Systems Alliance

VPP-3.4: Instrument Driver Programmatic Developer Interface Specification

Revision 2.5

April 14, 2008

VPP-3.4 Revision History

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

Revision 1.0, February 21, 1995

This edition reflects the first revision of this specification.

Revision 1.1, June 1, 1995

This edition includes error corrections, clarifications, and modifications to simplify writing drivers.

Revision 2.0, November 27, 1995

This edition reflects a restructuring of the document into a general specification section and framework specific sections.

Revision 2.1, February 2, 1996

This edition reflects error corrections, clarifications, and modifications.

Revision 2.2, December 4, 1998

This edition includes the addition of ViAttr and ViConstString to the Compatible Data Types Table. Information regarding contacting the alliance was updated. References to the VPP-5 Component Knowledge Base specification, which was obsoleted by the alliance, were removed.

Revision 2.3, March 17, 2000

This edition includes a recommendation and observation regarding building conventions for both the WIN95 and WINNT frameworks. Changed reference to HP VEE to Agilent VEE.

Revision 2.4, October 30, 2006

This edition adds a permission to use 64-bit integers to support instruments which use numbers greater than the range of a 32-bit integer.

Revision 2.5, February 14, 2008

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

Revision 2.5, April 14, 2008

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

Contents

Figures

Tables

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

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

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

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

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

Section 2 Overview of the Instrument Driver Programmatic Developer Interface Specification

2.1 Introduction

This section introduces and summarizes the *Instrument Driver Programmatic Developer Interface Specification*. The Instrument Driver Technical Working Group wrote the specification and performed the technical work that the specification discusses.

Also included is general information that the reader may need in order to understand, interpret, and implement individual aspects of this specification. These aspects include the following:

- The objectives of this specification.
- The audience for this specification.
- The scope and organization of this specification.
- Application of this specification.
- References.
- Definitions of terms and acronyms.
- Conventions.

2.2 Objectives of This Specification

The *Instrument Driver Programmatic Developer Interface Specification* provides a set of standards for specifying the programmatic interface to an instrument driver such that applications written in different Application Development Environments (ADEs) can interface with the driver.

High-quality, turn-key instrument drivers offer tremendous benefits to end users of instrumentation systems. Instrument drivers can get users started very quickly and can save them development time and cost. Instrument drivers that are open and well documented let end users customize their operation in order to fine tune system performance. Using instrument drivers that are modular can result in easier system debugging and maintenance, instrument interchangeability within a system, a foundation for easy expansion, and the ability to upgrade. These benefits can translate into substantial savings in time and money over the life of a system.

A single standard architecture can have the following benefits:

- Streamline the instrument driver development process for the benefit of instrument driver developers.
- Improve the consistency of instrument drivers from all vendors for the benefit of end users.
- Improve the quality of the drivers.
- Minimize duplicated effort in the industry.
- Improve ease of use for end users by providing a consistent methodology for using instrument drivers from a variety of vendors.
- A single instrument driver can be used by different Application Development Environments.
- Instrument drivers from several vendors can be combined in a single user application.

2.3 Audience for This Specification

This specification has two audiences. The first audience consists of instrument driver developers: instrument vendors, system integrators, and end users, who want to implement instrument driver software that is compliant with this specification. The second audience consists of instrumentation end users and application programmers who want to implement applications that use instrument drivers compliant with this specification.

2.4 Scope and Organization of This Specification

This specification is organized in sections. Each section discusses a particular aspect of the VXI*plug&play* Systems Alliance standard for instrument drivers.

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

Section 2 summarizes this specification and discusses its objectives, scope and organization, application, references, definition of terms, acronyms, and conventions.

Section 3 presents the general requirements for the instrument driver Programmatic Developer interface for all frameworks. These requirements define the interface between the instrument drivers and the most common Application Development Environments. Information is also presented on how some of the specific restrictions were determined.

Sections 4 and 5 define the framework-specific requirements for the instrument driver programmatic developer interface for the supported frameworks. Table 2-1 lists the supported frameworks and sections defining them.

Table 2-1. Framework Specific Sections

Section	**Framework**
Section 4	WINNT framework
Section 5	G frameworks

2.5 Application of This Specification

This specification is intended to be used by developers of VXI*plug&play* instrument drivers. It is also useful as a reference for end users of VXI*plug&play* instrument drivers. This specification is intended to be used in conjunction with the *Instrument Driver Functional Body* (VPP-3.2) and the *VISA Specifications* (VPP-4.X). These related specifications describe the implementation details for specific instrument drivers that are used with specific system frameworks. VXI*plug&play* instrument drivers developed in accordance with these specifications can be used in a wide variety of higher-level software environments, as described in the VXI*plug&play System Frameworks Specification* (VPP-2).

2.6 References

Several other VXI*plug&play* Systems Alliance documents and specifications are related to this specification. These other related documents include the following:

- VPP-1 *Charter Document*
- VPP-2 *System Frameworks Specification*
- VPP-3.1 *Instrument Drivers Architecture and Design Specification*
- VPP-3.2 *Instrument Driver Functional Body Specification*
- VPP-3.3 *Instrument Driver Interactive Developer Interface Specification*
- VPP-4.*x* *Virtual Instrument Software Architecture Specifications*
- VPP-6 *Installation and Packaging Specification*
- VPP-7 *Soft Front Panel Specification*
- VPP-9 *Instrument Vendor Abbreviations*

2.7 Definitions of Terms and Acronyms

The following are some commonly used terms within this document:

• ADE	Application Development Environment
• Callback function	A function that the instrument driver is to call back when certain conditions occur, such as an interrupt.
• Compatible	A programming feature, mechanism or capability that is available in all of the target ADEs.
• DLL	Dynamic Link Library.
• Exported function	A function that is accessible from outside the instrument driver.
• External interface	The set of exported functions from the instrument driver.
• Instrument driver	Library of functions for controlling a specific instrument.
• LabVIEW	Graphical programming ADE.
• LabWindows/CVI	C-based ADE.
• Agilent VEE	Graphical programming ADE.
• Language-Specific	A programming feature, mechanism or capability that is not available in all of the target ADEs.
• VISA	Virtual Instrument Software Architecture.

2.8 Conventions

The following headings appear on paragraphs throughout this specification. These headings give special meaning to the paragraphs.

Rules must be followed to ensure compatibility with the system framework. A rule is characterized by the words **SHALL** or **SHALL NOT** in bold upper-case characters. These words are not used in this manner for any other purpose.

Recommendations contain advice to implementors. This advice affects the usability of the final device. Recommendations are included in this standard to draw attention to particular characteristics that the authors believe to be important to end-user success.

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

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

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

Section 3 General Requirements for the Instrument Driver Programmatic Developer Interface

3.1 Introduction

A VXI*plug&play* instrument driver is a set of high-level instrument functions that provides a programmatic and interactive interface to an instrument. This driver consists of a set of software modules that must interface with other software in the system to communicate with the instrument, higher-level software, and end users.

The instrument driver external interface model shows how an instrument driver interfaces to the rest of the system (see Figure 3-1). This document is concerned with specifying the interface between a user program and the instrument driver in such a way as to ensure that compatible Application Development Environments (ADEs) may be used to develop the user program. This section will specify the general requirements on the programmatic developer interface to the instrument driver.

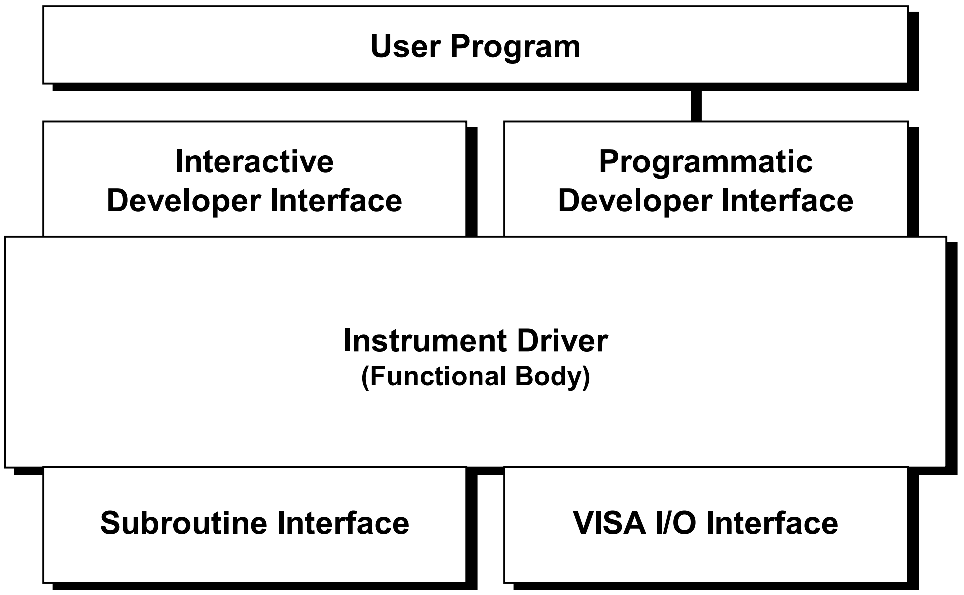
User Program
Interactive Developer Interface
Programmatic Developer Interface
Instrument Driver (Functional Body)
Subroutine Interface
VISA I/O Interface

Figure 3-1. Instrument Driver External Interface Model

The specifications in this document relate only to the external interface to the instrument driver. Functions that are used strictly within the driver, but are not directly accessible from outside the driver, are not limited by these restrictions. The terms *exported function* and *external interface* are used to denote those functions that are accessible by the ADEs.

