

Systems Alliance

VPP-3.1: Instrument Drivers Architecture and Design Specification

Revision 4.2

April 14, 2008

VPP-3.1 Revision History

This section is an overview of the revision history of the VPP-3.1 specification.

Revision 1.0, July 15, 1994

This edition reflects a non-technical revision for style and format issues.

Revision 1.1, August 17, 1994

This edition reflects edits to technical omissions and inconsistencies between VPP documents.

Revision 2.0, November 28, 1995

This edition reflects technical changes made at the November 3, 1994 Technical Working Group and subsequent edits made during review.

Revision 3.0, February 21, 1995

This edition reflects changes to the prototypes of required functions to make them language and framework non-specific, use the compatible data types from VPP-3.4, and add completion codes and changes to better specify the error/status reporting mechanism.

Revision 4.0, February 5, 1996

This edition is the result of a reorganization of the entire VPP 3.X series of specifications. The goal of the reorganization was to improve the documentation of instrument driver requirements for all frameworks.

Revision 4.1, December 4, 1998

This edition updated the information regarding contacting the alliance. References to the VPP-5 Component Knowledge Base specification, which was obsoleted by the alliance, were removed.

Revision 4.2, February 14, 2008

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

Revision 4.2, 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.

NOTICE

VPP-3.1: *Instrument Drivers Architecture and Design Specification* is authored by the IVI Foundation member companies. For a vendor membership roster list, please visit the IVI Foundation web site at www.ivifoundation.org.

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

Warranty

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.

Contents

Section 1

Introduction to the <i>VXIplug&play</i> Systems Alliance and the IVI Foundation	1-1
---	-----

Section 2

Overview of Instrument Drivers Architecture and Design Specification.....	2-1
2.1 Introduction.....	2-1
2.2 Objectives of this Specification	2-1
2.3 Audience for this Specification.....	2-2
2.4 Scope and Organization of this Specification.....	2-2
2.5 Application of this Specification	2-3
2.6 References.....	2-4
2.7 Definitions of Terms and Acronyms	2-5
2.8 Conventions	2-6

Section 3

Introduction to Instrument Drivers	3-1
3.1 Introduction.....	3-1
3.2 Historical Evolution of Instrument Drivers	3-1
3.2.1 Basic Rack and Stack Systems	3-1
3.2.2 The Personal Computer Influence	3-2
3.2.3 Standard Instrument Command Sets.....	3-3
3.2.4 Modular Instrument Driver Software	3-3
3.2.5 The Impact of VXI on Instrument Drivers	3-4
3.2.6 <i>VXIplug&play</i> Instrument Drivers.....	3-5
3.2.7 IVI Instrument Drivers.....	3-6
3.3 Use of Instrument Drivers.....	3-7
3.4 Purpose and Benefits of Instrument Drivers.....	3-7

Section 4

Instrument Driver Requirements	4-1
4.1 Introduction.....	4-1
4.2 Full-Feature Control	4-1
4.3 Source Code Distribution.....	4-2
4.4 Modularity and Hierarchy.....	4-2
4.5 Consistency in Design and Implementation	4-2
4.6 Error Handling	4-3
4.7 Help Information.....	4-4
4.8 Documentation.....	4-4
4.9 Revision Control	4-4

Section 5

Instrument Driver Architecture..... 5-1

- 5.1 Introduction..... 5-1
- 5.2 Instrument Driver External Interface Model 5-1
 - 5.2.1 Functional Body..... 5-2
 - 5.2.2 VISA I/O Interface..... 5-3
 - 5.2.3 Subroutine Interface..... 5-3
 - 5.2.4 Programmatic Developer Interface..... 5-4
 - 5.2.5 Interactive Developer Interface 5-4
- 5.3 Instrument Driver Internal Design Model 5-5
 - 5.3.1 Application Functions..... 5-7
 - 5.3.2 Component Functions 5-7
 - 5.3.3 Initialize Function 5-8
 - 5.3.4 Capability Classes..... 5-9
 - 5.3.5 Utility Functions 5-13
 - 5.3.6 Close Function 5-14

Section 1

Introduction to the VXIplug&play Systems Alliance and the IVI Foundation

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

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

Section 2

Overview of Instrument Drivers Architecture and Design Specification

2.1 Introduction

This section introduces the *Instrument Driver Architecture and Design Specification*. The Instrument Driver Technical Working Group both wrote the specification and performed the technical work that the specification discusses.

This section summarizes the *Instrument Drivers Architecture and Design Specification* and contains 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 Drivers Architecture and Design Specification* provides a general overview of the VXIplug&play System Alliance standard for the design, development, and distribution of multi-vendor instrument drivers. This specification describes a single model for the architecture of all instrument drivers. Subsequent specifications document the guidelines for the development of specific instrument drivers so that they can be used in a variety of system frameworks.

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—either instrument vendors, system integrators, or 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 *VXIplug&play* Systems Alliance standard for instrument drivers.

Section 1 explains the *VXIplug&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 summarizes instrument drivers and includes a brief history of the evolution of instrument programming and the need for standard instrument drivers. It defines the term *instrument driver*, and lists the benefits offered by instrument drivers. This section explains the scope, purpose, and use of instrument drivers.

Section 4 identifies the requirements that standard instrument drivers must address. Many of the requirements are for the benefit of the end user and relate to how instrument drivers are actually used in the context of an overall system. This section shows how these requirements led to the standard architecture for all *VXIplug&play* instrument drivers described in this specification.

Section 5 describes the standard architecture for *VXIplug&play* instrument drivers. The standard architecture is described using two models. The first is a general model for the external interface of an instrument driver. This model identifies how an instrument driver interacts with other software components in the system and with the end user of the driver. The second model is a more detailed internal model that defines the organization of the various component pieces contained in a standard instrument driver. This standard architecture for instrument drivers provides the framework whereby all instrument driver developers can have a consistent approach for the design, packaging, and use of standard, multi-vendor instrument drivers.

2.5 Application of this Specification

This specification is intended to be used by developers of *VXIplug&play* instrument drivers. It is also useful as a reference for end users of *VXIplug&play* instrument drivers. This specification is intended to be used in conjunction with the *Instrument Driver Functional Body Specification* (VPP-3.2), the *Instrument Driver Interactive Developer Interface Specification* (VPP -3.3), the *Instrument Driver Programmatic Developer Specification* (VPP -3.4), and the *VISA Library Specifications* (VPP-4.X). These related specifications describe the implementation details for specific instrument drivers that are used with specific system frameworks. *VXIplug&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 *VXIplug&play System Frameworks Specification* (VPP-2).

