to the device. Count must be less than or equal to the size of the data array.	`ViInt64`
`ReturnCount ( C, COM)`	The number of bytes actually written.	`ViInt64`

.NET Exceptions

Section 12, *Common IVI.NET Exceptions and Warnings*, in *IVI-3.2 Inherent Capabilities Specification*, defines general exceptions that may be thrown, and warning events that may be raised, by this property.

Compliance Notes

1. After writing the bytes in the array to the instrument, the driver appends any necessary termination characters or END signals needed to terminate the write. This implies that `data` must contain a complete command to be written to the connected device.

16.2.4 Write String (IVI-COM and IVI.NET)

Description

Write a string to the device.

.NET Method Prototype

```
System.WriteString(String data);
```

COM Method Prototype

```
HRESULT WriteString([in] BSTR data);
```

C Function Prototype

```
N/A
```

Parameters

Inputs	Description	Data Type
`Data`	The string to be written to the device.	`String`

.NET Exceptions

Section 12, *Common IVI.NET Exceptions and Warnings*, in *IVI-3.2 Inherent Capabilities Specification*, defines general exceptions that may be thrown, and warning events that may be raised, by this property.

Compliance Notes

1. After writing the bytes in the array to the instrument, the driver appends any necessary termination characters or END signals needed to terminate the write. This implies that `data` must contain a complete command to be written to the connected device.

2. The .NET (Unicode) string is converted to the correct format for the connected device before it is sent to the instrument.

16.3 Direct I/O Interfaces

This section gives a complete description of each Direct I/O interface.

- System (IVI-COM and IVI.NET)

16.3.1 System (IVI-COM and IVI.NET)

Description

Contains elements related to the system operation of a connected device, including direct I/O communication to/from the device.

.NET Interface

```
interface I<driverName>System;
```

COM Interface

```
interface I<driverName>System : IUnknown;
```

Compliance Notes

1. An IVI-COM or IVI.NET specific driver that uses message-based communication shall implement the System interface. That interface shall include all implemented standard Direct I/O methods and properties described in this document.

16.4 Direct I/O C Hierarchy (IVI-C)

An IVI-C specific driver that uses message-based communication shall include a level 1 System level in both attribute and function hierarchies. That hierarchy level shall include all implemented standard Direct I/O functions and attributes described in this document.

17. Asynchronous I/O (IVI.NET)

Some input/output (I/O) operations can take a long time relative to other client program operations. In such cases, an IVI specific driver may provide asynchronous programming support to execute the I/O operation on a different thread. This permits the client program to perform other operations, including driver operations, without waiting for the I/O operation to complete.

An IVI specific driver that provides an asynchronous I/O API should use the Asynchronous Programming Model (APM) to implement the asynchronous I/O. In the APM, an asynchronous operation is implemented as two methods: `Begin<Operation>` and `End<Operation>`, where `<Operation>` is the name of an I/O method that operates synchronously.

After calling the `Begin<Operation>` method, a client program can continue executing instructions on the calling thread while the specific driver performs `<Operation>` on a different thread. For each call to the `Begin<Operation>` method, the client program calls the `End<Operation>` method to get the results of the operation.

Example of asynchronous IO:

```
using (NIScope scopeSession = new NIScope("scope1", false, false))
{
   // …
   // Configure scope channel "0" properties.
   // …
   // Start the asynchronous Read operation.
   IAsyncResult result =
   scopeSession.Channels["0"].Measurement.BeginRead(null);
   // …
   // Do any additional work that can be done here
   // while the Read operation executes on a different thread.
   // …
   try
   {
      // EndRead blocks until the async work is complete.
      IWaveform<double> waveform =
      scopeSession.Channels["0"].Measurement.EndRead(result);
   }
   catch (Ivi.Scope.UnableToPerformMeasurementException ex)
   {
      // Handle UnableToDoMeasurementException from async Read operation.
   }
   catch (Ivi.Scope.ChannelNotEnabledException ex)
   {
      // Handle ChannelNotEnabledException from async Read operation.
   }
   catch (Ivi.Scope.MaxTimeExceededException ex)
   {
      // Handle MaxTimeExceededException from async Read operation.
   }
   //…
}
```

Note that the `IAsyncResult` class includes the following properties that can be useful to a client program:

- AsyncState
- AsyncWaitHandle
- CompletedSynchronously

- IsCompleted

See MSDN documentation for a complete explanation of each property. AsyncState is a reference the object that is supplied to `Begin<Operation>` as a parameter of the same name.

17.1 Asynchronous I/O Methods

This section gives a complete description of each Asynchronous I/O method.

- Begin<Operation>
- End<Operation>

where <Operation> is the name of a synchronous I/O method.

17.1.1 Begin<Operation>

Description

The `Begin<Operation>` method begins asynchronous operation <Operation> and returns an object that implements the `IAsyncResult` interface.

.NET Method Prototype

```
IAsyncResult Begin<Operation>(<Operation input parameters>, Object AsyncState);
```

Parameters

Inputs	Description	Data Type
`<Operation input parameters>`	The `Begin<Operation>` method has the same input parameters as the synchronous `<Operation>` method.	(Depends on input parameters)
`AsyncState`	Any object the client program wants to pass to `Begin<Operation>` to access through `IAsyncResult.AsyncState`. This object typically qualifies or contains information about the asynchronous operation.	Object
<Return Value>	`The IAsyncResult` interface that is passed to the `End<Operation>` method.	`IAsyncResult`

Compliance Notes

1. Any exception that occurs in asynchronous operation `<Operation>` shall be thrown when the client program calls the corresponding `End<Operation>` method.
2. The `Begin<Operation>` method may have additional parameters, such as a timeout.

17.1.2 End<Operation>

Description

The End<Operation> method ends the asynchronous operation <Operation>.

