NATIONAL INSTRUMENTS™
Measurement Studio™

Measurement Studio LabWindows/CVI Instrument Driver Developers Guide

NATIONAL INSTRUMENTS™

July 2001 Edition
Part Number 320684F-01

Worldwide Technical Support and Product Information

ni.com

National Instruments Corporate Headquarters

11500 North Mopac Expressway Austin, Texas 78759-3504 USA Tel: 512 794 0100

Worldwide Offices

Australia 03 9879 5166, Austria 0662 45 79 90 0, Belgium 02 757 00 20, Brazil 011 284 5011,
Canada (Calgary) 403 274 9391, Canada (Montreal) 514 288 5722, Canada (Ottawa) 613 233 5949,
Canada (Québec) 514 694 8521, Canada (Toronto) 905 785 0085, China (Shanghai) 021 6555 7838,
China (ShenZhen) 0755 3904939, Denmark 45 76 26 00, Finland 09 725 725 11, France 01 48 14 24 24,
Germany 089 741 31 30, Greece 30 1 42 96 427, Hong Kong 2645 3186, India 91805275406,
Israel 03 6120092, Italy 02 413091, Japan 03 5472 2970, Korea 02 596 7456, Malaysia 603 9596711,
Mexico 5 280 7625, Netherlands 0348 433466, New Zealand 09 914 0488, Norway 32 27 73 00,
Poland 0 22 528 94 06, Portugal 351 1 726 9011, Singapore 2265886, Spain 91 640 0085,
Sweden 08 587 895 00, Switzerland 056 200 51 51, Taiwan 02 2528 7227, United Kingdom 01635 523545

For further support information, see the *Technical Support Resources* appendix. To comment on the documentation, send e-mail to `techpubs@ni.com`.

Copyright © 1994, 2001 National Instruments Corporation. All rights reserved.

Important Information

Warranty

The media on which you receive National Instruments software is warranted against defects in materials and workmanship for a period of 90 days from date of shipment, as evidenced by receipts or other documentation. National Instruments will, at its option, repair or replace equipment that proves to be defective during the warranty period. This warranty includes parts and labor.

The media on which you receive National Instruments software are warranted not to fail to execute programming instructions, due to defects in materials and workmanship, for a period of 90 days from date of shipment, as evidenced by receipts or other documentation. National Instruments will, at its option, repair or replace software media that do not execute programming instructions if National Instruments receives notice of such defects during the warranty period. National Instruments does not warrant that the operation of the software shall be uninterrupted or error free.

A Return Material Authorization (RMA) number must be obtained from the factory and clearly marked on the outside of the package before any equipment will be accepted for warranty work. National Instruments will pay the shipping costs of returning to the owner parts which are covered by warranty.

National Instruments believes that the information in this document is accurate. The document has been carefully reviewed for technical accuracy. In the event that technical or typographical errors exist, National Instruments reserves the right to make changes to subsequent editions of this document without prior notice to holders of this edition. The reader should consult National Instruments if errors are suspected. In no event shall National Instruments be liable for any damages arising out of or related to this document or the information contained in it.

EXCEPT AS SPECIFIED HEREIN, NATIONAL INSTRUMENTS MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AND SPECIFICALLY DISCLAIMS ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. CUSTOMER'S RIGHT TO RECOVER DAMAGES CAUSED BY FAULT OR NEGLIGENCE ON THE PART OF NATIONAL INSTRUMENTS SHALL BE LIMITED TO THE AMOUNT THERETOFORE PAID BY THE CUSTOMER. NATIONAL INSTRUMENTS WILL NOT BE LIABLE FOR DAMAGES RESULTING FROM LOSS OF DATA, PROFITS, USE OF PRODUCTS, OR INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF. This limitation of the liability of National Instruments will apply regardless of the form of action, whether in contract or tort, including negligence. Any action against National Instruments must be brought within one year after the cause of action accrues. National Instruments shall not be liable for any delay in performance due to causes beyond its reasonable control. The warranty provided herein does not cover damages, defects, malfunctions, or service failures caused by owner's failure to follow the National Instruments installation, operation, or maintenance instructions; owner's modification of the product; owner's abuse, misuse, or negligent acts; and power failure or surges, fire, flood, accident, actions of third parties, or other events outside reasonable control.

Copyright

Under the copyright laws, this publication may not be reproduced or transmitted in any form, electronic or mechanical, including photocopying, recording, storing in an information retrieval system, or translating, in whole or in part, without the prior written consent of National Instruments Corporation.

Trademarks

CVI™, IVI™, National Instruments™, NI™, and ni.com™ are trademarks of National Instruments Corporation.

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

WARNING REGARDING USE OF NATIONAL INSTRUMENTS PRODUCTS

(1) NATIONAL INSTRUMENTS PRODUCTS ARE NOT DESIGNED WITH COMPONENTS AND TESTING FOR A LEVEL OF RELIABILITY SUITABLE FOR USE IN OR IN CONNECTION WITH SURGICAL IMPLANTS OR AS CRITICAL COMPONENTS IN ANY LIFE SUPPORT SYSTEMS WHOSE FAILURE TO PERFORM CAN REASONABLY BE EXPECTED TO CAUSE SIGNIFICANT INJURY TO A HUMAN.