2.6 References

Several other VXIplug&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-3.2 *Instrument Driver Functional Body Specification*
- VPP-3.3 *Instrument Driver Interactive Developer Interface Specification*
- VPP-3.4 *Instrument Driver Programmatic Developer 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
- LabWindows/CVI C based ADE
- LabVIEW Graphical programming ADE
- Agilent VEE Graphical programming ADE
- Instrument Driver Library of functions for controlling a specific instrument
- Template or Required Function
Instrument Driver function required to be implemented in all *VXIplug&play* instrument drivers.
- Application Function A high-level, test-oriented, instrument driver function. It is typically developed from the instrument driver subsystem functions.
- VISA Virtual Instrument Software Architecture
- VI LabVIEW program or Virtual Instrument
- LLB LabVIEW VI library
- DLL Dynamic Link Library
- IVI Interchangeable Virtual Instruments
- IVI driver A software module that controls a hardware device and that complies with the IVI Foundation specifications. For IVI Foundation compliance requirements, refer to *IVI-3.1: Driver Architecture Specification*.

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 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

Introduction to Instrument Drivers

3.1 Introduction

This section gives an introduction to instrument driver software. Instrument driver software has become increasingly popular over the last several years, and both users and vendors have taken advantage of the technology. Increased use of instrument drivers has fueled a continuous improvement process that has resulted in high-quality instrument drivers. The Alliance has a guiding principle to treat instrument drivers as part of the instrument. VXI instruments do nothing without software, and the Alliance is committed to delivering standard software with every *VXIplug&play* instrument.

The *VXIplug&play* instrument driver architecture leverages off of existing technology by building on the most popular and successful instrument driver solutions on the market. In addition, the *VXIplug&play* instrument driver architecture adds a variety of useful features to improve even further on pre-existing technology for instrument drivers.

3.2 Historical Evolution of Instrument Drivers

To understand a standard architecture and methodology for the development of instrument drivers, it is useful to understand some of the history of the evolution of modern instrument driver software. A first step in this process is to answer the question “What is an instrument driver?” It is not a new concept. Instrumentation users have been writing their own instrument drivers for years, as many are aware. An instrument driver, in the simplest definition, is a piece of software that handles the details of controlling and communicating with a specific instrument.

The task of programming instruments in a test system has always been a major concern for end users and a major cost for the overall system development. Many users know that programming can often be the most time-consuming part of developing a system. The developer spends much valuable time learning the specific programming requirements of each instrument in the system, including the undocumented or unexpected features.

3.2.1 Basic Rack and Stack Systems

“Rack and stack” IEEE 488, or GPIB, instruments with their ASCII message-string programming methodology, have long been the most widely used type of instruments for multi-vendor systems. Almost all GPIB instruments are designed for interactive use through a physical front panel and also offer remote control capability via a GPIB port on the back of the instrument. The details for how to program the instrument remotely are usually documented in

the instrument manual in the form of ASCII command sets that cause the instrument to perform the desired operation. Documenting an instrument command set in the user manual, along with some example program listings, has traditionally been the standard method for an instrument vendor to assist the end user in programming the instrument. These documentation methods have served the industry well for many years, but this approach still places the responsibility for writing the program code on the user, many of whom may end up writing very similar application programs.

In the early days of computer-controlled instrumentation systems, the interpreted BASIC language was a widely-used programming environment. To control the instrument from the computer, the users would typically use I/O statements throughout their application program to send and receive the appropriate command and data strings to and from the various instruments in the system. The BASIC language, with its built-in formatted I/O capabilities, matched well with the formatted ASCII command and data strings used to control message-based GPIB instruments. Over the years, special-purpose versions of BASIC or similar languages with ever more powerful tools oriented specifically toward message-based instrument programming appeared on the market and enjoyed various levels of success. During this era, the concept of an instrument driver as referred to in this specification did not exist.

3.2.2 The Personal Computer Influence

Throughout the 1980s, computer-controlled instrumentation became more widespread. Several key factors helped this occur and influenced the evolution of instrument drivers.

The personal computer revolution enabled more users to take advantage of automated computer-controlled systems. Not only did many more scientists and engineers become comfortable using computers, but also the lower cost of PC-based systems made such technology applicable to many more applications. Also during this time, computer programming technology made tremendous strides. Compiled languages such as Pascal and C offered many benefits over interpreted BASIC, including faster execution speed and more powerful program and data structures. In addition, these new languages offered the ability to make programs more modular by organizing them into separate pieces that could be developed, maintained, and managed as independent, but related, software objects.

As computer-controlled instrumentation became more widespread and successful, many users began to realize that they often used the same instruments in a variety of systems. In addition, maintaining or enhancing existing systems often entailed replacing older instruments with newer or less expensive models. These realizations led to two identifiable trends in instrumentation applications: standard instrument command sets and modular instrument driver software.

3.2.3 Standard Instrument Command Sets

If the same commands work for multiple instruments, regardless of the manufacturer, users can interchange or upgrade instruments and reduced the amount of changes to their application programs. In particular, many of the installed base of users who had substantial investments in BASIC or other software environments that did not easily lend themselves to software modularity lobbied for this approach. Through the mid to late 1980s, many standard organizations, including the IEEE, worked on this objective with little progress. The IEEE 488.2 specification, completed in 1987, more carefully defined the operation of instruments but did not address the issue of standard command sets.

Finally, in 1990, the SCPI (standard commands for programmable instruments) Consortium was formed. This organization, many of whose members were also involved in the VXIbus Consortium and are now involved in the *VXIplug&play* Systems Alliance, approved a specification for standardized commands for message-based programmable instruments. The SCPI Consortium is still active and the standard document is updated on a yearly basis as new commands are added. While more and more companies continue to endorse the SCPI standard and use it in their new instrument designs, the majority of instruments available today do not use this standard command set. In addition, while many users of SCPI instruments applaud the progress and have experienced improvement in productivity once they learn the standard command set, instrument interchangeability is still not a reality because most instruments have different, often unique functionality.

While the SCPI standard is certainly recognized by the entire industry as a step forward, the lack of progress on this issue encouraged both users and vendors to explore other approaches before SCPI was completed. They needed to decrease the time required to program instruments, facilitating instrument interchangeability and easing system maintenance. Rather than trying to solve the problems by standardizing the instruments from all suppliers, both users and vendors began to take advantage of new computer science technology to address the issues by making software more modular and flexible.

3.2.4 Modular Instrument Driver Software

In the early 1980s, as users gained experience with computer-controlled instrumentation and programming tools improved, they began to make their code more modular and reuse the portions that control specific instruments. Over the years, many users developed their own guidelines for writing, documenting, and managing the test software their companies developed. In some cases, these internally-developed software standards evolved over a number of years and represent tens or even hundreds of man-years of development.