This section covers the following areas:

- Compatible Application Development Environments.
- Instrument Driver Function Prototypes.
- Naming Conventions.
- Data Types.
- Parameter Conventions.
- Language-Specific Capabilities.
- Header Files.
- Bindings for the Required Functions.

3.2 Compatible Application Development Environments

3.2.1 Target ADEs

The purpose of this document is to specify the programmatic interface to instrument drivers in such a way that many Application Development Environments may be used to create user programs. To accomplish this it is necessary to first identify the most commonly used ADEs in instrument driver applications, and then to develop the specification such that those target ADEs are compatible. However, this is just the starting point; another goal in the development of this specification is to keep the programmatic interface sufficiently general to allow other ADEs to be compatible as well.

The target Application Development Environments for the Instrument Driver Programmatic Developer Interface are framework dependent. The supported ADEs for a specific framework can be found in the appropriate framework section of the VPP-2 document. The set of all the target ADEs from all the frameworks were used to determine the requirements for the external interface to instrument drivers. Drivers compliant with this standard will work with the specified and newer revisions of these ADEs.

3.2.2 Non-Compatible Capabilities

Providing sufficient capabilities in the programmatic interface to allow the creation of high-performance applications is another goal of this specification. This requires providing access to more advanced instrument driver capabilities. This goal conflicts with the previous goal of providing a very general interface.

To achieve both goals, the interface to the instrument driver is defined in terms of *compatible* functions and exceptions. The compatible functions are those functions that use only capabilities that are available in all of the target ADEs. This set of functions must provide a complete interface to the instrument (that is, access to all capabilities). The result is a programmatic interface to all the instrument capabilities that can be accessed from all compatible ADEs.

The exceptions are functions that use capabilities that are not available in all of the target ADEs. These functions provide the application developers with more advanced features such as callbacks and user-defined data types. Such capabilities are not compatible with all of the target ADEs, and throughout this document they will be referred to as *language-specific* capabilities. For each function that uses language-specific capabilities, an equivalent compatible function must be provided.

RULE 3.1

IF a VXI*plug&play* instrument driver contains a function that uses language-specific features, **THEN** it **SHALL** also provide equivalent functionality through compatible mechanisms (that is, mechanisms available in all of the target ADEs).

The language-specific functions and capabilities are not accessible to all ADEs, and it is important that they be clearly identified and documented as language-specific functions. This information must be specified in the instrument driver help file, header file (`PREFIX.h`), and function panel file (`PREFIX.fp`). It may also be appropriate to place these functions in a separate part of the function tree, or to indicate that they are language specific through the naming conventions for the function panel or class.

RULE 3.2

IF a VXI*plug&play* instrument driver contains functions that use language-specific features, **THEN** it **SHALL** clearly document these functions as such in the help file, header file (`PREFIX.h`), and function panel file (`PREFIX.fp`).

3.3 Instrument Driver Function Prototypes

3.3.1 General Form

The Application Development Environments obtain the required information about the instrument driver functions through the instrument driver header file. The programmatic interface to the instrument driver is specified by the function prototypes in this file. These function prototypes must conform to the standards of this document in order to ensure compatibility with the ADEs.

The general function prototype form is:

```
ViStatus _VI_FUNC <function name> ( <parameter> [, <parameter>]* ) ;
```

3.3.2 Return Type Specification

The first element of an instrument driver function prototype is the specification of the type of the value returned from the function. This must always be the type `ViStatus`. This type is defined in the header file `visatype.h`. Many of the standard return values for instrument driver functions are defined in the header files `visatype.h` and `vpptype.h`. These header files are described in Section 3.8, *Header Files*.

RULE 3.3

All VXI*plug&play* instrument driver functions **SHALL** return a value of type `ViStatus`.

3.3.3 Function Access Specifications

The function access specifications provide information to the Application Development Environments about access options and conventions. These specifications occur between the return value specification and the function name. The function access specifications must always be specified as the macro `_VI_FUNC`. This macro is defined in the header file `visatype.h`, and is dependent on the target ADE and the framework. ADEs and frameworks that do not require access modifiers for the function prototypes may define this value to be null.

RULE 3.4

All VXI*plug&play* instrument driver functions **SHALL** include the modifier macro `_VI_FUNC` before the function name.

3.3.4 Other Elements

The remaining elements of the function prototype include the function name, the parameter list, and the terminating semicolon. The function name must begin with the instrument prefix, as specified in VPP-3.1. Additional restrictions and requirements on the function name are defined in Section 3.4, *Naming Conventions*. The specifications for the parameter list are covered also in this section and Section 3.6, *Parameter Conventions.*

3.4 Naming Conventions

3.4.1 Introduction

The naming conventions for VXI*plug&play* instrument drivers specify the requirements and restrictions on function names, parameter and variable names, type names, and pre-processor macros. The naming conventions are provided to ensure compatibility with as many Application Development Environments as possible. Naming conventions include such items as case sensitivity, legal characters, name lengths, multi-word name separation, and so on.

3.4.2 Legal Characters and Name Length

The most fundamental aspects of naming conventions to consider are the set of legal characters in a name and the name length restrictions. These limitations are imposed by the Application Development Environments. To ensure compatibility with as many ADEs as possible, the most restrictive requirements from the set of target ADEs are used to establish these specifications.

- All external names (that is, names visible to the ADEs) must begin with an alphabetic character.
- Names may only include the alphanumeric characters and the underscore character.
- All external names must be restricted to 255 characters in length, all of which are significant in determining the uniqueness of the name. This requirement was dictated by the limitations of Visual Basic.

RULE 3.5

All external names in a VXI*plug&play* instrument driver **SHALL** begin with an alphabetic character (that is, the characters `a-z` and `A-Z`).

RULE 3.6

All external names in a VXI*plug&play* instrument driver **SHALL** contain only the characters `a-z`, `A-Z`, `0-9`, and `_`.

RULE 3.7

All external names in a VXI*plug&play* instrument driver **SHALL** be limited to 255 characters in length, all of which are significant.

3.4.3 Name Collisions

A name collision will occur in an application program if it includes multiple instrument drivers that define the same globally visible name. This may cause an error that the Application Development Environment can detect, resulting in a failure to build or run the application.

Another much more serious possibility is that the ADE does not detect the collision, resulting in an error within the application at run-time. This type of error may cause unpredictable results ranging from incorrect data from the measurement system, to physical damage to a device under test or test instrument. To ensure that this type of error cannot occur in application programs, it is necessary to require that all globally visible names in an instrument driver (for example, function names, type names, and macro names) are unique. This is accomplished by beginning each name with the instrument prefix and an underscore for a separator. Notice that parameter names are visible only within the scope of the function prototype and not globally visible; thus, they are not subject to name collisions.

RULE 3.8

All globally visible names in a VXI*plug&play* instrument driver **SHALL** begin with the instrument prefix followed by an underscore. This includes, but is not limited to, function names, type names, and macro names.

3.4.4 Case Sensitivity

Case sensitivity is an important consideration in naming conventions. Not all Application Development Environments support case sensitivity; therefore, it is important that case sensitivity not be a determining factor in the resolution of names within an instrument driver. Character case may be used to improve readability or for connotation of meaning, but should not be used for determining the uniqueness of names.

RULE 3.9

VXI*plug&play* instrument drivers **SHALL NOT** use case sensitivity to determine the uniqueness of names.

3.4.5 Case Conventions

Character case is often used to connote the meaning of a name. The following conventions are used in the VISA specifications (VPP-4.*x*) to connote the meaning of names.

- A name in all uppercase is a macro name. Example: `VI_ATTR_TERMCHAR`
- A name with the first character of each word in upper-case and all subsequent letters in lowercase represents a type name or variable name. Example: `ViEventInfo`
- A name starting with a lowercase character for the first word and uppercase for the first character of subsequent words is a function or variable name.
  Example: `viAssertTrigger`

Character case may also be used to improve the readability of names in programs. There are two standard mechanisms for separating the words in multi-word names— using underscores for separation, and using an upper-case character for the first character of each word. Both mechanisms are used within the VISA specifications. The underscore method must be used in macro names if they are always in upper-case characters. VISA uses this approach for macros and then uses an upper-case character for the first letter of each word for all other names. This has the advantage of resulting in shorter names than the underscore method, but results in a less readable name. An example of several methods is shown below:

`settriggerandreaddata`	not recommended
`setTriggerAndReadData`	fairly readable
`set_trigger_and_read_data`	the most readable, but longest

3.4.6 Recommended Conventions

The VXI*plug&play* instrument driver specifications do not require that any of these conventions be used in instrument drivers. Different conventions are used in the instrument driver specifications (VPP-3.1 and VPP-3.2) and the VISA specifications (VPP-4.*x*). However, to encourage consistency in the naming conventions used in instrument drivers some recommendations are provided.