.NET Method Prototype

```
<Operation return type> End<Operation>(IAsyncResult asyncResult,
                                       <Operation output parameters>);
```

Parameters

Inputs	Description	Data Type
asyncResult	The reference to the pending asynchronous request to finish.	IAsyncResult
<Operation output parameters>	The End<Operation> method has the same output parameters as the synchronous <Operation> method	(Depends on output parameters
<Return Value>	The End<Operation> method has the same return type a the synchronous <Operation> method.	<Operation return type> (Depends on return value)

Compliance Notes

1. The asyncResult parameter shall be the first parameter of the End<Operation> method signature.
2. A client program must pass the instance of IAsyncResult returned by the corresponding call to the Begin<Operation> method. The End<Operation> method shall throw a System.InvalidOperationException in the following scenarios:
   - End<Operation> is called multiple times with the same IAsyncResult instance.
   - End<Operation> is called with an IAsyncResult instance that was not returned by the related Begin<Operation>.
3. If the operation represented by the IAsyncResult instance has not completed when the End<Operation> method is called, the End<Operation> method shall block the calling thread until the operation is complete. If the operation times out, it is considered to be complete.
4. An IVI specific driver shall notify the caller that the asynchronous operation completed by setting IsCompleted property to true and signaling the AsyncWaitHandle in the IAsyncResult instance corresponding to the asynchronous operation.
5. The Begin<Operation> method may have additional parameters, such as a timeout.

18. Instrument Class Specification Layout

Each instrument class specification shall contain the sections in Table 18-1~~Table 15-1~~.

Table 18-1 Section Labeling

Section Number	Section Topic
1	Overview
2	<ClassName> Class Capabilities
3	General Requirements
4	Base Capability Group
5	First Extension Group
…	
n	Last Extension Group
n+1	Attribute ID Definitions
n+2	Attribute Value Definitions
n+3	Function Parameter Value Definitions
n+4	Error and Completion Code Value Definitions
n+5	Hierarchies

Each instrument class specification shall contain the appendixes in Table 18-2~~Table 15-2~~.

Table 18-2 Appendix Labeling

Appendix Letter	Appendix Topic
A	Specific Driver Development Guidelines
B	Interchangeability Checking Guidelines

Tables and figures are numbered <main section>-<sequential number>. Numbers re-start at one in each major section.

Insert a page break before every first level section.

18.1 Overview Layout

Each Overview section shall contain the subsections:

1.1 Introduction

1.2 <ClassName> Class Overview

1.3 References

1.4 Definitions of Terms and Acronyms

Define terms which are specific to the instrument class. Terms of more general interest are defined in *IVI-5: Glossary*. When an instrument class specification working group discovers a term of general interest, it should submit the term to the glossary working group.

18.2 Capabilities Groups Layout

Each Capability Group section shall contain the subsections:

2.1 Introduction

2.2 <ClassName> Group Names

2.3 Repeated Capability Names

If the instrument class does not define any repeated capabilities, section 2.3 shall contain the text "The <ClassName> Class Specification does not define any repeated capabilities." If it does define one or more repeated capabilities, they shall be listed in section 2.3 and section 2.3 shall contain a sub-section for each different repeated capability describing how the name appears in the Configuration Store.

2.4 Boolean Attribute and Parameter Values

2.5 .NET Namespace

18.3 General Requirements Layout

Each General Requirements section shall contain the subsections:

3.1 Minimum Class Compliance

3.1.1 Disable Function

3.2 Capability Group Compliance

Section 3.1 shall include a statement that an IVI specific driver shall comply with the instrument class specification, IVI-3.1, and IVI-3.2.

If an instrument class needs to describe additional compliance rules for inherent capabilities beyond the requirements in *IVI-3.2: Inherent Capabilities Specification*, they shall appear in Sections 3.1.x. A section is added for each inherent capability with additional requirements.

18.4 Capability Group Section Layout

Each capability group, base and extensions, section shall contain the following subsections:

18.4.1 Overview:

An overview of the capability group that describes its purpose and general use.

18.4.2 Attributes: (Optional)

Defines the attributes that are a part of the capability group. For each attribute the capability group defines the name of the attribute, the data type, the access (read and write - R/W, or read only - RO), a description, defined values, and additional compliance requirements. Refer to Section 18.5~~16.4.1~~, Attribute Section Layout, for more information regarding the layout of attribute subsections. There is one attribute section for each attribute in a capability group.

If the instrument class capability group does not contain any attributes, the section is omitted.

The title of section shall be <Capability Group Name> Attributes.

After the list of attributes, include this paragraph:

This section describes the behavior and requirements of each attribute. The actual value for each attribute ID is defined in Section n+1, *<ClassName> Attribute ID Definitions*.

18.4.3 Functions: (Optional)

Defines the functions that are part of the capability group. For each function the capability group defines the function name, description, input parameters, output parameters, completion codes, and additional compliance requirements. Refer to Section 18.6~~16.4.2~~, *Function Section Layout*~~*Function Section Layout*~~ for more information regarding the layout of function subsections. There is one function section for each function in a capability group.

Formatted: Font: Italic

If the instrument class capability group does not contain any functions, the section is omitted.

The title of section shall be <Capability Group Name> Functions.

After the list of functions, include this paragraph:

This section describes the behavior and requirements of each function.

18.4.4 Behavior Model:

Defines the relationships between IVI driver attributes and functions with instrument behavior.

The title of section shall be <Capability Group Name> Behavior Model.

18.4.5 Group Compliance Notes: (Optional)

Section 3, General Requirements defines the general rules an IVI specific driver must follow to be compliant with a capability group. This section specifies additional compliance requirements and exceptions that apply to a particular capability group.

If the instrument class capability group does not contain any additional compliance requirements, the section is omitted.

The title of section shall be <Capability Group Name> Compliance Notes.

Adjust the section numbers as needed. If an optional section does not exist do not skip a level 2 heading number. For example, if a capability has no attributes, the Functions section is numbered n.2.

18.5 Attribute Section Layout

Insert a page break before each attribute section.

If the Attribute is only defined for a subset of the covered APIs, add a parenthetical note to the title of the attribute section indicating the APIs for which the attribute is defined. For example:

4.2.6 Measurement State (IVI.NET)

Each Attribute section shall contain the following subsections:

Capabilities Table:

A table with five columns with the following titles:

Data Type:

Specifies the data type of the attribute. Refer to Table 5-6, *Compatible Data Types for IVI Drivers* in *IVI-3.1: Driver Architecture Specification* for a complete list of allowed data types.