(2) IN ANY APPLICATION, INCLUDING THE ABOVE, RELIABILITY OF OPERATION OF THE SOFTWARE PRODUCTS CAN BE IMPAIRED BY ADVERSE FACTORS, INCLUDING BUT NOT LIMITED TO FLUCTUATIONS IN ELECTRICAL POWER SUPPLY, COMPUTER HARDWARE MALFUNCTIONS, COMPUTER OPERATING SYSTEM SOFTWARE FITNESS, FITNESS OF COMPILERS AND DEVELOPMENT SOFTWARE USED TO DEVELOP AN APPLICATION, INSTALLATION ERRORS, SOFTWARE AND HARDWARE COMPATIBILITY PROBLEMS, MALFUNCTIONS OR FAILURES OF ELECTRONIC MONITORING OR CONTROL DEVICES, TRANSIENT FAILURES OF ELECTRONIC SYSTEMS (HARDWARE AND/OR SOFTWARE), UNANTICIPATED USES OR MISUSES, OR ERRORS ON THE PART OF THE USER OR APPLICATIONS DESIGNER (ADVERSE FACTORS SUCH AS THESE ARE HEREAFTER COLLECTIVELY TERMED "SYSTEM FAILURES"). ANY APPLICATION WHERE A SYSTEM FAILURE WOULD CREATE A RISK OF HARM TO PROPERTY OR PERSONS (INCLUDING THE RISK OF BODILY INJURY AND DEATH) SHOULD NOT BE RELIANT SOLELY UPON ONE FORM OF ELECTRONIC SYSTEM DUE TO THE RISK OF SYSTEM FAILURE. TO AVOID DAMAGE, INJURY, OR DEATH, THE USER OR APPLICATION DESIGNER MUST TAKE REASONABLY PRUDENT STEPS TO PROTECT AGAINST SYSTEM FAILURES, INCLUDING BUT NOT LIMITED TO BACK-UP OR SHUT DOWN MECHANISMS. BECAUSE EACH END-USER SYSTEM IS CUSTOMIZED AND DIFFERS FROM NATIONAL INSTRUMENTS' TESTING PLATFORMS AND BECAUSE A USER OR APPLICATION DESIGNER MAY USE NATIONAL INSTRUMENTS PRODUCTS IN COMBINATION WITH OTHER PRODUCTS IN A MANNER NOT EVALUATED OR CONTEMPLATED BY NATIONAL INSTRUMENTS, THE USER OR APPLICATION DESIGNER IS ULTIMATELY RESPONSIBLE FOR VERIFYING AND VALIDATING THE SUITABILITY OF NATIONAL INSTRUMENTS PRODUCTS WHENEVER NATIONAL INSTRUMENTS PRODUCTS ARE INCORPORATED IN A SYSTEM OR APPLICATION, INCLUDING, WITHOUT LIMITATION, THE APPROPRIATE DESIGN, PROCESS AND SAFETY LEVEL OF SUCH SYSTEM OR APPLICATION.

Conventions

The following conventions are used in this manual:

»	The » symbol leads you through nested menu items and dialog box options to a final action and through the Table of Contents in the LabWindows/CVI Help to a topic. The sequence **File»Page Setup»Options** directs you to pull down the **File** menu, select the **Page Setup** item, and select **Options** from the last dialog box.
	This icon denotes a note, which alerts you to important information.
	This icon denotes a caution, which advises you of precautions to take to avoid injury, data loss, or a system crash.
bold	Bold text denotes items that you must select or click on in the software, such as menu items and dialog box options. Bold text also denotes parameter names.
italic	Italic text denotes variables, emphasis, a cross reference, or an introduction to a key concept. This font also denotes text that is a placeholder for a word or value that you must supply.
`monospace`	Text in this font denotes text or characters that you should enter from the keyboard, sections of code, programming examples, and syntax examples. This font is also used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, operations, variables, filenames and extensions, and code excerpts.
`monospace italic`	Italic text in this font denotes that you must enter the appropriate words or values in the place of these items.

Contents

Chapter 1
Instrument Driver Overview

What Is an Instrument Driver? 1-1
Historical Evolution of Instrument Drivers 1-2
Types of IVI Drivers 1-3
About Instrument Drivers 1-4
How Users Operate the Instrument Driver 1-5
Instrument Driver Architecture 1-5
- Instrument Driver External Interface Model 1-5
  - Functional Body 1-7
  - Programmatic Developer Interface 1-7
  - Interactive Developer Interface 1-7
  - IVI Engine 1-7
  - VISA I/O Interface 1-7
  - Subroutine Interface 1-8
- Instrument Driver Internal Design Model 1-8
  - Application Functions 1-10
  - Component Functions 1-10
  - Initialization Functions 1-12
  - Configuration Functions 1-12
  - Action/Status Functions 1-12
  - Measurement Functions 1-13
  - Utility Functions 1-13
  - Close Function 1-14
  - Attribute Functions 1-14
  - Callback Functions 1-14
  - Achieving Interchangeability 1-14