The recommended naming convention for instrument drivers is to use the mixed-case mechanism for function, parameter and type names, and to use all upper-case characters with underscore separators for macro names. Function and parameter names should use a lower-case character for the first character of the first word, while type names should use an upper-case character. Function, type, and macro names should begin with the instrument prefix in lower-case, followed by an underscore.

The examples below show the recommended naming conventions.

`hpe1413a_triggerAndReadData`	function name
`triggerSource`	parameter name
`tkvx4750_MySrqHandler`	type name
`rd2351_TRIGGER_EXT`	macro name

RECOMMENDATION 3.1

Function names in an instrument driver should consist of the instrument prefix in lower-case, an underscore separator, and the remainder of the name specified using lower-case characters, with an upper-case character as the first character of all but the first word.

RECOMMENDATION 3.2

Parameter names in an instrument driver function should use lower-case characters, with an upper-case character for the first character of all but the first word.

RECOMMENDATION 3.3

Type names in an instrument driver should consist of the instrument prefix in lower-case, an underscore separator, and the remainder of the name specified using lower-case characters with an upper-case character for the first character of each word.

RECOMMENDATION 3.4

Macro names in an instrument driver should consist of the instrument prefix in lower-case, an underscore separator, and the remainder of the name specified using all upper-case characters with underscore characters as separators.

3.4.7 Name Consistency

When character case is used to either improve readability or connote some meaning, it is important that it be used consistently. This has two manifestations within an instrument driver.

- For a particular name everywhere within an instrument driver.
- For all names of the same kind (for example, a macro, type, variable or function), throughout an instrument driver.

RECOMMENDATION 3.5

All names within a VXI*plug&play* instrument driver should use consistent naming conventions.

OBSERVATION 3.1

The required functions in a VXI*plug&play* instrument driver do not use the naming conventions recommended in this document to ensure backward compatibility for applications written to the VPP-3.1 specification.

3.5 Data Types

3.5.1 Compatible Types

The data types allowed in an instrument driver function prototype fall into two classes—compatible types and language-specific types. The compatible types are the data types that are supported by the common Application Development Environments. Table 3-1 specifies the list of compatible types. These types are defined in the VISA header file `visatype.h`.

Table 3-1. Compatible Data Types

Type Name	**Direction**	**Definition**
`ViBoolean`	IN	Boolean value
`ViPBoolean`	OUT	Pointer to a `ViBoolean` value
`ViBoolean[]`	IN/OUT	Pointer to an array of `ViBoolean` values
`ViInt16`	IN	Signed 16-bit integer
`ViPInt16`	OUT	Pointer to a `ViInt16` value
`ViInt16[]`	IN/OUT	Pointer to an array of `ViInt16` values
`ViInt32`	IN	Signed 32-bit integer
`ViPInt32`	OUT	Pointer to a `ViInt32` value
`ViInt32[]`	IN/OUT	Pointer to an array of `ViInt32` values
`ViInt64`	IN	Signed 64-bit integer
`ViPInt64`	OUT	Pointer to a `ViInt64` value
`ViInt64[]`	IN/OUT	Pointer to an array of `ViInt64` values
`ViReal32[]`	IN/OUT	Pointer to an array of `ViReal32` values
`ViReal64`	IN	64-bit floating point number
`ViPReal64`	OUT	Pointer to a `ViReal64` value
`ViReal64[]`	IN/OUT	Pointer to an array of `ViReal64` values
`ViString`	IN	Pointer to a C string
`ViPString`	OUT	Pointer to a C string
`ViChar[]`	IN/OUT	Pointer to a C string
`ViRsrc`	IN	A VISA resource descriptor `(ViString)`
`ViPRsrc`	OUT	Pointer to a `ViRsrc` value `(ViPString)`
`ViSession`	IN	A VISA session handle
`ViPSession`	OUT	Pointer to a `ViSession`
`ViSession[]`	IN/OUT	Pointer to an array of `ViSession` values
`ViStatus`	IN	A VISA return status type
`ViConstString`	IN	A constant C string
`ViAttr`	IN	An attribute ID

PERMISSION 3.1

You **MAY** use the `ViA<Type>` datatypes in place of the `Vi<Type>[]` datatypes with the exception of `ViChar[]`.

RULE 3.10

All exported functions from a VXI*plug&play* instrument driver **SHALL** use only the compatible data types defined in Table 3-1, **EXCEPT** language-specific functions as defined in Section 3.7, *Language-Specific Capabilities*.

3.5.2 Type Limitations

The list of types allowed in instrument driver functions excludes some standard data types such as unsigned integers, enumerated types, and structures. These are excluded because they are not available in all the target Application Development Environments. A workaround for these types exists for most of the ADEs. Unsigned values can often be cast to the equivalent size signed values. Structures may be broken into separate parameters, or passed as arrays if homogeneous in type. Enumerated types may be passed as the type `ViInt16` with appropriate type casting on each end. Use of 64-bit integers in Visual Basic 6 is not supported. Therefore, instrument drivers that use ViInt64 data types cannot be used in Visual Basic 6 applications.

RECOMMENDATION 3.6

Enumerated types should be passed to instrument driver functions as the type `ViInt16`**.**

The output string types (`ViPString` and `ViPRsrc`) when used as parameters are restricted to a maximum of 61,440 characters (including the null terminator). This limit is imposed by Agilent VEE.

RULE 3.11

Output string parameters (`ViPString` and `ViPRsrc`) in a VXI*plug&play* instrument driver **SHALL** be limited to 61,440 characters in length (including the null terminator).

3.5.3 Type Usage

The types specified in Table 3-1 can be classified into three categories—scalar types used for input parameters (IN), scalar types used for output parameters (OUT), and array types that may be input, output or both (IN/OUT). These types are primarily defined as a triplet—a base type for use as an input parameter, a version of the type for output, and the array forms of the type. The input type is defined first, as a scalar type. Then the output and array forms are defined as pointers to this type. A naming convention is used to indicate that the types are related, and to indicate their usage. All the types begin with an initial `Vi` prefix (indicating that these are VISA types) and end with the name of the base (input) type. The output type includes an upper-case `P` (originally for *pointer*) after the `Vi` prefix, and one array type includes an upper-case `A` (for *array*) after the `Vi` prefix. The other form of array type uses the scalar input type name followed by square brackets (`[]`) to indicate an array of the base type.

The three types are defined and named to connote their purpose and direction to the application programmer. The scalar input types are passed by value, which means that constants and expressions may be used as the actual parameters. This also assures the application programmer that the driver function will not change the value of the actual parameter. The scalar output types (`ViP<Type>`) indicate that a single value will be returned from the function into the parameter. These types are passed by reference, and require that a variable be specified as the actual parameter. The array types (`ViA<Type>` and `Vi<Type>[]`) are also passed by reference, but these types indicate that multiple values may be returned from the function. The array types may also be used to pass values into the function, thus, their designation as input/output types. The direction of data flow for an array can be determined only from the documentation of the function. Conventions for specifying or determining the size of arrays are described in Section 3.6.5, *Arrays and Strings as Parameters*, and conventions for managing the memory for array parameters are described in Section 3.6.6, *Data Buffers*.

The direction column of Table 3-1 indicates the usage model for the supported types. The declaration of input parameters is straight-forward: the input types are used. For scalar output parameters there are two ways to declare the parameter—by using the `ViP<Type>` types, or by explicitly declaring the base (input) type by reference (`Vi<Type> *`). The preferred method is to use the pre-defined types, `ViP<Type>`. The following examples show two equivalent explicit declarations.

```
ViPInt32 myOutputParameter
ViInt32 * myOutputParameter
```

RULE 3.12

Scalar output parameters in a VXI*plug&play* instrument driver function **SHALL** be declared using one of three forms:

```
ViP<Type>  parmName
Vi<Type> * parmName
```

RECOMMENDATION 3.7

Scalar output parameters in a VXI*plug&play* instrument driver function should be declared using the `ViP<Type>` (OUT) types from Table 3-1.

RECOMMENDATION 3.8

The scalar output types, for example `ViP<Type>`, are output datatypes. Therefore, VXI*plug&play* instrument drivers should not reference the initial value of these parameters. IN/OUT parameters should use `Vi<Type>[]` types.

The string types `ViString`, `ViPString`, and `ViChar[]` are anomalies to the conventions mentioned above. There are two ways to view a string—as a scalar data type, or as an array of characters. At the high level, a string can be viewed as a scalar data type. Many ADEs support constants and expressions for strings, just like any other scalar type. To support this view of strings as scalar types, two identical types are defined for string parameters—one for use as an input parameter (`ViString`), and the other for the corresponding output parameter (`ViPString`). These string types are always passed by reference. Because the input string type (`ViString`) is passed by reference, rather than by value, a driver function could change the value of the parameter. This would cause an application error because application developers do not expect input-only (pass by value) parameters to change. To provide the application programmer with *pass by value* semantics for the `ViString` type, driver functions are not allowed to change the value of a `ViString` parameter.