Whenever possible, the IVI-C API type name shall be used. For attributes where a IVI.NET prototype exists and the .NET data type has no IVI-C API equivalent, the IVI.NET API type name shall be listed. In some cases it may be necessary to list both the IVI-C API type name and the IVI.NET API type name. If the IVI.NET type name is listed, and it is (1) not a .NET Framework type, and (2) not defined in the Ivi.<InstrType> namespace, then the type name shall be qualified with the namespace name.

Access:

Specifies the kind of access the user has to the attribute. Possible values are RO, WO, and R/W.

- R/W (read/write) – indicates that the user can get and set the value of the attribute.
- RO (read-only) – indicates that the user can only get the value of the attribute.
- WO (write-only) – indicates that the user can only set the value of the attribute.

Applies to:

Specifies whether the attribute applies to the instrument as a whole or applies to a repeated capability. The field contains either the name of the repeated capability or N/A. Examples of repeated capability names are Trace, Display, and Channel.

Coercion:

Some attributes represent a continuous range of values, but allow the IVI specific driver to coerce the value that the user requests to a value that is more appropriate for the instrument. For these cases the specification defines the direction in which the IVI specific driver is allowed to coerce a value. Possible values are Up, Down, and None.

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

Any other kind of coercion is indicated with a documentation note.

- High Level Function(s):

Lists all high level functions that access the attribute.

For example,

Data Type	Access	Applies to	Coercion	High Level Functions
ViReal64	R/W	N/A	None	Configure Acquisition Record

.NET Property Name:

The name of the IVI.NET property as it appears in the IVI.NET interface or class definition file. The name of the property is preceded by all the interface reference pointers, separated by dots, needed to reach this method in the hierarchy. Collections are indicated by plural – for example, Channels. Items in a collection are indicated by the collection name followed by [] – for example, Channels[].

If an IVI.NET property does not exist, use N/A here and refer to any function or attribute that provides equivalent functionality.

.NET Enumeration Name:

If the value of the IVI.NET property is an enumeration, its name appears here. If it is not an enumeration, then this section should be omitted. The namespace shall be added before the enumeration name unless the enumeration is a .NET framework enumeration, or the namespace of the enumeration is "Ivi.<InstrType>". If the namespace is "Ivi.<InstrType>", it may be omitted.

COM Property Name:

The name of the IVI-COM property as it appears in the IDL file. The name of the property is preceded by all the interface reference pointers, separated by dots, needed to reach this method in the hierarchy.

If an IVI-COM property does not exist, use N/A here and refer to any function or attribute that provides equivalent functionality.

COM Enumeration Name:

If the value of the IVI-COM property is an enumeration, its name appears here. If it's not an enumeration, then this section should be omitted.

C Constant Name:

The constant identifier used to access the attribute and defined in the include file.

If an IVI-C attribute does not exist, use N/A here and refer to any function or attribute that provides equivalent functionality.

Description:

Describes the attribute and its intended use.

Defined Values: (Optional)

Defines all the attribute values that the instrument class specifies for the attribute using a table with two main columns. The left column is labeled Name and contains the generic name for a value and the right column Description. Under the Description column are entries for the Identifier used in every language of interest to IVI. Note that namespaces should be omitted for .NET values.

For example,

Name	***Description***

		Language	*Identifier*
Current Trip	The power supply disables the output when the output current is equal to or greater than the value of the Current Limit attribute.		
		.NET	`CurrentLimit.Trip`
		C	`IVIDCPWR_VAL_CURRENT_TRIP`
		COM	`IviDcPwrCurrentLimitTrip`
Current Regulate	The power supply restricts the output voltage such that the output current is not greater than the value of the Current Limit attribute.		
		.NET	`CurrentLimit.Regulate`
		C	`IVIDCPWR_VAL_CURRENT_REGULATE`
		COM	`IviDcPwrCurrentLimitRegulate`

The actual numeric value associated with a defined value is specified in the Attribute Value Definitions section at the back of the specification.

Do not include a Defined Values subsection for Boolean attributes.

If the attribute does not have any defined values, the section is omitted.

.NET Exceptions

Defines the possible exceptions that may be thrown by the property. This section shall contain at least the text "The *IVI-3.2: Inherent Capabilities Specification* defines general exceptions that may be thrown, and warning events that may be raised, by this property."

If the property can return additional instrument class specific exceptions or warnings, they shall be listed in tables as appropriate, as shown:

“The table below specifies additional class-defined exceptions for this property.”

Exception Name	Description
Waveform In Use Exception	The function generator is currently configured to produce the specified waveform or the waveform is part of an existing sequence.

“The table below specifies additional class-defined warnings for this property.”

Exception Name	Description
Invalid Waveform Element	One of the elements in the waveform array is invalid.

Compliance Notes: (Optional)

The General Requirements section defines the general rules an IVI specific driver must follow to be compliant with an attribute. This section specifies additional compliance requirements and exceptions that apply to a particular attribute.

If the attribute does not contain any additional compliance requirements, the section is omitted.

18.6 Function Section Layout

Insert a page break before each function section.

If the function is only defined for a subset of the covered APIs, add a parenthetical note to the title of the attribute section indicating the APIs for which the attribute is defined. For example:

4.3.2 Configure Measurement (IVI-C Only)

Each function section shall contain the following parts:

Description

Describes the behavior and intended use of the function.

.NET Method Prototype:

Defines the IVI.NET prototype. The name of the method is preceded by all the interface reference pointers, separated by dots, needed to reach this method in the hierarchy. Each parameter is preceeded by its type. If applicable, each parameter is preceded by an indication of whether it is out, or ref. Collections are indicated by plural – for example, Channels. Items in a collection are indicated by the collection name followed by [] – for example, Channels[]. For example,

```
void Channel[].Range.Configure (Double lower,
                                Double upper);
```

If a parameter type is not defined within the "Ivi.<ClassType>" namespace or in the .NET Framework classes, the type shall be qualified with the namespace.

If an IVI.NET method does not exist, use N/A here and refer to any function or attribute that provides equivalent functionality.