Chapter 2
IVI Architecture Overview

What is IVI? 2-1
Introduction to IVI Instrument Drivers 2-2
- How IVI Instrument Drivers Work 2-3
- Programming with IVI Instrument Drivers 2-5

Driver Functions and Attribute Model 2-6
- Types of Attributes 2-7
- Get/Set/Check Functions 2-7
- Callbacks 2-8

Creating and Declaring Attributes 2-9
Attribute IDs 2-9
Inherent IVI Attributes 2-9
Class Attributes 2-9
Instrument-Specific Attributes 2-10
Attribute Flags 2-10
Range Tables 2-12
Range Table Structures 2-13
Discrete Range Table Example 2-15
Coerced Range Table Example 2-15
Ranged Range Table Example 2-16
Static and Dynamic Range Tables 2-16
Default Check and Coerce Callbacks 2-17
Comparison Precision 2-17
IVI State-Caching Mechanism 2-18
Initial Instrument State 2-19
Special Cases 2-19
Changing the Value of One Attribute Invalidates Another 2-19
Two Attributes Invalidate Each Other 2-20
Setting or Getting the Values of Two Attributes in One Command 2-20
Instrument Coerces Values 2-20
Enabling and Disabling State-Caching 2-21
Attribute Callback Functions 2-21
Read Callback 2-22
Write Callback 2-22
Check Callback 2-23
Coerce Callback 2-23
Compare Callback 2-24
Range Table Callback 2-25
Session Callback Functions 2-25
Operation Complete Callback 2-25
Check Status Callback 2-26
Instruments without Error Queues 2-27
Channels 2-27
Virtual Channel Names 2-28
Passing Channel Names to IVI Functions 2-28
Coercing and Validating Channel Names 2-28
High-Level Driver Functions 2-28
Range Checking 2-29
Status Checking 2-30
Simulation 2-31
Multithread Safety 2-32
Configuration Entries 2-33

Inherent IVI Attributes....2-34
Inherent Attribute Reference....2-35
IVI_ATTR_CACHE....2-35
IVI_ATTR_CHECK_STATUS_CALLBACK....2-36
IVI_ATTR_CLASS_MAJOR_VERSION....2-36
IVI_ATTR_CLASS_MINOR_VERSION....2-36
IVI_ATTR_CLASS_PREFIX....2-37
IVI_ATTR_CLASS_REVISION....2-37
IVI_ATTR_DRIVER_MAJOR_VERSION....2-37
IVI_ATTR_DRIVER_MINOR_VERSION....2-38
IVI_ATTR_DRIVER_REVISION....2-38
IVI_ATTR_DRIVER_SETUP....2-38
IVI_ATTR_ENGINE_MAJOR_VERSION....2-38
IVI_ATTR_ENGINE_MINOR_VERSION....2-39
IVI_ATTR_ENGINE_REVISION....2-39
IVI_ATTR_ERROR_ELABORATION....2-39
IVI_ATTR_FUNCTION_CAPABILITIES....2-39
IVI_ATTR_GROUP_CAPABILITIES....2-39
IVI_ATTR_INTERCHANGE_CHECK....2-40
IVI_ATTR_IO_SESSION....2-40
IVI_ATTR_LOGICAL_NAME....2-40
IVI_ATTR_MODULE_PATHNAME....2-41
IVI_ATTR_NUM_CHANNELS....2-41
IVI_ATTR_OPC_CALLBACK....2-41
IVI_ATTR_PRIMARY_ERROR....2-42
IVI_ATTR_QUERY_INSTR_STATUS....2-42
IVI_ATTR_RANGE_CHECK....2-42
IVI_ATTR_RECORD_COERCIONS....2-43
IVI_ATTR_RESOURCE_DESCRIPTOR....2-44
IVI_ATTR_SECONDARY_ERROR....2-44
IVI_ATTR_SIMULATE....2-44
IVI_ATTR_SPECIFIC_PREFIX....2-45
IVI_ATTR_SPY....2-45
IVI_ATTR_VISA_RM_SESSION....2-45

Chapter 3
Programming Guidelines for Instrument Drivers

General Guidelines....3-1
Writing an Instrument Driver....3-2
Naming the Driver....3-3
Using the Instrument Driver Development Wizard....3-3
Selecting an Instrument Driver Template....3-5
Running the Preliminary I/O Tests from the Wizard....3-7