RULE 3.13

A VXI*plug&play* instrument driver function **SHALL NOT** modify a parameter of type `ViString`.

OBSERVATION 3.2

A VXI*plug&play* instrument driver function cannot modify a parameter of type `ViConstString`.

At the primitive level, strings are arrays of characters. This means that the conventions for specifying the size of arrays and for managing the memory for arrays, also apply to strings. Arrays are passed to a function by reference and can be used as an input parameter, output parameter or both. To support this view of a string as an array, the form `ViChar[]` is defined. This form is also useful for indicating to an ADE that memory must be allocated for the return value (as described below).

Some ADEs, such as LabWindows®/CVI, must be able to distinguish between an array parameter and an output scalar parameter. This distinction is important because it indicates when an array or data buffer must be passed into a function or automatically allocated to allow multiple values to be returned. For this reason, we explicitly disallow the use of any form of scalar output parameter (that is, the `ViP<Type>` types or the explicit equivalents) to be used for passing arrays. Array parameters may be declared in one of two ways—by using the array types (`ViA<Type>`), or by declaring an array of some base type (`Vi<Type>[]`). Some ADEs key off of the "`[]`" symbols to determine when an array parameter is being specified. For this reason, the `Vi<Type>[]` notation is the recommended form for declaring array parameters. The examples below show the complete declarations for some array parameters.

`ViInt32 myIntArray[]`	Array of `ViInt32`
`ViChar  myString[]`	String (array of char)

The types `ViA<Type>` and `ViPString` may be used to declare array parameters and output strings, respectively, in the instrument driver header file. When these forms are used, steps must be taken to ensure that memory is allocated in the Application Development Environment. This is accomplished by forcing the parameter type entries in the function panel file (`PREFIX.fp`) to use the `Vi<Type>[]` form. This flags the parameter as an array type to ADEs that read the function panel file.

RULE 3.14

Array parameters in a VXI*plug&play* instrument driver function **SHALL** be declared using one of two forms:

```
ViA<Type> parmName
Vi<Type> parmName[]
```

RECOMMENDATION 3.9

Array parameters (including output strings) in a VXI*plug&play* instrument driver function should be declared using the `Vi<Type>[]` forms from Table 3-1 (for example, `ViInt16 myArray[]` for an array of `ViInt16` values).

3.6 Parameter Conventions

3.6.1 Introduction

The parameter passing conventions for an instrument driver specify the requirements and restrictions on the parameters to the functions. Some of these issues are defined earlier in this document, such as the allowable types of parameters as specified in Section 3.5. Other issues are specified in the framework specific sections of this document, or in other documents. This section will define general specifications such as the number of parameters, parameter naming, parameter passing, and so on.

3.6.2 The ViSession Parameter

Many VXI*plug&play* instrument driver functions require a session handle as an input parameter to identify the target instrument. To provide consistency in the use model for the application developer it is necessary to define the type and location for this parameter. As such, we require that the session handle for such functions be defined as the type `ViSession` and that it be the first parameter in the argument list.

RULE 3.15

Every VXI*plug&play* instrument driver function that requires a session handle to identify the target instrument **SHALL** define the parameter to be of type `ViSession` and specify it as the first parameter in the argument list.

RULE 3.16

`ViSession` datatypes **SHALL** be used only for session handles returned from VISA.

3.6.3 Named Parameters

One very important aspect of parameter conventions deals with whether the parameter name is included in the function prototype. This is not required in the function prototype specifications for the C language, but it is essential for the Application Development Environments. The function panel file (`PREFIX.fp`) does contain this information, but it is not in a form that is readily accessible to most ADEs. Thus, it will be required that the name for every parameter be included in the function prototype.

RULE 3.17

The function prototypes within a VXI*plug&play* instrument driver header file **SHALL** include the name of each parameter.

3.6.4 Number of Parameters

Restrictions on the number of parameters in a driver function are part of the parameter conventions. In order to have a consistent, well-specified interface to each function it is important that all parameters be separately specified. This means that instrument driver functions should not be designed to accept an arbitrary number of parameters, nor to specify optional parameters. Either of these mechanisms would make it difficult to develop the interactive interface to the instrument driver and the function panel for the driver function.

The number of parameters allowed for a driver function will be limited to 18. This limit is constrained by the LabVIEW Application Development Environment. LabVIEW allows a maximum of 20 terminals on an icon, including the input and output error cluster terminals. Thus, a maximum of 18 terminals can be supported for the function parameters. However, to improve the usability of instrument drivers and the readability of application programs, it is desirable to minimize the number of parameters; therefore, a practical limit of eight is suggested.

RULE 3.18

A VXI*plug&play* instrument driver function **SHALL NOT** use variable parameter lists nor optional parameters.

RULE 3.19

VXI*plug&play* instrument driver functions **SHALL** be limited to a maximum of 18 parameters.

RECOMMENDATION 3.10

A VXI*plug&play* instrument driver function should be limited to a maximum of eight parameters.

3.6.5 Arrays and Strings as Parameters

Arrays are passed to an instrument driver function to provide input data to the function, receive data from the function, or both. Knowledge of the size of an array is necessary for proper communication between the application program and the driver function. Before discussing mechanisms for communicating the size of arrays, the restrictions on the maximum size of an array must be considered. Each Application Development Environment has its own limitations on the size of arrays that can be accessed. All of the target ADEs support the development of programs that can use extremely large arrays. Many are limited only by the maximum value of a signed 32-bit integer (that is, 2 GB). .

The actual limit on the size of arrays that may be used in an application program generally is not imposed by the ADEs, but rather by the amount of virtual memory available in the system. This amount is framework dependent. .

Arrays and strings are always passed to an instrument driver function by reference. When used as an output parameter, they both appear to the driver function as indeterminate size arrays. As input parameters, a distinction can be made from the perspective of the instrument driver function. The difference is that the end or size of an input string (`ViString`) can be determined from the string itself (because of the null terminator); this is not true of input arrays. Thus, a convention is required by either the application program or the instrument driver function for determining the size of arrays and output strings (`ViPString` and `ViChar[]`).

One convention for determining the size of an array or string is to include a separate parameter in the function call that specifies the size. This allows the driver function to determine the actual size, and is the most robust method because it allows for different sizes to be used in different invocations. An alternative is to include a macro in the header file to specify the maximum size of each array or output string that each function can handle. This allows the application program to allocate a sufficiently large array or string. However, this approach works only when the array or string that is passed to a function is always the same size, and always the maximum size. Another alternative is to document the maximum array or string size in the help files. This alternative is similar to the macro approach, and has the same disadvantage.

RULE 3.20

For each array (`ViA<Type>` and `Vi<Type>[]`) or output string (`ViPString` and `ViChar[]`) parameter, an instrument driver function **SHALL** provide either a parameter specifying the size, a macro specifying the maximum size, or documentation in the help file specifying the maximum size.

RECOMMENDATION 3.11

An instrument driver function should include a parameter to specify the size of each array or output string parameter. This parameter should indicate the actual size of an input parameter, and the maximum size of an output parameter. The size should be specified in terms of the number of elements for an array or the number of characters for a string.

3.6.6 Data Buffers

Parameter conventions are needed for controlling the memory management of the data buffers used for passing arrays and strings between the application program and the instrument driver functions. In order to reduce the likelihood of dynamic memory access problems, it is important that all data buffers are defined in the calling context and passed into the instrument driver functions. This is one reason a distinction is made between the scalar output types (`ViP<Type>`) and the array types (`ViA<Type>` and `Vi<Type>[]`). Although these are in fact the same type (that is, both pointers to the base type), the naming convention clearly indicates when an array must be passed in.

Instrument driver functions are allowed to allocate dynamic memory. However, the instrument driver must be responsible for managing its own memory allocation and deallocation; therefore, it should not pass a pointer to a locally allocated buffer back to the calling context. This can result in dangling pointers that can generate serious system errors in the application. Another problem that exists when mixing the memory allocation and deallocation conventions of the application and instrument driver is the potential for causing memory leaks.

RULE 3.21

All memory buffers required for returning data from a VXI*plug&play* instrument driver function **SHALL** be passed into the function.

RULE 3.22

A VXI*plug&play* instrument driver function **SHALL NOT** pass allocated memory buffers back to the calling context.

RULE 3.23

All VXI*plug&play* instrument drivers **SHALL** be responsible for their own memory allocation and deallocation.

3.7 Language-Specific Capabilities

3.7.1 Introduction

This section discusses the mechanisms for handling language-specific capabilities within instrument driver functions. Rule 3.1 places the requirement on the instrument driver developer to provide an equivalent compatible function for every function that is defined using language-specific capabilities. There are currently three language-specific capabilities that have been identified as potentially useful in instrument drivers. These cases will be explicitly defined and discussed. All other cases will be discussed in terms of exceptions, and are up to the driver developer to define.