At the same time, instrumentation suppliers began to deliver powerful, PC-based software tools that combined the latest advances in general-purpose PC software with instrumentation-specific tools. Libraries of pre-written software routines for particular instruments, known as instrument driver libraries, appeared on the market and began to be used for the first time. These initial instrument drivers were often written by the software supplier, rather than the instrument vendor,

and the list of available drivers was small but growing. Numerous software suppliers entered the market and promoted their particular application software tools along with the associated libraries of instrument drivers.

Through the mid 1980s until today, more and more users have taken advantage of instrument driver concepts, whether internally designed or through a commercial software product. While the benefits were real, one drawback was that each software package had a unique approach to instrument drivers and unique instrument drivers. As years passed and the marketplace evaluated these variety of approaches, the natural process of competition helped the market refine the good ideas and discard the others. It also allowed the developers of successful products to continue the investment needed to continuously improve and refine the product itself and its associated instrument drivers, take advantage of the latest technology, and maintain compatibility with the rapidly evolving PC market.

By the early 1990s, both personal computers and instrument driver technology had become a mainstream technology for instrumentation users. Though software products from different vendors still had unique instrument drivers, the more mature packages generally all featured a very long list of available instrument drivers. Some instrument vendors had recognized the value an instrument driver adds to their instrument and were actively involved in writing and promoting instrument drivers. Other instrument vendors, on the other hand, continued to simply document their command sets in the instrument manual and leave the instrument driver up to someone else, often the software supplier.

The most successful instrument driver concepts have always distributed instrument drivers in source code and provided end users with access to the same tools developers use to write drivers. With this philosophy, new instrument drivers were often easily developed by end users by modifying an existing driver for another instrument. End users, in general, had come to view the availability of an instrument driver as an important factor in the choice of a particular instrument. However, users still had access to standard instrument driver development tools and source code for other instruments.

3.2.5 The Impact of VXI on Instrument Drivers

In 1987 the VXIbus Consortium produced the first VXIbus specification. Over the next several years, VXI enjoyed tremendous success and growth. As VXI grew, so did users' expectations for instrument drivers. There are several reasons for this. First, it is widely known that VXI instruments can only be used with software. Whereas the evolutionary history of GPIB instrument control resulted in users never really expecting driver code to be included with a GPIB instrument, VXI set a new expectation. In the past, the burden of developing instrument-specific driver code had been placed on each and every user of each and every instrument. Because VXI instruments can only be used with software, VXI users and potential users began to make it known that they expected a better answer.

Another very important aspect is that VXI actually has two paradigms for instrument control. The first paradigm is message-based control, which is very similar to the message-based control

methodology used for years with GPIB instruments. With this paradigm, the instrument is controlled using fairly high level ASCII command and data strings which are communicated to/from all instruments in a standard way and interpreted by each instrument in a device-specific way.

The other paradigm available in VXI is register-based control, which is a new paradigm for instrument control. With register-based control, the instrument is controlled through very low-level peeks and pokes of individual binary registers. Because there is no overhead for the instrument to interpret a message string, and because the software can directly access binary registers and data on the instrument without any formatting or interpretation, register-based control can, in some cases, offer higher performance than a message-based approach. However, because each register-based device has its own unique set of registers that operate in a unique way for each instrument, the software to control a register-based module is device-specific. Register-based control software, which is very low-level and unique to each instrument, is similar in concept to the “device driver” software modules required for individual computer plug-in boards such as network interfaces, display adapters, disk controllers, and so on.

Because VXI has two paradigms available for instrument control, it promises the benefit of higher performance than GPIB technology. A particular instrument can use one or the other of these paradigms, or a mixture of both on the same module. Some of the VXI performance potential lies in register-based programming, but higher performance is also available through the use of other unique new VXI concepts such as fast data channels, shared memory, and distributed, multi-processor architectures that work with both message-based and register-based devices. Taking full advantage of many of these “high performance” programming techniques is a far more complex task than the simple message-string approach with which GPIB users are familiar and instrument driver concepts had evolved. Therefore, end-user programming of these unique features without a pre-written instrument driver may be more of a challenge and take more time and expense than with previous-generation GPIB technology. For these reasons, instrument drivers are recognized as a key component in the successful implementation of VXI technology by end users.

3.2.6 VXI*plug&play* Instrument Drivers

VXI has the potential to be much easier to use, even for the novice, than any other instrumentation technology available. Most everything in VXI can be controlled by software, allowing flexibility in presentation to end users. One of the keys to delivering on this flexibility, however, in the minds of the VXI*plug&play* Systems Alliance and its members, is to deliver high-quality, easy to use instrument drivers with every VXI instrument. These instrument drivers must be standard so that drivers supplied by multiple vendors can be used in the same system. In addition, they must be consistent in their architecture, design, and use, and must be open, fully documented, and distributed and used in a consistent fashion.

To achieve these features, the VXI*plug&play* Systems Alliance was formed, and its Instrument Driver Technical Working Group developed this document and its associated subsequent documents. The concepts embodied in this standard represent the culmination of over ten years

of evolution of instrument driver technology. This standard combines the best of existing, proven technology with new guidelines and concepts to maximize compatibility and consistency in a multi-vendor system. *VXIplug&play* instrument drivers provide libraries for programmatic control that can be used in a wide variety of higher-level software development environments. The standard, multi-vendor instrument driver software defined in this document maximizes ease of use for end users, and ensures that VXI remains an open, multi-vendor technology at both the module and system level.

3.2.7 IVI Instrument Drivers

The IVI Foundation is a group of end-user companies, system integrators, and instrument vendors working together to define standard approaches to a variety of test oriented tasks that use instrument drivers. The IVI Foundation members believe that many difficult programming tasks faced by test system developers can be facilitated by taking advantage of IVI features such as standard instrument APIs, instrument interchangeability, execution performance, and simulation.

The C-based IVI instrument driver architecture leverages the architecture standards defined by the *VXIplug&play* Alliance. Some of the IVI requirements remain the same as those defined by the *VXIplug&play* Alliance, such as function panel format and sub file format. Furthermore, several of the required IVI instrument driver functions are similar to the functions required by *VXIplug&play* specifications. For example, IVI instrument drivers must have Initialize, Reset, Self-Test, and Close functions.

For some shared requirements, IVI instrument driver specifications reference appropriate sections of the *VXIplug&play* specifications. Other requirements, such as error handling and naming formats that were built on the existing *VXIplug&play* specifications, are redefined in the IVI Foundation specifications.

Instrument driver developers may choose to develop either *VXIplug&play* instrument drivers or IVI instrument drivers, as suits their customer needs.

3.3 Use of Instrument Drivers

Instrument drivers have always been an important component of instrumentation system software. They can dramatically increase productivity by reducing test development times and making test software modular, so that it is easier to maintain and reuse.