COM Method Prototype:

Defines the IVI-COM prototype. The name of the method is preceded by all the interface reference pointers, separated by dots, needed to reach this method in the hierarchy. As in IDL, each parameter is preceded by an indication of whether it is in, out, or retval as well as its type. For example,

```
HRESULT Measurements.IsWaveformElementInvalid ([in] DOUBLE MeasurementValue,
                                    [out, retval] VARIANT_BOOL *IsInvalid);
```

If an IVI-COM method does not exist, use N/A here and refer to any function or attribute that provides equivalent functionality.

C Prototype:

Defines the IVI-C Language prototype. For example,

```
ViStatus IviScope_IsInvalidWfmElement (ViSession Vi,
                                       ViReal64 MeasurementValue,
                                       ViBoolean *IsOverrange);
```

If an IVI-C function does not exist, use N/A here and refer to any function or attribute that provides equivalent functionality.

Parameters:

Describes each function parameter using one or two tables. The first table is for input parameters. It has three columns. The left column is labeled Inputs and the middle column is labeled Description. The right column is labeled Base Type.

For example:

Inputs	Description	Base Type
`Vi`	Instrument handle	ViSession
`MeasurementValue`	Pass the measurement value you obtain from one of the Read or Fetch functions.	ViReal64

If the function generates one or more outputs, this section contains a second table for output parameters. It also has three columns. The left column is labeled Outputs and the middle column is labeled Description. The right column is labeled Base Type.

For example:

Outputs	Description	Base Type
`IsOverrange`	Returns whether the `MeasurementValue` is a valid measurement or an overrange condition. Valid Return values: `True` - overrange condition occurred. `False` - valid measurement.	ViBoolean

Whenever possible, the IVI-C API type name shall be used. For parameters where an IVI.NET prototype exists and the .NET data type has no IVI-C API equivalent, the IVI.NET API type name shall be listed. In some cases it may be necessary to list both the IVI-C API type name and the IVI.NET API type name.

If a parameter is only defined for a subset of the covered APIs, add a parenthetical note in the first column indicating the APIs for which the attribute is defined.

Defined Values for a Parameter: (Optional)

Occasionally, function parameters are enumerations without a corresponding attribute. In this case, the enumeration shall be described using a table of the form used to describe the defined values for an enumerated attribute. The title of this part shall use the actual name of the parameter in place of the words "a Parameter".

Do not include a Defined Values subsection for Boolean parameters.

Return Values (C/COM)

Defines the possible completion codes for the function. This section shall contain at least the text "The *IVI-3.2: Inherent Capabilities Specification* defines general status codes that this function can return."

If the function can return additional instrument class specific values, they shall be listed in a table as appropriate, as shown:

“The table below specifies additional instrument class-defined status codes for this function.”

Completion Codes	Description
Waveform In Use	The function generator is currently configured to produce the specified waveform or the waveform is part of an existing sequence.

.NET Exceptions

Defines the possible exceptions that may be thrown by the IVI.NET method. This section shall contain at least the text "The *IVI-3.2: Inherent Capabilities Specification* defines general exceptions that may be thrown, and warning events that may be raised, by this method."

If the function can return additional instrument class specific exceptions or warnings, they shall be listed in tables as appropriate, as shown:

“The table below specifies additional class-defined exceptions for this method.”

Completion Codes	Description
Waveform In Use Exception	The function generator is currently configured to produce the specified waveform or the waveform is part of an existing sequence.

“The table below specifies additional class-defined warnings for this method.”

Completion Codes	Description
Invalid Waveform Element	One of the elements in the waveform array is invalid.

Compliance Notes: (Optional)

The General Requirements section defines the general rules an IVI specific driver must follow to be compliant with a function. This section specifies additional compliance requirements and exceptions that apply to a particular function.

If the function does not contain any additional compliance requirements, the section shall be omitted.

18.7 Attribute ID Definitions Layout

This section shall contain a table titled, “<ClassName> Attribute ID Values”, with two columns. The left column is titled "Attribute Name" and under it are all the attributes defined in the instrument class specification. The right column is titled "ID Definition" and under it is a numerical value for the attribute. This value may be in terms of other defined values.

18.8 Attribute Value Definitions Layout

This section shall contain a table for each attribute which has defined values. These tables each have four columns. The left column is titled "Value Name" and under it are all the values defined in the instrument class specification. The second column is titled "Language" and under it are entries for every languages of interest to IVI. The third column is titled "Identifier" and under it are the identifiers used in the various

languages. The right column is titled "Actual Value" and under it is a numerical value for the value. This value may be in terms of other defined values.

All IVI.NET enumerations shall be zero-based. Note that extension bases are not used in IVI.NET.

For example,

Current Limit Behavior

Value Name	*Language*	*Identifier*	*Actual Value*
Current Regulate	.NET	`CurrentLimit.Regulate`	`0`
	C	`IVIDCPWR_VAL_CURRENT_LIMIT_REGULATE`	`0`
	COM	`IviDCPwrCurrentLimitRegulate`	`0`
Current Trip	.NET	`CurrentLimit.Trip`	`1`
	C	`IVIDCPWR_VAL_CURRENT_LIMIT_TRIP`	`1`
	COM	`IviDCPwrCurrentLimitTrip`	`1`
Current Limit Behavior Class Ext Base	C	`IVIDCPWR_VAL_CURRENT_LIMIT_BEHAVIOR_CLASS_EXT_BASE`	`100`
	COM	`N/A`	
Current Limit Behavior Specific Ext Base	C	`IVIDCPWR_VAL_CURRENT_LIMIT_BEHAVIOR_SPECIFIC_EXT_BASE`	`1000`
	COM	`N/A`	

18.9 Function Parameter Value Definitions Layout

This section contains a subsection for each function which has one or more parameters which have defined values. The title of the subsection is the generic name of the function.

Each subsection shall contain a table for each parameter in the function which has defined values. The table has the same format at the tables in Attribute Value Definitions Layout with an additional top row containing Parameter and the parameter name.

All IVI.NET enumerations shall be zero-based.