These language-specific capabilities are provided as extensions to the standard driver interface. They are considered extensions because such capabilities are not available in all Application Development Environments. In order for all ADEs to be able to use an instrument driver header file that contains language-specific capabilities, it is necessary to provide a mechanism that enables these capabilities only for ADEs that support them. This is accomplished with the standard C preprocessor directives `#ifdef ..#endif`, using a symbolic name appropriate for the capability.

3.7.2 Callback Mechanisms

3.7.2.1 Introduction

A callback mechanism is a useful capability within an instrument driver. It allows for an event handling function within the application to be called directly when that event is detected—either within the instrument driver, or within another software module such as the VISA library. This provides a very efficient mechanism for handling events such as interrupts. The symbolic name `INSTR_CALLBACKS` is used to conditionally include the callback definitions. This symbol should be defined only for ADEs that can implement a callback mechanism.

RULE 3.24

IF a VXI*plug&play* instrument driver header file includes type definitions and function prototypes that provide a callback mechanism, **THEN** it **SHALL** conditionally include those definitions based on the symbol `INSTR_CALLBACKS`.

OBSERVATION 3.3

If an instrument driver includes a function that provides a callback mechanism, then it must also provide equivalent functionality through a compatible function utilizing either a polling or blocking mechanism.

3.7.2.2 Operation of Callbacks

A callback mechanism provides a means for the instrument driver or some other software module (such as VISA) to directly call an event handler within the application program when an event is detected. This event can be an interrupt from the operating system, a user triggered event, or a condition detected in the driver.

The operation of the callback mechanism involves two steps—installation of the callback function and the calling of the handler upon detection of the trigger event. The first step involves passing a pointer to an application function (the handler is also known as the callback function) into the instrument driver function that provides the callback mechanism (the callback setup function). This function either maintains the pointer within the driver itself or installs it in another software module (such as VISA) by passing it into another callback setup function.

Detection of the trigger event is handled by the function in which the callback was installed. The callback may be triggered by an outside event, another callback function, or some condition occurring in the software module. When the trigger event is detected, the installed handler (that is, the saved pointer to the callback function) is called.

The most common type of callback used in an instrument driver is triggered by an event in the I/O system, such as an SRQ. There are two ways to handle this type of callback. The first is to have the driver function install the callback directly into VISA. When the event occurs, VISA calls directly into the application program handler. This provides a very efficient mechanism since the instrument driver is not involved; however, it can be prone to errors if the handler is not properly written. For example, if the handler is supposed to explicitly clear the trigger event, but does not, it can result in an infinite loop and a hung system.

The second way to handle a callback is to have the driver maintain the callback pointer and to install its own handler into VISA. When the trigger event occurs, VISA calls back into the handler in the instrument driver. The driver handler can then perform some set-up operations before calling the application handler, and can also pass additional information into the handler. The driver handler can also check and respond to a return value from the application handler and finally clear the trigger if needed before returning to VISA. This method does involve additional overhead in the instrument driver handler, but it provides a means for better protecting the user and is the recommended approach.

RECOMMENDATION 3.12

A VXI*plug&play* instrument driver that implements callback functions should provide its own callback handler to catch the VISA callback and then call the application function.

3.7.2.3 Callback Example

The following code segment shows an example of the definition of a callback function type and an instrument driver function that provides a callback mechanism.

```
#ifdef INSTR_CALLBACKS

typedef void ( * _VI_FUNCH vxi1234a_ReadHandler ) ( ViInt32
value,
     ViInt16 reason );

ViStatus _VI_FUNC vxi1234a_readAndCallback ( ViSession vi,
     vxi1234a_ReadHandler readCallback );

#endif
```

The first statement is the type definition that specifies the interface of the function in the application program to be called back. This definition consists of five parts. The first part is the C `typedef` keyword that indicates a new type is being defined. The next part is the specification of the return value from the callback function. This return value is often specified as `void` indicating that no value is returned. However, a status value can be returned from the callback function. The next part of the definition is the pointer and access specification. The access specification should always be the macro `_VI_FUNCH`. The next element is the name of the type, and should follow the type naming conventions. The last part is the parameter list, that specifies the type and name for each parameter in the callback function.

Note: ***Because the callback function is part of the application program, and not the instrument driver, the interface for the callback function does not need to conform to the specifications in this document (that is, it can use any types for the parameters and return value).***

The second statement is the instrument driver function definition. As such, it resembles any other instrument driver function, with the exception that one of its parameters is a language-specific type that represents a pointer to the callback function. This driver function enables the callback mechanism by either maintaining the callback pointer within the instrument driver, or by installing it into a VISA function that supports callbacks.

3.7.3 String Arrays

String arrays can provide an efficient mechanism for communicating with certain types of instruments such as logic analyzers and switches. String arrays may be used in a language-specific function as long as a function that uses compatible types is also provided. Functions that use string arrays as parameters should be conditionally included based on the symbolic name `INSTR_STRING_ARRAYS`. This symbol should only be defined for ADEs that support string arrays as parameters. To promote the consistent use of string arrays in instrument drivers, the VISA header file `visatype.h` includes the following definition of a string array type:

```
typedef ViString ViAString;
```

RULE 3.25

IF a VXI*plug&play* instrument driver header file includes function prototypes that utilize string arrays (`ViAString`) as parameters, **THEN** it **SHALL** conditionally include those definitions based on the symbol `INSTR_STRING_ARRAYS`**.**

3.7.4 Other Language-Specific Capabilities

There may be other language-specific capabilities such as multi-dimensional arrays, which also provide highly efficient interfaces to an instrument driver. These capabilities can be accessed as extensions to the standard (compatible) interface to the driver. Instrument driver functions that use these language-specific capabilities must be conditionally included based on the symbolic name `INSTR_LANGUAGE_SPECIFIC`. Caution should be used when enabling these features, as multiple language-specific capabilities may be enabled simultaneously.

RULE 3.26

IF a VXI*plug&play* instrument driver header file includes function prototypes that utilize language-specific capabilities not explicitly defined in the previous sections (3.7.2 through 3.7.4), **THEN** it **SHALL** conditionally include those definitions based on the symbol `INSTR_LANGUAGE_SPECIFIC`**.**

3.8 Header Files

3.8.1 Introduction

This section defines the header files required by application programs in order to use a VXI*plug&play* instrument driver. The hierarchy of header files for a C-compatible application is shown in Figure 3-2. The application program includes one header file that represents the external interface to the instrument driver. This instrument driver header file includes the instrument driver specific type and macro definitions from the VISA header file `vpptype.h`. The file `vpptype.h` includes the VISA header file `visatype.h` that defines all the fundamental types and macros to interface with the VISA library.

Figure 3-2 also shows the header file hierarchy for Visual Basic applications. For Visual Basic applications, the header files are not included into the application, but instead either the header files are added to the project or the relevant parts of the header files are copied into a project file.

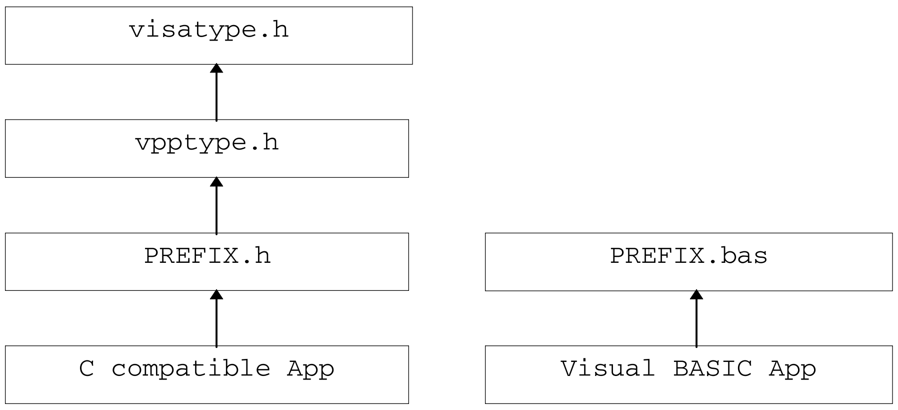

Figure 3-2. The Header File Hierarchy

3.8.2 The Visatype.h Header File

The base VISA header file is `visatype.h`. The location of this file is framework specific and can be found in the appropriate framework section of the VPP-6 document. This file contains the definitions of the compatible types defined in Table 3-1. This file also contains the following definitions that simplify the use of Boolean variables:

```
#define VI_TRUE     1
#define VI_FALSE    0
```

A listing of the file `visatype.h` can be found in VPP-4.3.2*: VISA Implementation Specification for Textual Languages*. The equivalent Visual Basic bindings can be found in the header file `visa.bas` or `visa32.bas`.