VXIplug&play instrument drivers are conceptually one layer above the traditional command set instrument programming methodology of the past. Rather than not delivering software with an instrument and requiring a user to include individual I/O statements throughout their application program, a *VXIplug&play* instrument driver includes all the communication details of a particular instrument in high-level software functions that are directly usable by end users as part of their application programs. The *VXIplug&play* instrument driver architecture defined in this document accommodates traditional message-based instruments, both SCPI and non-SCPI, as well as direct control of register-based modules.

VXIplug&play instrument drivers have a consistent architecture and are developed and used in a consistent fashion. *VXIplug&play* instrument drivers are delivered in source code and ready to use libraries, and are fully documented. In addition, the software tools that were used to develop the instrument driver are openly available to any vendor or user. In this way, users can understand the operation of the driver itself, and can modify or enhance the operation of a particular instrument driver to achieve the optimum level of performance and flexibility for their applications.

3.4 Purpose and Benefits of Instrument Drivers

The benefits of standard *VXIplug&play* instrument drivers are tremendous. First, vendors of instruments can be responsible for developing drivers for their own instruments and be guaranteed of interoperability with those from other vendors. This approach is desirable because the developer of a particular instrument knows how to use the unique capabilities of that instrument, and therefore is the best candidate for developing the instrument driver. The quality of the driver will be high, and users can hold the instrument vendor accountable for that quality and give direct feedback. A standard ensures that drivers from multiple vendors are designed, packaged, and used in a consistent way, which increases ease of use for end users. System-level openness and multi-vendor interoperability, therefore, are greatly enhanced.

The *VXIplug&play* instrument driver standard minimizes duplication of effort in the industry. *VXIplug&play* instrument drivers can work with a wide variety of higher-level software tools. This capability unifies the market and promotes both de facto and formal standards in many other areas. When evaluating competing higher-level software packages, users can focus on understanding the differences in the methodology of higher-level software tools themselves, rather than worrying about subtle undocumented differences in the different instrument drivers used in the different packages.

Section 4

Instrument Driver Requirements

4.1 Introduction

VXI*plug&play* instrument drivers provide comprehensive access to the test and measurement capabilities of the instrument. To develop a standard for multi-vendor instrument drivers, it is important to identify the fundamental requirements. The first fundamental requirement is that instrument drivers should provide turn-key, ready-to-go software modules that the user can use directly in his or her own application program. Other areas of interest include the scope of functionality of the instrument driver, modularity and hierarchy, consistency in design and implementation, source code distribution, error handling, help information, documentation, revision control, distribution media, and installation.

4.2 Full-Feature Control

In the context of an automated test system, in which performance is always a factor, the perfect instrument driver is one that does exactly what the user needs for a particular application—and nothing else. The overhead associated with a full-function driver may not be acceptable for a performance-critical application that requires only a subset of the instrument's capability. For these reasons, the working group used three key architectural guidelines in defining this standard for instrument driver design.

First, it is imperative that instrument drivers be provided in source code so users can customize and optimize their operation. Second, the architecture must be modular, providing multiple levels of access to functionality, so that users can use functional subsets of individual drivers. Finally, the structure between VXI*plug&play* instrument drivers from different vendors is similar; therefore, the knowledge gained in learning to use one instrument driver is highly leveraged across all VXI*plug&play* instrument drivers.

One of the benefits of software is flexibility in presenting an instrument's functionality to a user. With software, a single hardware instrument can have multiple panels and function calls targeted for particular applications. For these reasons, this specification does not define the specific requirements for each type of generic instrument, such as a multimeter, digitizer, spectrum analyzer, and so on. Rather, this specification defines consistent standards for architecture and packaging so that users get the unique benefits of individual instruments, yet in a consistent manner that promotes ease of use and is interoperable between vendors.

RECOMMENDATION 4.1

VXI*plug&play* instrument drivers should provide comprehensive control of features of the instrument.

Some vendors of ADEs publish additional instrument driver guidelines to ensure that instrument drivers integrate well within the environment. For example, to enable users to easily access LabVIEW instrument driver functions, G language instrument drivers must also provide palette menu files. ADE-specific instrument driver guidelines build upon the *VXIplug&play* specification by describing enhanced capabilities within the environment and techniques for ensuring consistency with other software libraries. Instrument driver developers may choose to follow the ADE-specific guidelines as long as they are consistent with the *VXIplug&play* specifications.

4.3 Source Code Distribution

VXIplug&play instrument drivers are provided in source code as well as a pre-compiled library. In this way, users can view *VXIplug&play* instrument drivers not as a special piece of closed code that performs low-level I/O details, but rather as a general-purpose software module that can be integrated directly with the rest of their system. Source code enables users to understand, modify, and enhance drivers to meet their needs.

RULE 4.1

The functional body of a *VXIplug&play* instrument driver **SHALL** be provided in source code.

4.4 Modularity and Hierarchy

As mentioned previously, one of the benefits of software is flexibility in presenting an instrument's functionality to a user. Users have their own unique requirements and want to take advantage of software flexibility in their own application. Delivering an instrument driver as a single closed module, therefore, is not an appropriate solution. Instrument drivers must be delivered in a form that still gives users flexibility in how they use the driver.

VXIplug&play instrument drivers have a well defined modular design that is hierarchical in nature. If a user wants a simple, single-function interface to an instrument, such an interface is provided by application functions. If, however, the user wants more flexibility and access to more of the instrument's functionality, the lower-level subsystem components of the driver are also available as individual modular pieces that can be used individually.

4.5 Consistency in Design and Implementation

The guidelines for design and implementation of standard *VXIplug&play* instrument drivers are carefully defined in this and related specifications. These guidelines are written for the instrument driver developer to ensure consistency between drivers developed by different parties. The objective of these guidelines is for all developers of instrument drivers to deliver

drivers that have the same modular architecture, while at the same time delivering the unique functionality and features of a particular instrument. In addition, end users will see the same consistent packaging and use of drivers from all vendors.

These guidelines benefit both developers and users. A consistent standard for implementation enables developers to streamline the training and support processes associated with driver development, and to maximize their efficiency by leveraging from one driver to the next as they develop expertise in each step of development. End users will benefit from these guidelines because the knowledge required to use one instrument driver is highly leveraged across other instrument drivers.

4.6 Error Handling

Error handling is a very important aspect of instrument drivers. Consistent guidelines for error handling ensure that instrument drivers developed by different parties use a consistent approach to error handling with consistent error messages. These guidelines save time for developers by defining the mechanisms, and improve ease of use for end users by providing consistency between drivers from different vendors.

RULE 4.2

All *VXIplug&play* instrument driver functions **SHALL** return error status information as the return value of the function. The return value **SHALL** be of type `ViStatus`.