For example,

Configure Output Range

Parameter: `RangeType`

COM Enumeration Name: `IviDcPwrRangeTypeEnum`

Value Name	*Language*	*Identifier*	*Actual Value*
Current	.NET	`Range.Current`	`0`
	C	`IVIDCPWR_VAL_RANGE_CURRENT`	`0`
	COM	`IviDCPwrRangeCurrent`	`0`
Voltage	.NET	`Range.Voltage`	`1`
	C	`IVIDCPWR_VAL_RANGE_VOLTAGE`	`1`
	COM	`IviDCPwrRangeVoltage`	`1`
Configure Output Range	C	`IVIDCPWR_VAL_RANGE_TYPE_CLASS_EXT_BASE`	`500`

Class Ext Base	COM	N/A	
Configure Output Range Specific Ext Base	C	IVIDCPWR_VAL_RANGE_TYPE_SPECIFIC_EXT_BASE	1000
	COM	N/A	

18.10 Error, Completion Code, and Exception Class Definitions Layout

Defines all the IVI-C and IVI-COM error and completion codes, and IVI.NET exceptions and warning GUIDs that the instrument class specifies. This section contains two tables. The first table lists the error name as it is used throughout the instrument class specification, a more complete description of the error, the identifiers used in languages of interest to IVI with an associated value. The table has this form:

Table 18-34. IviScope Error and Completion Codes

Error Name	Description		
	Language	Identifier	Value (hex)
Invalid Waveform Element	One of the elements in the waveform array is invalid.		
	.NET		2733A6B6-13E2-4480-9D60-B97FC11B68FC
	C	IVISCOPE_WARN_INVALID_WFM_ELEMENT	0x3FFA2001
	COM	S_IVISCOPE_INVALID_WFM_ELEMENT	0x80042001
Channel Not Enabled	Specified channel is not enabled.		
	.NET	ChannelNotEnabledException	N/A
	C	IVISCOPE_ERROR_CHANNEL_NOT_ENABLED	0xBFFA2001
	COM	E_IVISCOPE_CHANNEL_NOT_ENABLED	0x80042001
Max Time Exceeded	Maximum time exceeded before the operation completed.		
	.NET	Ivi.Driver.MaxTimeExceededException	IVI defined exception (see IVI-3.2)
	C	IVISCOPE_ERROR_MAX_TIME_EXCEEDED	0xBFFA2003
	COM	E_IVISCOPE_MAX_TIME_EXCEEDED	0x80042003

The second table defines the format of the message string associated with the error. In IVI-C, this string is returned by the Error Message function. In IVI-COM, this string is the description contained in the ErrorInfo object. In IVI.NET, this string is returned in the exception's Message property. The second table has this form:

Note: In the description string table entries listed below, **%s** is always used to represent the component name.

Table 18-45. IviScope Error Message Strings

Name	Message String
Invalid Waveform Element	"%s: Invalid waveform element
Channel Not Enabled	"%s: Channel not enabled

Table 18-45. IviScope Error Message Strings

Name	Message String
Unable To Perform Measurement	"%s: Unable to perform measurement
Max Time Exceeded	"%s: Maximum time exceeded"
Invalid Acquisition Type	"%s: Invalid acquisition type

18.11 Hierarchies

Each Hierarchies section shall contain the subsections: IVI.NET Hierarchy, IVI-COM Hierarchy, IVI-C Function Hierarchy, and IVI-C Attribute Hierarchy.

18.11.1 IVI.NET Hierarchy

This section shall contain a table with three columns showing which properties and methods are in each interface. The sample table shown here lists the inherent methods and properties though individual instrument class specifications shall not include them. The following paragraph shall be included before the table:

"The full <ClassName> .NET Hierarchy includes the Inherent Capabilities Hierarchy as defined in Section 4.1, *.NET Inherent Capabilities* of *IVI-3.2: Inherent Capabilities Specification*. To avoid redundancy, the Inherent Capabilities are omitted here."

The table shall have the form:

Table n+4-1. <ClassName> .NET Hierarchy

.NET Interface Hierarchy	Generic Name	Type
`DriverOperation`		
`Cache`	Cache	P
`ClearInterchangeWarnings`	Clear Interchange Warnings	M
`DriverSetup`	Driver Setup	P
`GetNextCoercionRecord`	Get Next Coercion Record	M
`GetNextInterchangeWarning`	Get Next Interchange Warning	M
`InterchangeCheck`	Interchange Check	P
`InvalidateAllAttributes`	Invalidate All Attributes	M
`LogicalName`	Logical Name	P
`QueryInstrumentStatus`	Query Instrument Status	P
`RangeCheck`	Range Check	P
`RecordCoercions`	Record Value Coercions	P
`ResetInterchangeCheck`	Reset Interchange Check	M
`IoResourceDescriptor`	Resource Descriptor	P
`Simulate`	Simulate	P
`Identity`		
`Description`	Component Description	P

.NET Interface Hierarchy	Generic Name	Type
Locator	Component Locator	P
Prefix	Component Prefix	P
Revision	Component Revision	P
Vendor	Component Vendor	P
InstrumentFirmwareRevision	Instrument Firmware Revision	P
GroupCapabilities	Class Group Capabilities	P
InstrumentManufacturer	Instrument Manufacturer	P
InstrumentModel	Instrument Model	P
SpecificationMajorVersion	Component Class Spec Major Version	P
SpecificationMinorVersion	Component Class Spec Minor Version	P
SupportedInstrumentModels	Supported Instrument Models	P
Utility		
Disable	Disable	M
ErrorQuery	Error Query	M
Reset	Reset	M
ResetWithDefaults	Reset With Defaults	M
SelfTest	Self Test	M

This section shall also contain two addition subsections: Interfaces and Interface Reference Properties.

18.11.1.1 <ClassName> IVI.NET Interfaces

This section list all of the interfaces, and describes all the interface reference properties used to navigate the IVI.NET hierarchy.

Starting with root I<ClassName> interface, the section contains a list of each interface that contains interface reference properties, with a list of interfaces referenced. For example:

"In addition to implementing IVI inherent capabilities interfaces, IviPwrMeter-interfaces contain interface reference properties for accessing the following IviPwrMeter interfaces:

- "IIviPwrMeterChannels
- "IIviPwrMeterMeasurement
- "IIviPwrMeterReferenceOscillator
- "IIviPwrMeterTrigger

"The IIviPwrMeterChannels interface contains methods and properties for accessing a collection of objects that implement the IIviPwrMeterChannel interface.