3.8.3 The Vpptype.h Header File

The instrument driver specific VISA header file is `vpptype.h`. The location of this file is framework specific and can be found in the appropriate framework section of the VPP-6 document. A listing of the file `vpptype.h` can be found in Appendix A of this document. The Visual Basic binding for this file, `vpptype.bas`, is included in Appendix B. The following elements are in `vpptype.h`:

- `#include <visatype.h>`
- `#define` specifications for some standard enumerated types
- `#define` specifications for the standard default return values

The following values are defined to simplify use of enumerated types representing a binary switch:

```
#define VI_ON    (1)
#define VI_OFF   (0)
```

The following values are defined to specify some of the standard default return values for warnings, as defined in VPP-3.1:

```
#define VI_WARN_NSUP_ID_QUERY     (0x3FFC0101L)
#define VI_WARN_NSUP_RESET        (0x3FFC0102L)
#define VI_WARN_NSUP_SELF_TEST    (0x3FFC0103L)
#define VI_WARN_NSUP_ERROR_QUERY  (0x3FFC0104L)
#define VI_WARN_NSUP_REV_QUERY    (0x3FFC0105L)
```

The following values are defined to specify some of the standard default error return values:

```
#define VI_ERROR_PARAMETER1       (_VI_ERROR+0xBFFC0001L)
#define VI_ERROR_PARAMETER2       (_VI_ERROR+0xBFFC0002L)
#define VI_ERROR_PARAMETER3       (_VI_ERROR+0xBFFC0003L)
#define VI_ERROR_PARAMETER4       (_VI_ERROR+0xBFFC0004L)
#define VI_ERROR_PARAMETER5       (_VI_ERROR+0xBFFC0005L)
#define VI_ERROR_PARAMETER6       (_VI_ERROR+0xBFFC0006L)
#define VI_ERROR_PARAMETER7       (_VI_ERROR+0xBFFC0007L)
#define VI_ERROR_PARAMETER8       (_VI_ERROR+0xBFFC0008L)
#define VI_ERROR_FAIL_ID_QUERY    (_VI_ERROR+0xBFFC0011L)
#define VI_ERROR_INV_RESPONSE     (_VI_ERROR+0xBFFC0012L)
```

As the interface requirements for instruments such as network analyzers become better understood, this file may include the definition of additional supported types such as complex numbers, arrays of complex numbers, and other definitions related to them.

3.8.4 The Instrument Driver Header File

3.8.4.1 Introduction

The instrument driver header file is used to provide information to the Application Development Environments on how to access the functions in the driver library. This file is scanned by some of the ADEs in order to obtain this information. As such, only the information necessary to access the driver functions should be included in this file. Any other extraneous specifications (for example, function prototypes for special support routines, data types used only within the driver functions and so on) that are needed by the C source file (`PREFIX.c`) should be included in other driver header files.

The instrument driver header file is `PREFIX.h`. The location of this file is framework specific and can be found in the appropriate framework section of the VPP-6 document. The instrument driver header file contains statements that fall into two categories— required preprocessor statements, and statements that specify the external interface to the driver.

3.8.4.2 Required Preprocessor Statements

The required preprocessor statements establish the program structure and define the compiler external naming conventions. The first set of preprocessor statements forms a conditional block around the other elements of the file. These statements protect the application program from errors due to multiple inclusions of the header file. This is accomplished using the C preprocessor directives `#ifndef .. #define ..#endif`. The symbolic name used to conditionally exclude the header file declarations is `__PREFIX_HEADER`, where `PREFIX` represents the instrument prefix.

RULE 3.27

Every VXI*plug&play* instrument driver header file **SHALL** contain preprocessor conditional statements to protect against errors due to multiple inclusions of the file. These statements have the form:

```
#ifndef __PREFIX_HEADER
#define __PREFIX_HEADER

/* all the other elements of the header file. */

#endif
```

The next required preprocessor statement specifies inclusion of the VISA instrument driver specific header file `vpptype.h`. The inclusion of this file makes all the appropriate VISA type definitions and macros available to the instrument driver.

RULE 3.28

Every VXI*plug&play* instrument driver header file **SHALL** include the `vpptype.h` header file (that is, `#include <vpptype.h>`).

RULE 3.29

VXI*plug&play* instrument driver header files **SHALL NOT** include the `visa.h` header files directly.

The last set of required preprocessor statements define the naming conventions that the C++ compiler uses for external names. By default, the C++ compiler generates *decorated* names for exported functions. These names include information on the type of the function parameters. To link programs written in languages other than C++ it is necessary to instruct the C++ compiler to generate standard C external names. This is accomplished by encapsulating all the function prototypes in the header file within the C++ statement block, `extern "C" { ... }`. However, these statements are valid only for the C++ compiler, and thus we must conditionally include them. The C++ compilers from the list of compatible ADEs automatically define one of two symbols to enable C++ conditional statements: `__cplusplus` and `__cplusplus__`. These symbols must be used in a preprocessor `#if .. #endif` block to conditionally include the `extern` block.

RULE 3.30

All VXI*plug&play* instrument driver header files **SHALL** include the C++ conditional specifications for C language naming conventions. These statements have the form:

```
#if defined(__cplusplus) || defined(__cplusplus__)
extern "C" {
#endif

     ...                       all the function prototypes in the header file

#if defined(__cplusplus) || defined(__cplusplus__)
}
#endif
```

3.8.4.3 The Driver External Interface

The external interface to the driver consists of four elements:

- The function prototypes for the instrument driver functions.
- Macro definitions for parameter values and ranges.
- Comments specifying usage information.
- Macros defining instrument-specific errors.

The function prototypes provide the ADEs with information on how to interface with the functions in the dynamically linked library. This includes return type specifications, access specifications, calling conventions, parameter type information, and parameter passing conventions. Without a function prototype, an ADE cannot call a driver function. Therefore, an instrument driver header file must specify a function prototype for every exported driver function.

RULE 3.31

Every VXI*plug&play* instrument driver header file **SHALL** contain the function prototypes for all functions exported to the Application Development Environments.

PERMISSION 3.2

A VXI*plug&play* instrument driver **MAY** include special purpose functions in the dynamically linked libraries that are not defined in the instrument driver header file.

Preprocessor macros and text comments can be used to simplify the interface between the application program and the instrument driver. The driver may define macros for all numeric parameters to indicate the allowed range of values. The application program uses these macros to set or test the limits of the parameters. Macros used for this purpose should indicate whether they represent the minimum or maximum value by including the word `MIN` or `MAX` at the end of the macro name. Alternatively, comments may be used to specify the limits of numeric parameters to the application developer. Comments are more appropriate than macros for specifying limits when they are dynamic (that is, the limits vary with other instrument settings).

The parameters to instrument driver functions often represent enumerated types. For example, the parameter `dvmFunction` might expect the values `0`, `1`, and `2` representing `ACV`, `DCV`, and `OHM`, respectively. It is much easier for the application developer to write a program using symbolic names rather than numeric values. Predefined macros should be specified in the instrument driver header file to represent the allowable values for enumerated types. These macros are also useful for making the application program more readable and maintainable. A macro defined to represent an enumerated value should include the enumeration name at the end of the macro name.

The instrument driver header file is viewed directly by the application programmer during development and should provide sufficient documentation for programming the instrument. Comments should be used liberally throughout the driver header file to document the purpose and operation of the driver and driver functions. The restrictions, limitations, and specifications for each parameter, as well as couplings between parameters, should be documented. Anything that is documented in the help files may also be documented by comments in the header file.

Each instrument driver may detect error conditions that are specific to that instrument or class of instruments. A range of values for the type `ViStatus` are reserved for these instrument-specific errors. This range is from `0xBFFC0800L` to `0xBFFC0FFFL`. To avoid compiler warnings regarding unsigned integer comparisons, these error codes should be defined as the value `_VI_ERROR` plus the desired error value with the most significant bit set to `0`. For example, to define the error code `VXI1234_ERROR_ILLEGAL_MODE` as `0xBFFC0834L`, use the following declaration:

```
#define VXI1234_ERROR_ILLEGAL_MODE  _VI_ERROR + 0x3FFC0834L
```

RECOMMENDATION 3.13

The instrument driver header file should include information about the maximum and minimum values of numeric parameters as `#define` statements or comments. For macro definitions, the name should begin with the instrument prefix, include the parameter name, and end with the name `MIN` or `MAX`.

RECOMMENDATION 3.14

The instrument driver header file should include `#define` statements specifying all the values for parameters representing enumerated types. The symbolic names for these macros should begin with the instrument prefix, include the parameter name, and end with a name describing the enumerated value.

RECOMMENDATION 3.15

The instrument driver header file should include comments describing the interface to the driver functions. This may include information on the operation of the function, the purpose and allowed values of parameters, the size and direction of arrays, and any other limitations and restrictions.

RECOMMENDATION 3.16

The instrument driver header file should include macro definitions for instrument-specific errors. The error values for these errors should be chosen in the range `0xBFFC0800L` to `0xBFFC0FFFL`. These error codes should be defined as the value `_VI_ERROR` plus the desired error value with the most significant bit set to `0`.