RULE 4.3

When a *VXIplug&play* instrument driver function detects an error it **SHALL** return an error value indicating the first error encountered in that function. If a *VXIplug&play* instrument driver function executes successfully, without detecting errors or warnings, it **SHALL** return the value `VI_SUCCESS`.

RULE 4.4

When a *VXIplug&play* instrument driver function reports status information, errors **SHALL** take precedence over warnings.

RULE 4.5

When a *VXIplug&play* instrument driver function encounters a warning condition, it **SHALL** continue executing.

OBSERVATION 4.1

Since a *VXIplug&play* instrument driver function continues executing after encountering a warning condition, it is possible that multiple warning conditions may be encountered. Therefore, the return value of the function is indeterminate and is left to the discretion of the instrument driver developer.

4.7 Help Information

Help information is a very important part of a *VXIplug&play* instrument driver because it is the mechanism for making the end user successful in using and understanding the instrument driver. Various levels of required and recommended help information are defined in VPP-3.3: *Instrument Driver Interactive Developer Interface Specification*.

4.8 Documentation

Documentation is also an important part of an instrument driver. Guidelines are provided not only for documenting how the end user can use the driver, but also for how to document the internal source code of the driver. Documenting the source code of the driver is especially useful for users who want to enhance or modify the operation of the driver.

RULE 4.6

For each function provided within a *VXIplug&play* instrument driver that conforms to a particular framework, documentation which describes the function, its usage, return values, and parameters **SHALL** be provided as defined by that framework.

RULE 4.7

The instrument driver documentation **SHALL** contain the *VXIplug&play* framework revision with which the instrument driver is compatible, the revision of the instrument driver, and the minimum instrument revision for which the driver is compatible.

4.9 Revision Control

VXIplug&play instrument drivers have a standard mechanism for revision control. The revision level can be obtained by calling the revision function of the driver so that the user's software application can determine the revision level. The revision control guidelines cover not only the revision level of the instrument driver itself, but also the revision levels of the instrument hardware that the instrument driver supports.

RULE 4.8

The instrument driver source code **SHALL** contain as a comment the *VXIplug&play* framework revision with which the instrument driver is compatible, the revision of the instrument driver, and the minimum instrument revision for which the driver is compatible.

Section 5

Instrument Driver Architecture

5.1 Introduction

To define a standard for instrument driver software design and development, it is necessary to use conceptual models around which the design specifications are written. This specification uses two architectural models for discussion.

The first model, called the instrument driver external interface model, shows how the instrument driver interfaces to the other software in the system. This model is described briefly in this section to give some insight into key architectural decisions with regard to *VXIplug&play* instrument drivers, and to give some context as to how instrument drivers are used. Subsequent *VXIplug&play* specifications use this model to explain how the *VXIplug&play* instrument driver development tools handle the various external interfaces when developing instrument drivers.

The second model, called the instrument driver internal design model, defines how an instrument driver software module is organized internally. This model is briefly covered in this section to show the consistency of approach to instrument driver design regardless of the instrument driver development tools used. In subsequent *VXIplug&play* specifications, this model is used to show the specific design and implementation issues associated with implementing instrument drivers with specific *VXIplug&play* development tools.

5.2 Instrument Driver External Interface Model

A *VXIplug&play* instrument driver consists of software modules that control a specific instrument. The software modules that make up an instrument driver must interact with other software in the overall system, both to communicate with the instrument and to communicate with higher-level software and/or end users who use the instrument driver. The first step in creating a standard for instrument drivers, therefore, is to define a model to explain how the instrument driver interacts with the rest of the system. Using this model, it is possible to discuss how specific instrument driver development tools address these external interfaces to understand how instrument drivers are packaged. In addition, defining these external interfaces enables independent development of other software that uses instrument drivers.

Figure 5-1 shows a general model for how an instrument interfaces with the rest of the system.

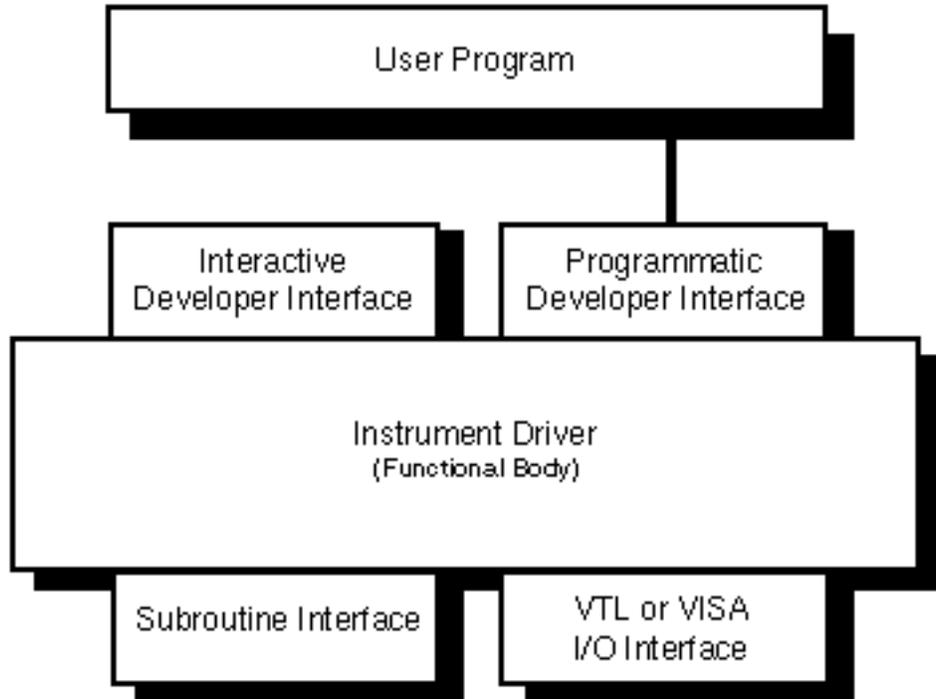


Figure 5-1. Instrument Driver External Interface Model

This general model contains the instrument driver *functional body*, which is the core of the instrument driver. The *programmable developer interface* to the instrument driver is the mechanism for calling the driver from a higher-level software program. The *interactive developer interface* is an interactive interface, often graphical, that assists the software developer in understanding what each particular instrument driver function does and how to use the programmable developer interface to call each function. The *VISA I/O interface* is the mechanism through which the driver communicates with the instrument hardware. The *subroutine interface* is the mechanism through which the driver may call other software modules it may need to perform its task. These other software modules may include operating system calls or calls to other unique libraries such as formatting and analysis functions.

5.2.1 Functional Body

The functional body of the instrument driver is the code for the driver. The details of the functional body are explained using the instrument driver internal design model beginning in Section 5.3. The functional body is further specified in the *Instrument Driver Functional Body Specification* (VPP-3.2).