Reviewing the Generated Driver Files 3-7
Generated Function Panels 3-7
.sub file 3-7
Source File 3-8
Include File 3-9
Extended Functions and Attributes 3-9
Customizing Wizard-Generated Driver Files 3-10
Attribute Editor 3-10
Modifying Existing Attributes and Functions 3-10
Deleting the Attributes and Functions 3-11
Adding New Attributes and Functions 3-11
General Modifications 3-12
Instrument Driver Attributes 3-12
Attribute ID Values 3-12
Attribute Value Definitions 3-13
Simulation 3-14
Data Types 3-15
Callbacks 3-15
Read and Coerce Callbacks for ViString Attributes 3-15
Write Callbacks 3-16
Reading Strings From the Instrument 3-16
Range Table Callbacks 3-18
Range Tables 3-18
Attribute Examples 3-19
Attributes that Represent Discrete Settings 3-19
Attributes that Represent a Continuous Range 3-21
Attributes that Represent a Continuous Range with Discrete Settings 3-22
Attributes with a Changing Valid Range 3-24
Check, Coerce, and Compare Callbacks 3-29
User-Callable Functions 3-30
Instrument Driver Function Structure 3-30
Locking/Unlocking the Session 3-33
Parameter Checking 3-33
Accessing Attributes 3-34
Performing Direct Instrument I/O 3-34
Simulating Output Parameters 3-34
Checking the Instrument Status 3-35
Functions that Only Set Attributes 3-36
Initialization Functions 3-37
Channel Strings 3-37
Close Functions 3-38

Developing Portable Instrument Drivers 3-38
Instrument Driver Data Types 3-38
Declaring Instrument Driver Functions 3-39
Using Scan and Fmt Functions 3-40
Error-Reporting Guidelines 3-41
General Programming Guidelines 3-42
Function Panels 3-43
Function Tree Hierarchy 3-43
Documentation Guidelines 3-44
Online Help 3-45
Documentation Files 3-49
Programming Guidelines for VXI Instruments 3-50
Instrument Driver Checklist 3-50

Chapter 4
Attribute Editor

Invoking the Attribute Editor 4-1
Requirements for Using the Attribute Editor 4-1
Limitations in Updates to Driver Files 4-2
Edit Driver Attributes Dialog Box 4-3
Instrument Attributes List Box 4-3
Restrictions on Modifications to Inherent and Class Attributes 4-4
Attributes List Box Command Buttons 4-4
Adding and Editing Instrument Attributes 4-6
Adding and Editing Range Tables 4-8

Chapter 5
Instrument Driver Examples

Example 1—Creating IVI Driver Files with the Instrument Driver Development Wizard 5-2
Example 2—Editing the Instrument Driver Attributes 5-12
Customizing the Measurement Function Attribute 5-13
Modifying the Write and Read Callbacks for the Measurement Function Attribute 5-16
Deleting Unused Attributes 5-17
Example 3—Editing High-Level Instrument Driver Functions 5-20
Editing the Fetch Function 5-21
Deleting Functions the Instrument Does Not Use 5-25
Example 4—Adding New Attributes and Functions 5-26
Adding the Hold Enable Attribute 5-26
Adding the Hold Threshold Attribute 5-28

Adding the Configure Hold Function Panel 5-30
Creating the Configure Hold Function Body 5-35
Example 5—Creating Instrument Driver Documentation..... 5-36
Creating the Instrument Driver .doc File 5-36
Creating Windows Help for the Driver..... 5-37
Example 6—Modifying an Existing IVI Driver to Work with a New Instrument..... 5-37

Appendix A
Technical Support Resources

Glossary

Index

1

Instrument Driver Overview

The *Measurement Studio LabWindows/CVI Instrument Driver Developers Guide* describes developing and adding instrument drivers to the LabWindows/CVI Instrument Library. This guide is for users who develop instrument drivers to control programmable instruments such as GPIB, VXI, and RS-232 instruments. The software tools you use to create instrument drivers are included in the standard LabWindows/CVI package.

The *Measurement Studio LabWindows/CVI Instrument Driver Developers Guide* is for users familiar with LabWindows/CVI fundamentals. This manual assumes that you are familiar with the material presented in the *Getting Started with Measurement Studio LabWindows/CVI* manual and the *LabWindows/CVI Help*, and that you are comfortable with the LabWindows/CVI software. Refer to the *Using LabWindows/CVI* section of the *LabWindows/CVI Help* for specific instructions on operating LabWindows/CVI.

This chapter introduces the concept of instrument drivers, explains how they have evolved, and describes their general structure.

What Is an Instrument Driver?

Programmers of early computer-controlled instrumentation systems used BASIC I/O statements in their applications to send and receive command and data strings to an instrument connected to their computer via GPIB. Each instrument responded to particular ASCII strings as documented in each instrument's user manual. Programmers had to learn each instrument's command set and write the control program.

Because programming was often the most time-consuming part of developing an automated test system—especially if programmers had to use a different command set for each instrument—this effect was compounded when programmers had to repeat their work when they created new applications with the same instruments. Eventually, programmers realized they could save much time and money if they wrote high-level routines that hid the low-level commands and were generic and modular enough to be reused in any future application that used the same instrument. These reusable routines became known as instrument drivers.

Historical Evolution of Instrument Drivers

Although the instrument driver concept had promise, early implementations had serious limitations. Some approaches were too closely linked to proprietary development. Others were too difficult to develop or modify. Users wanted drivers that were open, modifiable, and built around standards that allowed instruments from a variety of vendors to peacefully coexist in one application.