RULE 3.32

A VXI*plug&play* instrument driver header file **SHALL** include only the following types of statements:

- #ifndef .. #define .. #endif to prevent multiple inclusion
- #include statements
- #if .. #endif statements to specify C++ name generation
- function prototypes for all exported functions
- #define statements
- comments

- #ifdef .. #endif blocks for language-specific capabilities

An example instrument driver header file can be found in Appendix C of this specification.

3.8.5 The Visual Basic Header File

3.8.5.1 Introduction

A Visual Basic header file must be included with the instrument driver for all frameworks that support Visual Basic. This includes the WINNT framework. The Visual Basic header file is named `PREFIX.bas`. The location of this file is framework specific and can be found in the appropriate framework section of this document. The Visual Basic header file contains function, type, and constant declaration statements that can be copied into the application program. It must contain function declaration statements for all the compatible instrument driver functions, and constant declaration statements for all the associated macros. For functions that utilize language-specific capabilities that Visual Basic supports, the function declaration statement and any associated user-defined type statements and constant declarations must be included in the header file. It is the responsibility of the driver developer to determine if a Visual Basic binding can be defined for these capabilities.

3.8.5.2 Visual Basic Function Declarations

The Visual Basic header file for an instrument driver must contain function declarations for all the exported functions supported by Visual Basic. The Visual Basic function declaration statement takes the form:

```
Declare Function <function name> Lib "<dll name>"
     ( <parameter> [, <parameter>]* ) As Long
```

The `<function name>` field is the exported name of the function. The `<dll name>` field is the name of the instrument driver DLL; this must be `PREFIX.dll`. Notice that the return type is defined as the type `Long`. This is the appropriate Visual Basic type to represent a `ViStatus` type. The `<parameter>` specifications must include the parameter passing mode, the parameter name, and the parameter type.

Visual Basic defaults to passing parameters by reference; the `ByVal` keyword is used with numeric types to specify that the parameter is to be passed by value. The `ByVal` keyword is also used with Visual Basic `String` parameters to indicate that the string should be converted to a C string that is then passed by reference. The form of a parameter specification is:

```
    [ByVal] <parameter name> As <parameter type>
```

Table 3-2 shows the appropriate parameter passing mode and data type to use for each of the compatible data types from Table 3-1.

Table 3-2. Microsoft Visual Basic Type Equivalents

VISA Type	Visual Basic Type Declaration
`ViBoolean`	ByVal Integer
`ViPBoolean`	Integer
`ViABoolean`	Integer
`ViBoolean[]`	Integer
`ViInt16`	ByVal Integer
`ViPInt16`	Integer
`ViAInt16`	Integer
`ViInt16[]`	Integer
`ViInt32`	ByVal Long
`ViPInt32`	Long
`ViAInt32`	Long
`ViInt32[]`	Long
`ViReal32[]`	Float
`ViReal64`	ByVal Double
`ViPReal64`	Double
`ViAReal64`	Double
`ViReal64[]`	Double
`ViString`	ByVal String
`ViPString`	ByVal String
`ViChar[]`	ByVal String
`ViRsrc`	ByVal String
`ViPRsrc`	ByVal String
`ViSession`	ByVal Long
`ViPSession`	Long
`ViSession[]`	Long
`ViStatus`	ByVal Long

RULE 3.33

The Visual Basic header file for a VXI*plug&play* instrument driver **SHALL** contain the function declarations for all compatible functions, and for functions that utilize language-specific capabilities supported by Visual Basic.

3.8.5.3 Constant and Type Declarations

The Visual Basic header file does not require any type declaration statements for the compatible types from Table 3-1. Table 3-2, shows the equivalent Visual Basic types and parameter passing modes for these types. Type declaration statements are required in the Visual Basic header file for any language-specific types used by the driver functions that Visual Basic supports. These types and the appropriate Visual Basic bindings and declarations must be determined by the driver developer.

Constant declaration statements must be included along with the function declarations to provide a means for the application programmer to easily specify the values for enumerated types, and to test the limits of numeric parameters. These constant declarations are the Visual Basic binding equivalents for the macro definitions from the `PREFIX.h` file. These values are declared with the `global const` statement:

```
Global Const <macro name> = <value>
```

The `Global` keyword indicates that the constant will be global in scope. This may not be valid in all situations (for example, within a form), in which case the `Global` keyword should be deleted. The `<macro name>` field represents the name of the macro from the `PREFIX.h` file. This can be specified with the same name as the macro, in which case it will be a variant type based on the type of the value. Alternatively, the constant type can be explicitly defined by appending one of the suffixes from Table 3-3 to the end of the macro name.

Table 3-3. Microsoft Visual Basic Type Suffixes

Visual Basic Type	**Suffix**
Double	#
Integer	%
Long	&
String	$

The `<value>` field specifies the value of the constant. For numeric values, the appropriate suffix from Table 3-3 should be used, and for string values the string must be enclosed in double quotes. For `Integer` and `Long` numeric values, this field may be specified as a simple decimal value, or it may be specified in hexadecimal by putting the characters `&H` in front of the number. The `#` suffix is only needed to distinguish a `Double` value from an `Integer` or `Long` value. For example, the number `3.14` is implicitly a real number, but the number `1` is not. Thus, to specify the number `1` to be the real number `1.0`, it must be specified as `1#`.

Examples of some constant declarations are:

```
Global Const vxi1234a_COUPLING_AC = 0&       Variant Long in decimal
Global Const vxi1234a_COUPLING_DC = &H12&    Variant Long in hex
Global Const vxi1234a_UNITS       = "dBm"    Variant String
Global Const vxi1234a_RANGE       = 10#      Variant Double
Global Const vxi1234a_RANGE_MIN#  = 1.12     Explicit Double
Global Const vxi1234a_RANGE_MAX&  = 1000&    Explicit Long in decimal
Global Const vxi1234a_TRIGGER$    = "EXT"    Explicit String
```

An example of a Visual Basic header file for an instrument driver can be found in Appendix D of this specification.

RULE 3.34

The Visual Basic header file for a VXI*plug&play* instrument driver **SHALL** contain the language-specific type declarations and constant declarations associated with all the driver functions supported by Visual Basic.

Section 4
Instrument Driver Programmatic Developer Interface for the WINNT Framework

4.1 Introduction

This section defines the framework-specific requirements for the WINNT Framework. These specifications may be clarifications, enhancements, or restrictions on the corresponding requirements from the equivalent parts of Section 3.

4.2 Compatible Application Development Environments

The list of target Application Development Environments and supported versions for the WINNT Framework can be found in the WINNT Framework section of the VPP-2 document.

4.3 Instrument Driver Function Prototypes

The macro `_VI_FUNC` is used in the definition of every exported instrument driver call and specifies to the ADE how to access the function. `_VI_FUNC` is defined in the file `visatype.hc`. A listing of the file `visatype.h` can be found in VPP-4.3.2*: VISA Implementation Specification for Textual Languages*.

4.4 Naming Conventions

The general naming conventions defined in Section 3.4 are all valid and sufficient for the WINNT Framework.

4.5 Data Types

The types specified in Table 3-1 as pointer types are defined using the `_VI_PTR` macro from the header file `visatype.h`. A listing of the file `visatype.h` can be found in VPP-4.3.2*: VISA Implementation Specification for Textual Languages*.

4.6 Parameter Conventions

WINNT framework drivers must be compiled and built into a 32-bit DLL. Because all pointers are long addresses, obsolete parameter access specifications, such as `_VI_FAR` and `_VI_PTR`, are not required. .

4.7 Language-Specific Capabilities

A listing of the file `visatype.h` can be found in VPP-4.3.2*: VISA Implementation Specification for Textual Languages*.

4.8 Header Files

The list of locations of the header files and other important driver files for the WINNT Framework can be found in the WINNT Framework section of the VPP-6 document.

4.9 Build Conventions

RECOMMENDATION 4.1

An instrument driver that needs any external library should link to a runtime version of that library if available. For example, an instrument driver should link the import library for the C runtime library rather than the static library implementation itself.

OBSERVATION 4.1

This is important because libraries such as Microsoft C library uses resources that, if linked into many instrument drivers, can deplete available application resources.

If you use Microsoft Visual C/C++ to develop an instrument driver, set the code generation “Use RunTime library” option to “Multithreaded DLL”. This compiles with the “/MD” option. You will then also need to redistribute the Microsoft C-runtime DLL’s with your instrument driver. For example, MSVCRT.DLL and MSVCP60.DLL. If you develop the Instrument Driver in LabWindows/CVI, this is handled correctly automatically. Issues with other development environments are unknown.

Section 5
Instrument Driver Programmatic Developer Interface for the G Frameworks

5.1 Introduction

This section defines the framework-specific requirements for the G Frameworks. These specifications may be clarifications, enhancements, or restrictions on the corresponding requirements from the equivalent parts of Section 3.

5.2 Compatible Application Development Environments

The list of target Application Development Environments and supported versions for the G Frameworks can be found in the G Framework section of the VPP-2 document.

5.3 Naming Conventions

The general naming conventions defined in Section 3.4 are all valid and sufficient for the G Frameworks.