"The IIviPwrMeterChannel interface contains interface reference properties for accessing additional the following IviPwrMeter interfaces:

- "IIviPwrMeterAveraging
- "IIviPwrMeterDutyCycleCorrection
- "IIviPwrMeterRange"

18.11.1.2 Interface Reference Properties

This section list all of the interface reference property names, and their correspondence to interface names, in table form. The following paragraph shall be included before the table:

"Interface reference properties are used to navigate the <ClassName> .NET hierarchy. This section describes the interface reference properties that the [list of interfaces from previous section] interfaces define. All interface reference properties are read-only."

The table shall have the form:

Interface Data Type	Access
`IIviPwrMeterReferenceOscillator`	ReferenceOscillator
`IIviPwrMeterMeasurement`	Measurement
`IIviPwrMeterChannels`	Channels
`IIviPwrMeterChannel`	Channels[]
`IIviPwrMeterTrigger`	Trigger
`IIviPwrMeterInternalTrigger`	InternalTrigger
`IIviPwrMeterAveraging`	Channels[].Averaging
`IIviPwrMeterRange`	Channels[].Range
`IIviPwrMeterDutyCycleCorrection`	Channels[].DutyCycleCorrection

The Access may be via an interface reference property name or via IVI.NET collection indexer operators. Note that hierarchy information is shown.

18.11.2 IVI-COM Hierarchy

This section shall contain a table with three columns showing which properties and methods are in each interface. The sample table shown here lists the inherent methods and properties though individual instrument class specifications shall not include them. Do include a paragraph of the form:

"The full <ClassName> COM Hierarchy includes the Inherent Capabilities Hierarchy as defined in Section 4.2, *COM Inherent Capabilities* of *IVI-3.2: Inherent Capabilities Specification*. To avoid redundancy, the Inherent Capabilities are omitted here."

The table shall have the form:

Table n+4-1. <ClassName> COM Hierarchy

COM Interface Hierarchy	Generic Name	Type
`Close`	Close	M
`DriverOperation`		
`Cache`	Cache	P
`ClearInterchangeWarnings`	Clear Interchange Warnings	M
`DriverSetup`	Driver Setup	P

COM Interface Hierarchy	Generic Name	Type
GetNextCoercionRecord	Get Next Coercion Record	M
GetNextInterchangeWarning	Get Next Interchange Warning	M
InterchangeCheck	Interchange Check	P
InvalidateAllAttributes	Invalidate All Attributes	M
LogicalName	Logical Name	P
QueryInstrumentStatus	Query Instrument Status	P
RangeCheck	Range Check	P
RecordCoercions	Record Value Coercions	P
ResetInterchangeCheck	Reset Interchange Check	M
IoResourceDescriptor	Resource Descriptor	P
Simulate	Simulate	P
Identity		
Description	Component Description	P
Locator	Component Locator	P
Prefix	Component Prefix	P
Revision	Component Revision	P
Vendor	Component Vendor	P
InstrumentFirmwareRevision	Instrument Firmware Revision	P
GroupCapabilities	Class Group Capabilities	P
InstrumentManufacturer	Instrument Manufacturer	P
InstrumentModel	Instrument Model	P
SpecificationMajorVersion	Component Class Spec Major Version	P
SpecificationMinorVersion	Component Class Spec Minor Version	P
SupportedInstrumentModels	Supported Instrument Models	P
Initialize	Initialize With Options	M
Initialized	Initialized	P
Utility		
Disable	Disable	M
ErrorQuery	Error Query	M
LockSession	Lock Session	M
Reset	Reset	M
ResetWithDefaults	Reset With Defaults	M
SelfTest	Self Test	M
UnlockSession	Unlock Session	M

This section shall also contain two addition subsections: Interfaces and Interface Reference Properties.

18.11.2.1 Interfaces

This section describes all the IVI-COM interfaces exposed by the IVI-COM driver. It contains a subsection for each interface. It includes a table of the form:

Table n+4-56. <ClassName> Interface GUIDs

Interface	GUID
I<ClassName>	{00000000-0000-0000-0000-000000000000}
I<ClassName><Itf1>	{00000000-0000-0000-0000-000000000000}
I<ClassName><Itf2>	{00000000-0000-0000-0000-000000000000}
I<ClassName><Itf3>	{00000000-0000-0000-0000-000000000000}

The GUIDs in the table are the actual GUIDs associated with the interface in the IDL. Class developers shall obtain GUIDs for interfaces and type libraries from the IVI Shared Components Working Group chairman.

18.11.2.2 <ClassName> IVI-COM Interfaces

This section list all of the interfaces, describes all the interface reference properties used to navigate the IVI-COM hierarchy, and includes a list of interface GUIDs.

Starting with root I<ClassName> interface, the section contains a list of each interface that contains interface reference properties, with a list of interfaces referenced. For example:

"In addition to implementing IVI inherent capabilities interfaces, IviPwrMeter-interfaces contain interface reference properties for accessing the following IviPwrMeter interfaces:

- "IIviPwrMeterChannels
- "IIviPwrMeterMeasurement
- "IIviPwrMeterReferenceOscillator
- "IIviPwrMeterTrigger

"The IIviPwrMeterChannels interface contains methods and properties for accessing a collection of objects that implement the IIviPwrMeterChannel interface.

"The IIviPwrMeterChannel interface contains interface reference properties for accessing additional the following IviPwrMeter interfaces:

- "IIviPwrMeterAveraging
- "IIviPwrMeterDutyCycleCorrection
- "IIviPwrMeterRange"

This section also includes a table of the form:

Table n+4-6. <ClassName> Interface GUIDs

Interface	GUID
I<ClassName>	{00000000-0000-0000-0000-000000000000}
I<ClassName><Itf1>	{00000000-0000-0000-0000-000000000000}
I<ClassName><Itf2>	{00000000-0000-0000-0000-000000000000}
I<ClassName><Itf3>	{00000000-0000-0000-0000-000000000000}

The GUIDs in the table are the actual GUID associated with the interface in the IDL.