The VXI*plug&play* Systems Alliance was founded to address system-level software issues beyond the scope of the VXIbus Consortium, and they actively worked to improve existing instrument driver standards. The VXI*plug&play* instrument driver architecture leveraged existing popular technology by building on the successful LabWindows/CVI instrument driver standards.

Today, these standards use Virtual Instrumentation Software Architecture (VISA)-defined data types to define parameters of all instrument driver functions. For example, the return value is of type `ViStatus` (a 32-bit unsigned integer). These data types promote the portability of instrument drivers to new operating systems and programming languages. All instrument I/O is performed with VISA (Virtual Instrumentation Software Architecture) where possible. The initialize function is generic to the type of interface (GPIB or VXI) that you use to control the instrument. You pass all instrument addressing information to the initialize function via a string parameter.

However, the VXI*plug&play* model is not the last word on instrument drivers. The Interchangeable Virtual Instruments (IVI) Foundation was established to create open standards for instrument drivers that leverage the VXI*plug&play* model. Even though IVI stands for *Interchangeable Virtual Instruments*, the foundation addresses other system-level issues such as instrument simulation and higher performance. National Instruments IVI drivers comply with the IVI Foundation standards. Three of the most important features are as follows:

- **Interchangeability**—The IVI Foundation defines standard APIs for common types of instruments such as digital multimeters (DMM), oscilloscopes, function/arbitrary waveform generators, DC power supplies, switches, and so on. NI provides specific drivers and class drivers that comply with these standards. Developers can use the class drivers to implement hardware-independent test programs. By using the class drivers, developers can configure their test system to use different instruments without editing, recompiling, or re-linking their test programs.
- **Instrument simulation**—IVI drivers can simulate the operation of instruments when they are not available. National Instruments IVI drivers return simulated data from output parameters. With simulated data, developers can develop code for instruments even when the instruments are not available. All parameters are range checked, so that developers can determine if their program is configuring the instrument with valid settings even during simulation.

- **Instrument state-caching**—For standard VXI*plug&play* drivers, the state of the instrument is assumed to be unknown. Therefore, each measurement function sets up the instrument for the measurement even if the instrument is already configured correctly. National Instruments IVI drivers implement an attribute model that automatically caches the current state of the instrument. A driver function performs instrument I/O only when the instrument settings are different from what the function requires. This seemingly minor difference in approach leads to significant reductions in test time and cost.

For a comprehensive list of IVI features, refer to Chapter 2, *IVI Architecture Overview*. For more information on IVI drivers, refer to the IVI Foundation Web site at `www.ivifoundation.org`.

Types of IVI Drivers

The IVI Foundation has standardized on the following two driver architectures, IVI-C, based on ANSI C technology, and IVI-COM, based on Microsoft Component Object Model (COM) technology.

Note The drivers, tools, and methods described in this manual focus on the development of IVI-C drivers. Therefore, all references to IVI drivers in this manual refer to IVI-C drivers that are created using NI tools and that rely on the National Instruments IVI engine.

IVI-C drivers can be further classified as IVI-C specific drivers and IVI-C class drivers.

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

An IVI-C specific driver can be either a class-compliant driver or a custom driver. A class-compliant driver is an IVI specific driver that complies with one of the IVI Foundation defined class specifications, such as oscilloscope or DMM. A custom driver is an IVI specific driver that is not compliant with one of the defined IVI class specifications. Custom drivers are developed for specialized instruments such as an optical attenuator. This manual provides guidelines for creating both IVI-C class-compliant drivers and IVI-C custom drivers.

An IVI-C class driver is an IVI driver that you can use to interchange instruments when using IVI class-compliant specific drivers. NI provides advanced IVI-C class drivers as part of the IVI Driver Toolset or basic IVI-C class drivers, which are available for download at `ni.com/ivi`.

For more information on the IVI Driver Toolset, refer to the *IVI Driver Toolset User Manual*, available for free download at `ni.com/manuals`.

About Instrument Drivers

The purpose of an instrument driver is to control an instrument. The instrument can be a single physical instrument such as an oscilloscope or a DMM, a family of instruments that share common functions, or a hybrid instrument for which no single physical instrument exists.

In addition to controlling the instrument, an instrument driver formats the data it reads from the instrument into a form convenient for application programs. For example, the driver can convert a binary array of 2-byte wide numbers into an ASCII string or convert an ASCII string of X-Y coordinates into 2-integer arrays suitable for plotting.

An IVI instrument driver consists of the following six files.

- The source or object code, which can be a `.obj`, `.dll`, or `.c` file.
- The include (`.h`) file, which contains function declarations, constant definitions, and external declarations of global variables.
- The function panel file (`.fp`), which contains information that defines the function tree, the function panels, and the help text.
- The `.sub` file, which documents attributes and their possible values. The instrument driver user views the contents of the `.sub` file in certain instrument driver function panels. The instrument driver developer edits the contents of the `.sub` file through the **Tools»Edit Instrument Attributes** command.
- An ASCII text file (`.txt`), which contains driver specifications, such as models supported, driver version, and required software.
- A help file (`.hlp`), which contains documentation for the instrument driver.