Appendix A
The Vpptype.h Header File

```
/*---------------------------------------------------------    */
/* Distributed by VXIplug&play Systems Alliance              */
/*                                                            */
/* Do not modify the contents of this file.                  */
/*---------------------------------------------------------    */
/*                                                            */
/* Title   : VPPTYPE.H                                        */
/* Date    : 02-02-95                                         */
/* Purpose : VXIplug&play instrument driver header file      */
/*                                                            */
/*---------------------------------------------------------    */

#ifndef __VPPTYPE_HEADER
#define __VPPTYPE_HEADER

#include "visatype.h"

/*- Completion and Error Codes ----------------------  */

#define VI_WARN_NSUP_ID_QUERY      (0x3FFC0101L)
#define VI_WARN_NSUP_RESET         (0x3FFC0102L)
#define VI_WARN_NSUP_SELF_TEST     (0x3FFC0103L)
#define VI_WARN_NSUP_ERROR_QUERY   (0x3FFC0104L)
#define VI_WARN_NSUP_REV_QUERY     (0x3FFC0105L)

#define VI_ERROR_PARAMETER1        (_VI_ERROR+0x3FFC0001L)
#define VI_ERROR_PARAMETER2        (_VI_ERROR+0x3FFC0002L)
#define VI_ERROR_PARAMETER3        (_VI_ERROR+0x3FFC0003L)
#define VI_ERROR_PARAMETER4        (_VI_ERROR+0x3FFC0004L)
#define VI_ERROR_PARAMETER5        (_VI_ERROR+0x3FFC0005L)
#define VI_ERROR_PARAMETER6        (_VI_ERROR+0x3FFC0006L)
#define VI_ERROR_PARAMETER7        (_VI_ERROR+0x3FFC0007L)
#define VI_ERROR_PARAMETER8        (_VI_ERROR+0x3FFC0008L)
#define VI_ERROR_FAIL_ID_QUERY     (_VI_ERROR+0x3FFC0011L)
#define VI_ERROR_INV_RESPONSE      (_VI_ERROR+0x3FFC0012L)

/*- Additional Definitions --------------------------*/

#define VI_ON       (1)
#define VI_OFF      (0)

#endif
```

Appendix B
The Vpptype.bas Header File

```
' -------------------------------------------------------------------
'  Distributed by VXIplug&play Systems Alliance
'  Do not modify the contents of this file.
' -------------------------------------------------------------------
'  Title   : VPPTYPE.BAS
'  Date    : 02-14-95
'  Purpose : VXIplug&play instrument driver header file
' -------------------------------------------------------------------

Global Const VI_NULL                             = 0
Global Const VI_TRUE                             = 1
Global Const VI_FALSE                            = 0

' - Completion and Error Codes -------------------------------------

Global Const VI_WARN_NSUP_ID_QUERY               = &H3FFC0101&
Global Const VI_WARN_NSUP_RESET                  = &H3FFC0102&
Global Const VI_WARN_NSUP_SELF_TEST              = &H3FFC0103&
Global Const VI_WARN_NSUP_ERROR_QUERY            = &H3FFC0104&
Global Const VI_WARN_NSUP_REV_QUERY              = &H3FFC0105&

Global Const VI_ERROR_PARAMETER1                 = &HBFFC0001&
Global Const VI_ERROR_PARAMETER2                 = &HBFFC0002&
Global Const VI_ERROR_PARAMETER3                 = &HBFFC0003&
Global Const VI_ERROR_PARAMETER4                 = &HBFFC0004&
Global Const VI_ERROR_PARAMETER5                 = &HBFFC0005&
Global Const VI_ERROR_PARAMETER6                 = &HBFFC0006&
Global Const VI_ERROR_PARAMETER7                 = &HBFFC0007&
Global Const VI_ERROR_PARAMETER8                 = &HBFFC0008&
Global Const VI_ERROR_FAIL_ID_QUERY              = &HBFFC0011&
Global Const VI_ERROR_INV_RESPONSE               = &HBFFC0012&

' - Additional Definitions -----------------------------------------

Global Const VI_ON                               = 1
Global Const VI_OFF                              = 0
```

Appendix C
Example Instrument Driver Header File

```
/* VXIplug&play instrument driver for the vxi1234a   */

#ifndef __vxi1234a_HEADER
#define __vxi1234a_HEADER

#include <vpptype.h>

#if defined(__cplusplus) || defined(__cplusplus__)
extern "C" {
#endif


/*- Required functions ----------------------------------*/

ViStatus _VI_FUNC vxi1234a_init(ViRsrc rsrcName,
     ViBoolean id_query, ViBoolean reset, ViPSession vi);

ViStatus _VI_FUNC vxi1234a_close(ViSession vi);

ViStatus _VI_FUNC vxi1234a_reset(ViSession vi);

ViStatus _VI_FUNC vxi1234a_self_test(ViSession vi,
     ViPInt16 test_result, ViChar test_message[]);

ViStatus _VI_FUNC vxi1234a_revision_query(ViSession vi,
     ViChar driver_rev[],
     ViChar instr_rev[]);

ViStatus _VI_FUNC vxi1234a_error_query(ViSession vi,
     ViPInt32 error, ViChar error_message[]);

ViStatus _VI_FUNC vxi1234a_error_message(ViSession vi,
     ViStatus error, ViChar message[]);
/*- Application functions ------------------------------*/

/* setCoupling: set the coupling mode of the channel */
#define vxi1234a_COUPLING_AC  0    /* AC coupling */
#define vxi1234a_COUPLING_DC  1    /* DC coupling */
ViStatus _VI_FUNC vxi1234a_setCoupling(ViSession vi,
     ViInt16 coupling);

/* setRange: set the measurement range of the channel*/
#define vxi1234a_RANGE_MIN    1
#define vxi1234a_RANGE_MAX    1000
ViStatus _VI_FUNC vxi1234a_setRange(ViSession vi,
     ViInt16 range);

/* readValue: reads a value from the instrument      */
ViStatus _VI_FUNC vxi1234a_readValue(ViSession vi,
     ViPInt32 value);

/* waitForSrq: blocks for SRQ in lieu of callbacks   */
ViStatus _VI_FUNC vxi1234a_waitForSrq(ViSession vi);

/* onSrq: installs a handler for SRQ callbacks       */
#ifdef INSTR_CALLBACKS
typedef void (* _VI_FUNCH vxi1234a_SrqHandler) (ViInt32
data);
ViStatus _VI_FUNC vxi1234a_onSrq(ViSession vi,
     vxi1234a_SrqHandler srqCallback);
#endif


#if defined(__cplusplus) || defined(__cplusplus__)
}
#endif

#endif
```

Appendix D
Example Visual Basic Header File

Note: ***For the purpose of readability these declaration statements are shown on multiple lines; however, in a Visual Basic program each complete declaration must be on a single line.***

```
' VXIplug&play instrument driver for the vxi1234a

'- Input coupling constants -----------------------------
Global Const vxi1234a_COUPLING_AC  &H0&
Global Const vxi1234a_COUPLING_DC  &H1&

'- Input range constants --------------------------------
Global Const vxi1234a_RANGE_MIN    &H1&
Global Const vxi1234a_RANGE_MAX    1000&


'- Required functions -----------------------------------

Declare Function vxi1234a_init Lib "vxi1234a.dll"
     (ByVal rsrcName As String, ByVal id_query As Integer,
      ByVal reset As Integer, vi As Long) As Long

Declare Function vxi1234a_close Lib "vxi1234a.dll"
     (ByVal vi As Long) As Long

Declare Function vxi1234a_reset Lib "vxi1234a.dll"
     (ByVal vi As Long) As Long

Declare Function vxi1234a_self_test Lib "vxi1234a.dll"
     (ByVal vi As Long, test_result As Long,
      ByVal test_message As String) As Long

Declare Function vxi1234a_revision_query Lib "vxi1234a.dll"
     (ByVal vi As Long, ByVal driver_rev As String,
      ByVal instr_rev As String) As Long

Declare Function vxi1234a_error_query Lib "vxi1234a.dll"
     (ByVal vi As Long, error As Long,
      ByVal error_message As String) As Long

Declare Function vxi1234a_error_message Lib "vxi1234a.dll"
     (ByVal vi As Long, ByVal error As Long,
      ByVal message As String) As Long
```

```
'- Application functions ---------------------------------

' setCoupling: set coupling mode of the channel
Declare Function vxi1234a_setCoupling Lib "vxi1234a.dll"
     (ByVal vi As Long, ByVal coupling As Integer) As Long

' setRange: set the measurement range of the channel
Declare Function vxi1234a_setRange Lib "vxi1234a.dll"
     (ByVal vi As Long, ByVal range As Integer) As Long

' readValue: reads a value from the instrument
Declare Function vxi1234a_readValue Lib "vxi1234a.dll"
     (ByVal vi As Long, value As Long) As Long

' waitForSrq: blocks for SRQ in lieu of callbacks
Declare Function vxi1234a_waitForSrq Lib "vxi1234a.dll"
     (ByVal vi As Long) As Long
```