The most successful instrument driver products have historically taken the approach that the functional body of the driver be developed by writing (or drawing) some sort of program using a standard programming language. This is the approach used by *VXIplug&play* instrument drivers. The advantages are that the driver can implement more functionality, and the developer has more control over what the driver does. *VXIplug&play* instrument drivers in non-G frameworks are written using ANSI C. *VXIplug&play* instrument drivers in G frameworks are written in G.

5.2.2 VISA I/O Interface

An important consideration for instrument drivers is how they perform I/O to and from instruments. In the *VXIplug&play* architecture, the I/O interface is provided by a separate layer of software that is standard and available on numerous platforms.

This standard *VXIplug&play* I/O interface is called VISA (Virtual Instrument Software Architecture), and is documented in the VPP-4.X specifications. VISA includes a single interface library for controlling VXI, GPIB, RS-232, Ethernet, and other types of instrument interfaces. It provides access to all of the VXI capabilities, including both message-based and register-based programming, interrupt and event handling, and direct access to the VXI backplane.

RULE 5.1

VXIplug&play instrument drivers **SHALL** call only the VISA I/O Library (VPP-4.X) to control instruments.

5.2.3 Subroutine Interface

VXIplug&play instrument drivers are written using standard programming techniques and hierarchical design. Therefore, instrument drivers can access and use the features of other libraries through a *Subroutine Interface*. For example, instrument drivers use the subroutine interface to access VXI controller capabilities available from the VISA I/O interface. Additionally, the instrument driver may access other support libraries with the same technique.

RULE 5.2

IF a *VXIplug&play* instrument driver calls any support library functions that are not part of the instrument driver source code, **THEN** the support library functions **SHALL** be provided with the instrument driver.

RULE 5.3

Support libraries **SHALL** be functional on the minimum required system for a given framework.

PERMISSION 5.1

Support library functions are not required to be distributed in source code.

5.2.4 Programmatic Developer Interface

The programmatic developer interface is the mechanism for using the instrument driver as part of a test program application. With *VXIplug&play* instrument drivers, the driver consists of component functions and one or more application functions to control the instrument. The programmatic developer interface to these modular software functions is a standard software function.

With a high-level function call interface to instrument drivers, the end user's resultant test program code consists of a few calls to the instrument driver, each call using multiple parameters. A key benefit of this approach is that the interface to the instrument driver in the user's program is modular and easy to identify, and any interactive developer interface tools (discussed in the next section) that were used during development of the user's code can be recalled during debugging to understand how the program uses the instrument driver. The programmatic developer interface is documented in the *Instrument Driver Programmatic Developer Interface Specification* (VPP-3.4).

5.2.5 Interactive Developer Interface

When a *VXIplug&play* instrument driver is used as an integral part of a higher-level application software development environment, the programmatic developer interface to the instrument driver can be enhanced with graphical software panels or other tools. Such tools, referred to as the *interactive developer interface*, are designed to assist the programmer who is writing application software by making it easier to understand how to use the instrument driver. These tools can allow the programmer to operate the particular instrument driver function interactively to understand what the function does, and may assist the programmer in calling the function from their own application program. Such tools may even automate the process through automatic code-generation techniques.

One example of an interactive developer interface is a function panel which is used by ADEs in the WINNT framework. A function panel file represents the functions in an instrument driver, the parameter names and data types of each function, and associated help text. When used in conjunction with ADEs such as LabWindows/CVI, LabVIEW, and Agilent VEE, the panels graphically represent the instrument driver functions and allow the programmer to operate the functions interactively. Additionally, the programmer can view the help text to understand the driver operation, and can automatically generate instrument control code for his application.

When a *VXIplug&play* instrument driver is used in an environment that cannot interpret the function panel, such as Microsoft C, the programmatic developer interface to the instrument driver is still standard software function calls, but the interactive developer interface (function panel) is not available to assist the programmer. The documentation on how to use the

instrument driver in this environment is available by another means specific to that environment, such as a Windows help file for the WINNT framework.

The interactive developer interface is documented in the *Instrument Driver Interactive Developer Interface Specification* (VPP-3.3).

5.3 Instrument Driver Internal Design Model

This section gives an overview of the second architectural model used in this specification. The instrument driver internal design model, shown in Figure 5-2, defines the internal organization of the functional body of the driver. This model is especially important to instrument driver developers because it is the foundation upon which the development guidelines are based. As such, this model is used heavily in the remaining VPP-3.X specifications. It is also important to end users because all *VXIplug&play* instrument drivers are organized according to this model. Once users understand the model and how to use one instrument driver, their knowledge regarding the use of that instrument driver is highly leveraged across other instrument drivers.

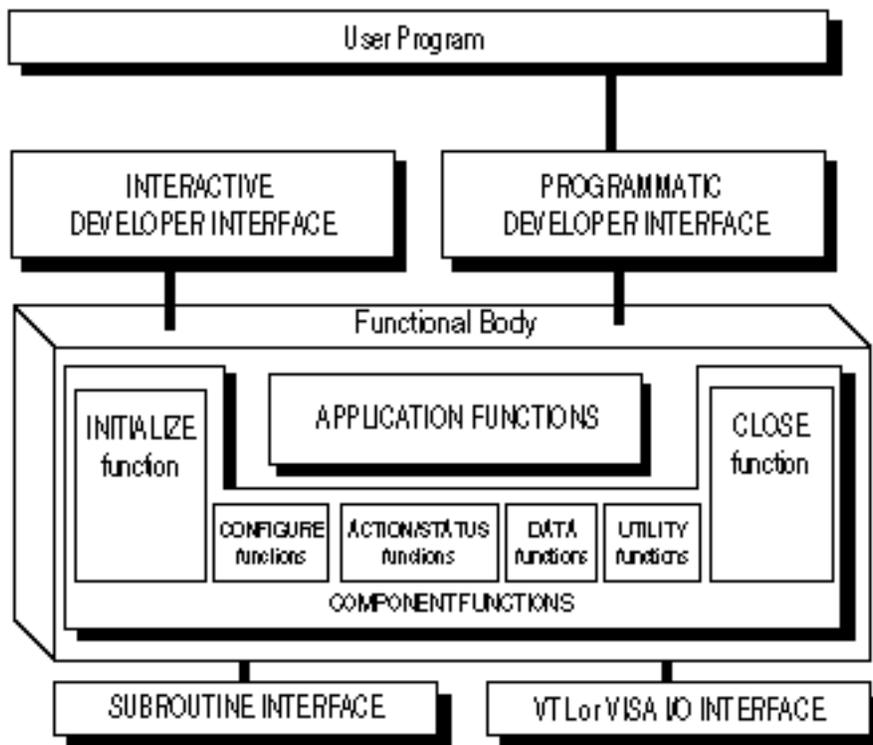


Figure 5-2. Instrument Driver Internal Design Model

Every instrument provides control over its capabilities through a command set or register set. The commands and registers manipulate the instrument hardware to control specific instrument

capabilities. Therefore the command or register set define the programmatic capabilities of an instrument.