The filenames of these six files consist of the driver name, followed by the appropriate extension. For example, if the instrument driver name for the Hewlett-Packard 34401A digital multimeter is hp34401a, its files are named `hp34401a.c` (`.obj` or `.dll`), `hp34401a.h`, `hp34401a.fp`, `hp34401a.sub`, `hp34401a.txt`, and `hp34401a.hlp`.

Although the IVI Foundation has chosen to define architectures and APIs for the ANSI C and COM interface technologies, the IVI specifications do not prevent driver developers from defining custom interfaces for languages and environments that are not standardized by the IVI Foundation, such as C++ and LabVIEW. Therefore, you can create and distribute additional files. However, this manual concentrates on developing IVI drivers within the LabWindows/CVI environment.

For more information on developing instrument drivers in LabWindows/CVI, refer to the **Using LabWindows/CVI»Instrument Drivers»Using Instrument Drivers** topic in the *LabWindows/CVI Help* installed with LabWindows/CVI.

How Users Operate the Instrument Driver

To the user, an instrument driver is a set of functions that perform instrument actions. Within LabWindows/CVI, the user selects an instrument driver from the **Instrument** menu. After selecting an instrument, the user selects a function within the instrument driver. A function panel appears that represents that instrument driver function.

A function panel displays symbolic controls that represent the parameters of the function. By manipulating the controls, the user constructs a specific function call that the user can execute or save into a program. Thus, the instrument driver function panel gives users the following two capabilities:

- Interactive control of the instrument
- The ability to generate function calls that can be included in an application program

In summary, the instrument driver provides functions to perform high-level instrument-related tasks. By including the function calls in an application program, the user can control an instrument without having to learn the programming protocol of the instrument.

Instrument Driver Architecture

To define a standard for instrument driver software design and development, one needs conceptual models around which to write the design specifications. This manual 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 components in the system. This model gives insight into key architectural decisions with regard to instrument drivers and adds context as to how instrument drivers are used.

The second model, called the instrument driver *internal* design model, defines how an instrument driver software module is organized internally. This model shows the consistency of approach to instrument driver design regardless of the type of instrument.

Instrument Driver External Interface Model

An 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. Therefore, the first step in creating a standard for instrument drivers is to define a model to explain how the instrument driver interacts with the rest of the system.

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

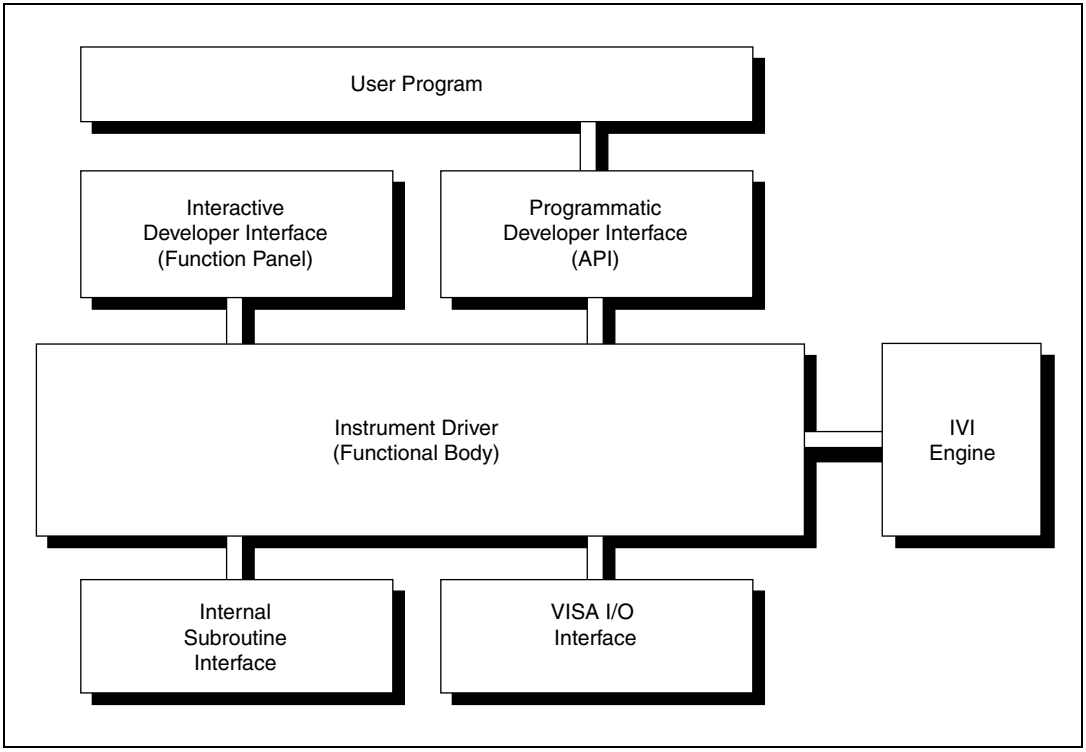

Figure 1-1. Instrument Driver External Interface Model

This general model contains the instrument driver *functional body*, which is the source code of the instrument driver. The *programmatic 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 graphical interface that assists the software developer in understanding what each particular instrument driver function does and how to use the programmatic developer interface to call each function. The *IVI engine* monitors attribute states and performs state-caching. The *VISA I/O interface* is the mechanism through which the driver communicates with the instrument hardware. The *internal subroutine interface* is the mechanism through which the driver can call other software modules it might use to perform its task. These other software modules can include operating system calls or calls to other unique libraries such as formatting and analysis functions.