18.11.2.3 Interface Reference Properties

This section list all of the interface reference property names, and their correspondence to interface names, in table form. The following paragraph shall be included before the table:

"Interface reference properties are used to navigate the <ClassName> COM hierarchy. This section describes the interface reference properties that the [list of interfaces from previous section] interfaces define. All interface reference properties are read-only."

The table shall have the form:

Interface Data Type	Access
IIviPwrMeterReferenceOscillator	ReferenceOscillator
IIviPwrMeterMeasurement	Measurement
IIviPwrMeterChannels	Channels
IIviPwrMeterChannel	Channels.Item()
IIviPwrMeterTrigger	Trigger
IIviPwrMeterInternalTrigger	InternalTrigger
IIviPwrMeterAveraging	Channels.Item().Averaging
IIviPwrMeterRange	Channels.Item().Range
IIviPwrMeterDutyCycleCorrection	Channels.Item().DutyCycleCorrection

18.11.2.4 IVI-COM Category

This section specifies the IVI-COM Category and Category ID (CATID). For example,

The IviFgen class IVI-COM Category shall be "IviFgen", and the Category ID (CATID) shall be {47ed5156-a398-11d4-ba58-000064657374}.

Class developers shall obtain CATIDs for categories from the IVI Shared Components Working Group chairman.

18.11.3 IVI-C Function Hierarchy

This section shall contain a table. The table shall follow the form of **Table 13-1** Prefix Function Hierarchy~~**Table 13-1** Prefix Function Hierarchy~~. Section 13.1 describes the contents of this table.

The table shall not include any of the inherent functions. Instead the section shall contain a paragraph of the form:

"The <ClassName> class function hierarchy is shown in the following table. The full <ClassName> C Function Hierarchy includes the Inherent Capabilities Hierarchy as defined in Section 4.3, *C Inherent Capabilities* of *IVI-3.2: Inherent Capabilities Specification*. To avoid redundancy, the Inherent Capabilities are omitted here."

18.11.4 IVI-C Attribute Hierarchy

This section shall contain a single table with two columns. The sample table shown here lists the inherent attributes though individual instrument class specifications shall not include them. Do include a paragraph of the form:

The <ClassName> class attribute hierarchy is shown in the following table. The full <ClassName> C Attribute Hierarchy includes the Inherent Capabilities Hierarchy as defined in Section 4.3, *C Inherent Capabilities* of *IVI-3.2: Inherent Capabilities Specification*. To avoid redundancy, the Inherent Capabilities are omitted here.

The table has the form:

Table n+4-5. <ClassName> IVI-C Attributes Hierarchy

Category or Generic Attribute Name	C Defined Constant
Inherent IVI Attributes	
User Options	
Range Check	<CLASS_NAME>_ATTR_RANGE_CHECK
Query Instrument Status	<CLASS_NAME>_ATTR_QUERY_INSTRUMENT_STATUS
Cache	<CLASS_NAME>_ATTR_CACHE
Simulate	<CLASS_NAME>_ATTR_SIMULATE
Record Value Coercions	<CLASS_NAME>_ATTR_RECORD_COERCIONS
Interchange Check	<CLASS_NAME>_ATTR_INTERCHANGE_CHECK
Class Driver Identification	
Class Driver Description	<CLASS_NAME>_ATTR_CLASS_DRIVER_DESCRIPTION
Class Driver Prefix	<CLASS_NAME>_ATTR_CLASS_DRIVER_PREFIX
Class Driver Vendor	<CLASS_NAME>_ATTR_CLASS_DRIVER_VENDOR
Class Driver Revision	<CLASS_NAME >_ATTR_CLASS_DRIVER_REVISION
Class Driver Class Spec Major Version	<CLASS_NAME >_ATTR_ CLASS_DRIVER_CLASS_SPEC_MAJOR_VERSION
Class Driver Class Spec Minor Version	<CLASS_NAME >_ATTR_ CLASS_DRIVER_CLASS_SPEC_MINOR_VERSION
Driver Identification	
Specific Driver Description	<CLASS_NAME >_ATTR_SPECIFIC_DRIVER_DESCRIPTION
Specific Driver Prefix	<CLASS_NAME >_ATTR_SPECIFIC_DRIVER_PREFIX
Specific Driver Locator	<CLASS_NAME >_ATTR_SPECIFIC_DRIVER_LOCATOR
Specific Driver Vendor	<CLASS_NAME >_ATTR_SPECIFIC_DRIVER_VENDOR
Specific Driver Revision	<CLASS_NAME >_ATTR_SPECIFIC_DRIVER_REVISION

Table n+4-5. <ClassName> IVI-C Attributes Hierarchy

Category or Generic Attribute Name	C Defined Constant
Specific Driver Class Spec Major Version	<CLASS_NAME >_ATTR_ SPECIFIC_DRIVER_CLASS_SPEC_MAJOR_VERSION
Specific Driver Class Spec Minor Version	<CLASS_NAME >_ATTR_ SPECIFIC_DRIVER_CLASS_SPEC_MINOR_VERSION
Driver Capabilities	
Supported Instrument Models	<CLASS_NAME >_ATTR_SUPPORTED_INSTRUMENT_MODELS
Class Group Capabilities	<CLASS_NAME >_ATTR_GROUP_CAPABILITIES
Instrument Identification	
Instrument Manufacturer	<CLASS_NAME >_ATTR_INSTRUMENT_MANUFACTURER
Instrument Model	<CLASS_NAME >_ATTR_INSTRUMENT_MODEL
Instrument Firmware Revision	<CLASS_NAME >_ATTR_INSTRUMENT_FIRMWARE_REVISION
Advanced Session Information	
Logical Name	<CLASS_NAME >_ATTR_LOGICAL_NAME
I/O Resource Descriptor	<CLASS_NAME >_ATTR_IO_RESOURCE_DESCRIPTOR
Driver Setup	<CLASS_NAME >_ATTR_DRIVER_SETUP

18.12 Appendix A: IVI Specific Driver Development Guidelines Layout

Each IVI Specific Driver Development Guidelines appendix shall contain the subsections:

A.1 Introduction

A.2 Disabling Unused Extension Groups

A.3 through n Special Considerations for ... (optional)

Section A.2 contains an entry for each extension group. The special consideration sections contain information about any topic of interest to IVI driver writers. For example, how to implement instrument class required features for instruments that do not follow the model assumed in the specification.