By definition, the command set or register set gives the user full-function control of an instrument. Interacting with an instrument at this level often requires users understand complex low-level command syntax, command interdependencies, and state transitions. Many application programs do not need this low level control and can utilize a higher-level instrument driver functions that combine instruments capabilities to accomplish a task.

High-level instrument driver functions may contain expert knowledge of the instrument control language and manage command sequence dependencies, state dependencies and error management. The functional interface presented to the application writer hides these complex interactions reducing the level of user knowledge required to utilize the instrument.

The *Instrument Driver Internal Design Model* is designed to give application developers both high and low-level access to the instrument's capabilities. At the highest level are the *Application Functions*. *Application Functions* are a collection of high-level functions that perform complete test and measurement operations. These functions tend to be more abstract and are usually viewed as the standard capabilities for a given instrument.

At a lower level, *Component Functions* control a specific area of the instrument's functionality. These functions give application developers an intermediate level access to the instruments capabilities. These functions typically decompose to lower levels of instrument control.

The modularity of the *Instrument Driver Internal Design Model* builds on proven technology. With a modular approach, a user has the granularity needed to control instruments properly in their software application. With this model users begin a transaction with an instrument by initializing the device. Then they can repetitively call Application functions and Component functions to control the operation of the instrument. As a last step, users close the device to terminate communications with the device.

The user can, for example, initialize all instruments once at the start, configure multiple instruments, and then trigger several instruments simultaneously. As another example, a user can initialize and configure an instrument once, and then retrieve measurements from the instrument several times.

RULE 5.4

The functional body of a *VXIplug&play* instrument driver **SHALL** follow the instrument driver internal design model shown in Figure 5-2.

RULE 5.5

A *VXIplug&play* instrument driver for a particular instrument **SHALL** support multiple instruments of that type in the system.

OBSERVATION 5.1

The above rule (RULE 5.5) does not imply that an instrument driver must support more than one application using the same instrument simultaneously.

5.3.1 Application Functions

The *Application Functions* are high-level routines that are also provided in source code. These are complete instrument driver functions that can be called via their own program interface when a user wants a single, high-level function interface to the driver. In most cases, these functions are complete operations that configure, start, and read the instrument.

An Application function is typically designed for a specific application domain. This application specific representation contains expert knowledge of both instrument control and the particular application area. An instrument driver may have more than one application domain and each application domain could have more than one Application Function.

RULE 5.6

All *VXIplug&play* instrument drivers **SHALL** include one or more application functions.

PERMISSION 5.2

In some instruments, the Application function may be the same as the capability component function.

The application functions are required not only because they provide a valuable example of how to control an instrument, but also because they are useful when users want a single-function interface to the driver rather than using the individual component functions.

RULE 5.7

VXIplug&play instrument driver application functions **SHALL NOT** call the initialize or close instrument driver component functions.

5.3.2 Component Functions

The *Component Functions* divide instrument capabilities into classes that control a specific area of the instrument's functionality. These functions give application developers an intermediate level access to the instrument's capabilities and typically decompose into lower-levels of instrument control.

The component functions are divided into four categories: *initialize*, one or more *capability classes*, *utility* functions, and *close*. Each of these categories may consist of several modular software routines. The specific routines in each category are further organized as either *required functions* or *developer-specified functions*.

The required functions are instrument driver functions that are common to the majority of instruments. These functions perform such operations as initialization, reset, self-test, error handling and close. The required instrument driver functions are defined in the *Instrument Driver Functional Body Specification* (VPP-3.2). All instrument driver function names have a PREFIX as defined in the *Instrument Driver Functional Body Specification* (VPP-3.2).

The remainder of instrument driver functions are known as developer-specified functions, and the actual operations performed by those routines are left up to the instrument driver developer. General guidelines in the *Instrument Driver Developer Guidelines* (VPP-3.A) define, organize, and structure the functions within each category. By following these guidelines, similar instruments will have similar sets of functions. In addition, example implementations for a variety of types of instruments are included in the *Instrument Driver Developer Guidelines* (VPP-3.A).

The *VXIplug&play* instrument driver specifications recommend that instrument drivers provide full function control of the instrument. They do not, however, attempt to mandate the required functionality of all instrument types, such as DMMs, counter/timers, and so on. Rather, the focus is on the architectural guidelines of all drivers. In this way, all driver developers have the flexibility to implement functionality unique to a particular instrument, yet all drivers are organized, packaged, and used in the same way.

5.3.3 Initialize Function

The initialize function initializes the software connection to the instrument. In addition, it may perform any necessary actions to place the instrument in its default power-on state or other specific state.

RULE 5.8

All *VXIplug&play* instrument drivers **SHALL** have an initialize function.

5.3.4 Capability Classes

Capability class functions group the instrument driver functions according to instrument's capabilities such as *measure*, *source*, *route*, etc. At the highest level within a capability class, each function executes a complete action. No prior instrument state is assumed. This allows these functions to be order independent. Execution of a capability class function produces a finished result such as a returned value of a measurement, production of the specified output by a source, or a connected route by a switch.

OBSERVATION 5.2

The capability classes decompose into lower-level function sub-classes that present more options and granularity. As a user moves to lower-level capability class functions, he gains access to more instrument capabilities and finer control over instrument operations. Typically, the instrument drivers for more complex instruments will tend to have more sub-classes.

For example, the measure class is divided into *configure* and *read* function classes. Configure functions configure the device for a particular measurement but do not execute the measurement. Read functions initiate the measurement and read the result from the instrument. Users can choose a single measure class function to configure, initiate, and read data from an instrument all in a single operation. However, if the user needs more control over the operation, he can use the configure functions to configure the device and read functions to initiate the measurement and read the result.

RULE 5.9

All *VXIplug&play* instrument drivers **SHALL** have one or more capability classes.

5.3.4.1 Measure

Functions in the measure class configure a device for a particular measurement, initiate that measurement, and read the result in a single operation. Typically, these functions include multiple parameters and require no interaction with other instrument driver operations. Figure 5-3 shows the model for the measure functions class.

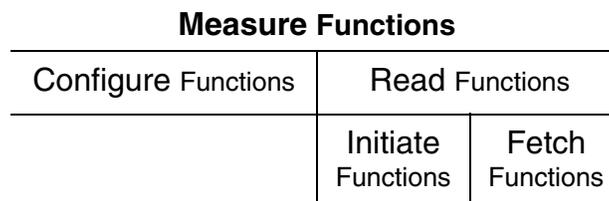


Figure 5-3. Measure Capability Class Model

RULE 5.10

Measure class functions **SHALL** configure a device for a particular measurement, initiate that measurement, and read the result in a single operation.