Non-IVI drivers have the same external interface model, but they do not use the IVI engine.

Functional Body

The functional body of a LabWindows/CVI instrument driver is a library of C functions for controlling a specific instrument. Because the functional body is developed with the standard tools provided in the LabWindows/CVI environment, users can easily view instrument driver source code and optimize it for their application. The details of the functional body are based on the instrument driver internal design model. Chapter 3, *Programming Guidelines for Instrument Drivers*, describes the guidelines for creating the instrument driver functional body.

Programmatic Developer Interface

The programmatic developer interface is the mechanism for using the instrument driver as part of a test program application. In the LabWindows/CVI instrument driver architecture, the software interface to an instrument driver is the same as for any other software library module that a user might want to develop or use. The programmatic developer interface is a standard software function call, with no special instrument-driver-specific requirements.

Interactive Developer Interface

When you use a LabWindows/CVI instrument driver as an integral part of a higher-level application software development environment, you can enhance the programmatic developer interface to the instrument driver with function panels, which are graphical representations of functions. Function panels, referred to as the *interactive developer interface*, help you learn how to use the instrument driver. Use the function panel interface to interactively execute an instrument driver function and to generate the instrument driver function calls into an application program.

IVI Engine

IVI is the name of an instrument driver architecture, a component engine that makes the architecture work, and a library that is an API to the engine. The IVI engine performs state-caching and tracks attributes. Chapter 2, *IVI Architecture Overview*, describes the IVI engine and its operation in detail.

VISA I/O Interface

An important consideration for instrument drivers is how they perform instrument I/O. In the LabWindows/CVI instrument driver architecture, the I/O interface is provided by a separate layer of software that is standard and available on numerous platforms. The VISA I/O interface is NI's next-generation I/O architecture. VISA includes a single interface library for controlling GPIB, VXI, RS-232, PXI, and other types of instruments.

VISA is controller independent and can communicate with instruments via GPIB, MXI, embedded VXI, and GPIB-VXI controllers.

For interfaces that VISA does not support, you can use another I/O library.

Subroutine Interface

Because LabWindows/CVI instrument drivers are written in standard ANSI C, the subroutine interface is simply a function call. Therefore, an instrument driver is a software program that can do anything any other program can do. Some specific instrument drivers do nothing more than perform simple message-based and register-based I/O to and from an instrument, but others might control multiple instruments or use support libraries to integrate data analysis or other specialized capabilities inside the driver. You can use this type of approach to build virtual instruments that combine hardware and software capabilities. You can develop and package complete high-level tests as instrument drivers that other test developers can use.

The concept of virtual instrumentation is very important, and instrument driver tools must allow users to take advantage of it. The LabWindows/CVI instrument driver standard defined in this document applies to instrument drivers that control only a single instrument and to instrument drivers that combine features of multiple instruments and additional software processing. For this reason, the LabWindows/CVI instrument driver standard has unlimited potential as a mechanism for delivering baseline instrument drivers. It also has unlimited potential as a standard vehicle for delivering much more sophisticated application-specific capability targeted at highly vertical markets or particular application areas.

The subroutine interface is often used to call instrument driver support functions. You can define these functions within the LabWindows/CVI instrument driver source file, or you can supply them in an external module. End users cannot call instrument driver support functions.

Instrument Driver Internal Design Model

The instrument driver internal design model, shown in Figure 1-2, defines the internal organization of the functional body of the driver.