18.13 Appendix B: Interchangeability Checking Rules Layout

Each Interchangeability Checking Guidelines appendix shall contain the subsections:

B.1 Introduction

B.2 When to Perform Interchangeability Checking

B.3 Interchangeability Checking Rules

Section B.3 shall contain the names of every base and extension in bold on a separate. Each of these names shall be followed by one or more paragraphs describing the driver's behavior for that group.

18.14 Obsolete: Appendix C & D

Specifications previously included IVI-C and IVI-COM source for the API. These shall no longer be included in class specifications.

19. Accessing Instrument Descriptions

Instruments vendors might provide instrument descriptions conforming to a specific standard format such as ATML Instrument Description IEEE Std. 1671.2. In some cases a vendor might provide such a description for each model of an instrument. For users to locate the instrument description for a particular instrument, it is necessary for instrument drivers to include a means of retrieving the location of instrument descriptions.

Methods to locate instrument descriptions may be included in a driver's instrument specific API. If methods that provide this functionality are needed in an instrument specific API, the method signatures shall match the signatures in this section. The methods in this section provide the location in the form of a URL (Universal Resource Locator). This allows flexibility for retrieving the information.

The placement of these methods within the hierarchy of the IVI driver is at the discretion of the driver supplier. However, the methods should located with other instrument-specific, system-related attributes and methods in the instrument-specific portion of the driver hierarchy.

The instrument driver should document the model description strings and format version strings that it recognizes.

19.1 Get<Format>InstrumentDescriptionLocation

Description

Get< Format >InstrumentDescriptionLocation represents a template for the name of the function that returns the URL for the instrument descriptor in the relevant format.

For IEEE Std 1671.2™ the function shall be called **GetATMLInstrumentDescriptionLocation**.

If ModelDescription is non-NULL and not empty, the function returns the URI of the description of the specified instrument model. If no description is found, the function returns an empty string in C/COM.

If the ModelDescription is NULL or empty and the driver is not connected to an instrument, the function shall return a NULL URI or an empty string.

.NET Method Prototype

```
String Get<Format>InstrumentDescriptionLocation(String formatVersion,
                                                String modelDescription)
```

COM Method Prototype

```
HRESULT Get<Format>InstrumentDescriptionLocation([In] BSTR FormatVersion,
               [In] BSTR ModelDescription,
               [Out] BSTR* Location )
```

C Prototype

```
ViStatus Get<Format>InstrumentDescriptionLocation (ViSession Vi,
               ViConstString FormatVersion,
               ViConstString ModelDescription,
               ViChar LocationBuffer[],
               ViInt32 LocationBufferSize)
```

Parameters

Inputs	Description	Base Type
Vi	Instrument handle	ViSession
FormatVersion	FormatVersion specifies the version of the format requested. Empty string or NULL requests the latest version available.	ViConstString
ModelDescription	ModelDescription identifies the instrument model. If empty string or NULL, the function returns the location o the description for the instrument to which the driver is connected.	ViConstString
LocationBufferSize	The number of bytes in the ViChar array that the user specifies for the LocationBuffer parameter.	ViInt32

Outputs	Description	Base Type
Return value (.NET) Location (COM)	URL of Instrument Description or empty string if not available	BSTR*
LocationBuffer (C)	URL of Instrument Description or empty string if not available	viChar[]

Return Values (C/COM)

The *IVI-3.2: Inherent Capabilities Specification* defines general status codes that this function can return.

.NET Exceptions

The *IVI-3.2: Inherent Capabilities Specification* defines general exceptions that may be thrown, and warning events that may be raised, by this method.

20. Expressing Tone

Grammatically, auxiliaries are added before verbs to express tone. Be precise when using auxiliaries so the reader is never confused about the intent of the specification. Omitting an appropriate auxiliary can be as confusing as the wrong one.

20.1 Requirement

The auxiliaries shown in Table 20-1~~Table 16-1~~ shall be used to indicate requirements strictly to be followed in order to conform to the specification and from which no deviation is permitted.

Table 20-1 Requirement

Auxiliary	Equivalent expressions for use in exceptional cases
shall	is to is required to it is required that has to only … is permitted it is necessary
shall not	is not allowed [permitted] [acceptable] [permissible] is required to be not is required that … be not is not to be

Do not use “must” as an alternative for “shall”. Do not use “may not” instead of “shall not” to express a prohibition. To express a direct instruction, for example referring to steps to be taken in a test method, use the imperative mood in English.

20.2 Recommendation

The auxiliaries shown in Table 20-2~~Table 16-2~~ shall be used to indicate that among several possibilities one is recommended as particularly suitable, without mentioning or excluding others, or that a certain course of action is preferred but not necessarily required, or that (in the negative form) a certain possibility or course of action is deprecated but not prohibited.

Table 20-2 Recommendation

Auxiliary	Equivalent expressions for use in exceptional cases
should	it is recommended that ought to
should not	it is not recommended that ought not to

20.3 Permission

The auxiliaries shown in Table 20-3~~Table 16-3~~ are used to indicate a course of action permissible within the limits of the specification.

Table 20-3 Permission

Auxiliary	Equivalent expressions for use in exceptional cases
may	is permitted is allowed is permissible
need not	it is not required that no … is required

Do not use “possible” or “impossible” in this context. Do not use “can” instead of “may” in this context.

“May” signifies permission expressed by the standard, whereas “can” refers to the ability of a user of the standard or to a possibility open to him.

20.4 Possibility and Capability

The auxiliaries shown in Table 20-4~~Table 16-4~~ are used for statements of possibility and capability, whether material, physical or causal.

Table 20-4 Possibility and Capability

Auxiliary	Equivalent expressions for use in exceptional cases
can	be able to there is a possibility of it is possible to
cannot	be unable to there is no possibility of it is not possible to