RULE 5.11

IF a VXI*plug&play* instrument driver implements sub-classes in the measure capability class, it **SHALL** include the sub-classes Configure and Read.

5.3.4.1.1 Configure and Read Functions

Configure and read functions of the measure class provide a highly abstract control interface to the capabilities of a measurement device. These functions correspond to the measure functions except they separate the functionality of configuration from the functionality of reading measured values from the instrument.

Configure functions configure an instrument for a particular measurement but do not initiate that measurement. No finished result is provided. Read functions complete the measurement function action by initiating the measurement and providing the result. Read functions have a specific sequence dependency with their complimentary configure function.

RULE 5.12

Configure sub-class functions of the Measure class **SHALL** only configure a device for a particular measurement and not perform that measurement.

RULE 5.13

Read class functions **SHALL** only initiate a measurement action and return the result. Read class functions **SHALL NOT** modify the device configuration.

The read functions depend on the instrument state produced by the configure functions. However, because the configure and read functions are separated, there is an opportunity to send additional low level commands which modify the instrument state prior to calling the read function. This allows the default action of a measurement function to be modified, providing more control over the test execution. For example, the application program may choose to defer measurement execution until all of the instrument modules have been configured.

Read functions may be further divided into initiate and fetch functions. These functions separate the initiation of a measurement from the data retrieval. This, gives the user finer control of the measurement operation.

5.3.4.2 Source

Functions in the source class configure a device to output a particular stimulus, and initiate the application of that stimulus in a single operation. Typically, these functions include multiple parameters and require no interaction with other instrument driver operations. Figure 5-4 shows the model for the source functions class.

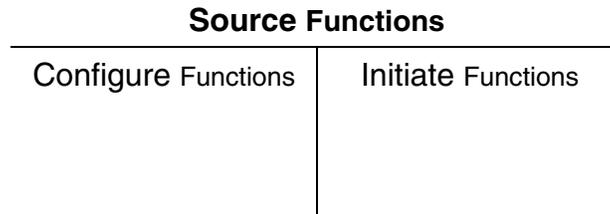


Figure 5-4. Source Capability Class Model

RULE 5.14

Source class functions **SHALL** configure a device to output a particular stimulus, and initiate the application of that stimulus in a single operation.

RULE 5.15

IF a *VXIplug&play* instrument driver implements sub-classes in the Source capability class, it **SHALL** at least include the sub-classes for either Configure or Initiate, or both.

5.3.4.2.1 Configure and Initiate Functions

Configure and initiate functions correspond to the source functions except they separate the functionality of configuration from the functionality of outputting the configured stimulus. Configure functions configure an instrument for a particular stimulus. Initiate functions complete pending source actions. Initiate functions have a specific sequence dependency with their complimentary configure function.

RULE 5.16

Configure class functions of the Source class **SHALL** only configure a device for a particular stimulus.

OBSERVATION 5.3

If a source operation is already initiated, configuring the device may change some characteristics of the output.

RULE 5.17

Initiate class functions of the Source class **SHALL** only output the configured stimulus. Initiate class functions **SHALL NOT** modify the device configuration.

The initiate functions depend on the instrument state produced by the configure functions. However, because the configure and initiate functions are separated, there is an opportunity to send additional low level commands which modify the instrument state prior to calling the initiate function. This allows the default action of a source function to be modified, providing more control over the test execution. For example, the application program may choose to defer source execution until all of the instrument modules have been configured.

5.3.4.3 Route

A route function routes a signal in a single operation. Typically, these functions include multiple parameters and require no interaction with other instrument driver operations. Figure 5-5 shows the model for the route functions class.

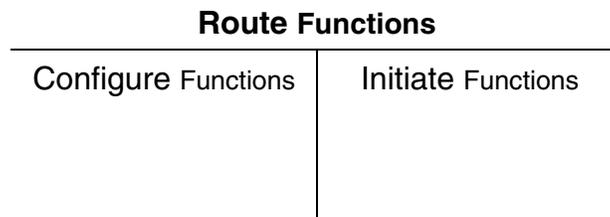


Figure 5-5. Route Capability Class Model

RULE 5.18

Route class functions **SHALL** route a signal in a single operation.

RULE 5.19

If a *VXIplug&play* instrument driver implements sub-classes in the Route capability class, it **SHALL** at least include the sub-classes for either Configure or Initiate, or both.

5.3.4.3.1 Configure and Initiate Functions

Configure and initiate functions correspond to the route functions except they separate the functionality of configuration from the functionality of initiating the configured routing. Configure functions configure an instrument for a particular routing operation. Initiate functions complete pending route actions. Initiate functions have a specific sequence dependency with their complimentary configure function.

RULE 5.20

Configure class functions of the Route class **SHALL** only configure a device for a particular routing operation.

RULE 5.21

Initiate class functions of the Route class **SHALL** only establish the configured route. Initiate class functions **SHALL NOT** modify the device configuration.

The initiate functions depend on the instrument state produced by the configure functions. However, because the configure and initiate functions are separated, there is an opportunity to send additional low level commands which modify the instrument state prior to calling the initiate function. This allows the default action of a route function to be modified, providing more control over the test execution. For example, the application program may choose to defer route execution until all of the instrument modules have been configured.

5.3.5 Utility Functions

The utility functions can perform a variety of operations. Some utility functions are required and some are defined by the developer.

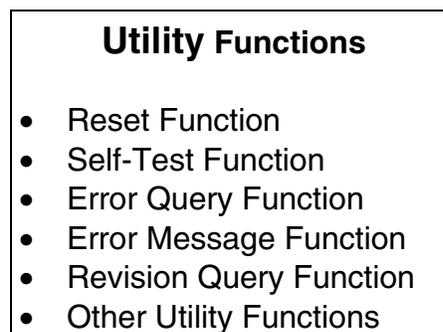


Figure 5-6. Utility Functions

RULE 5.22

All *VXIplug&play* instrument drivers **SHALL** have a class of functions called Utility.

RULE 5.23

The utility functions class **SHALL** contain the Reset, Self-Test, Error Query, and Revision Query functions.

RULE 5.14

IF a *VXIplug&play* instrument driver implements the Error Message function, it **SHALL** be included in the utility functions class.

OBSERVATION 5.4

The Error Message function is required by C language *VXIplug&play* drivers but not G language *VXIplug&play* instrument drivers. G language *VXIplug&play* instrument drivers make use other error handling techniques within the ADE to retrieve error information.

5.3.6 Close Function

The Close function terminates an instrument session and deallocates system resources associated with that session.

RULE 5.25

All *VXIplug&play* instrument drivers **SHALL** have a close function.