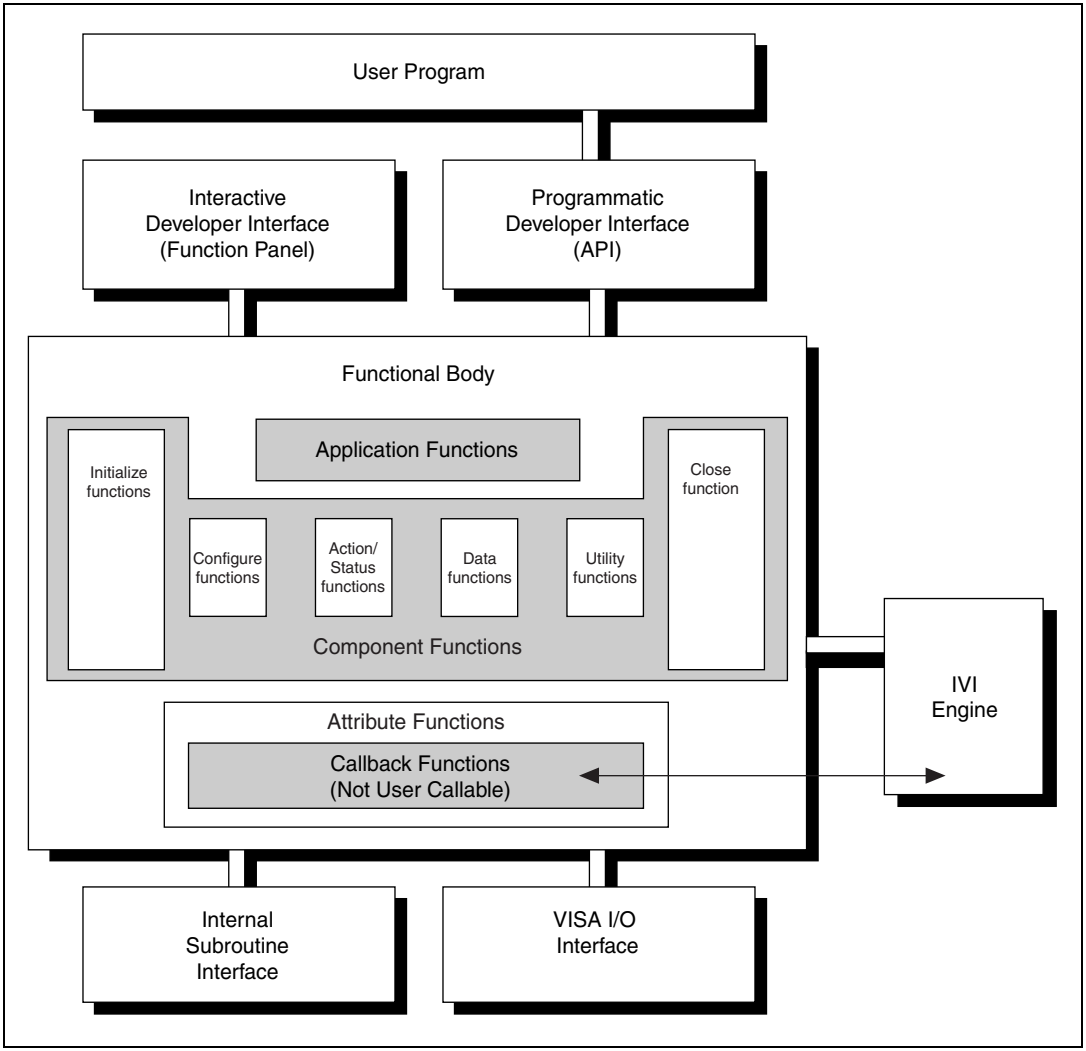

Figure 1-2. Instrument Driver Internal Design Mode

The functional body of a LabWindows/CVI instrument driver consists of three main categories. The first category is a collection of component functions, each of which controls a specific area of the instrument's functionality. The second category is a collection of application functions that invoke combinations of component functions to perform complete test and measurement operations. The third category is a collection of attribute functions and callback functions that perform such operations as reading and writing instrument settings or querying the state of an instrument. Non-IVI drivers do not have attribute functions or callback functions.

The modularity of LabWindows/CVI instrument drivers builds on proven technology. The modular approach gives users the granularity to control instruments properly in their application programs. For example, a user can initialize all instruments once, configure multiple instruments, and then trigger several instruments simultaneously. A user also can initialize and configure an instrument once, and then trigger and read from the instrument several times.

Application Functions

Application functions are high-level test and measurement oriented routines that call component instrument driver functions. Application functions commonly call a sequence of component functions to perform one high-level operation. For example, a DMM application function might configure the DMM and take a reading, all in one function call. Application functions aid users in understanding how component functions are combined to take instrument measurements.

Note Instrument driver application-level functions do not call the `Initialize` or `Close` functions.

Component Functions

LabWindows/CVI instrument drivers have component functions, which are typically divided into seven categories:

- Initialization
- Configuration
- Action/status
- Measurement
- Utility
- Attribute
- Close

These categories might differ for different types of instruments. For example, switches typically do not have a measurement category; switches have a route category instead. Each of these categories, with the exception of the initialize and close functions, consists of several modular software routines. Much of the critical work in developing an instrument driver lies in the initial design and organization of the instrument driver component functions. Some of the specific routines in each category are further categorized as *required functions*.

The required functions are instrument driver functions that are required by the IVI Foundation or are required for use with the IVI engine. The required functions are common to the majority of instruments. These functions perform the following instrument operations:

- Initialize and Initialize With Options
- Close
- Reset
- Self-Test
- Revision Query
- Error Query
- Error Message
- Get/Clear Error Info
- Get Next Coercion Record
- Get/Set Attribute
- Invalidate All Attributes
- Lock/Unlock Session
- Write/Read Instrument Data (message-based instruments)

The instrument driver developer specifies the remainder of the functions in the instrument driver. For example, all instruments have configuration functions, but some instruments can have multiple configuration functions for different types of configuration. General guidelines in Chapter 3, *Programming Guidelines for Instrument Drivers*, define, organize, and structure the functions within each category. Because of these guidelines, similar instruments have similar sets of functions.

The LabWindows/CVI instrument driver guidelines recommend that an instrument driver provide full functional control of the instrument. LabWindows/CVI does not 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.

The IVI Foundation class specifications define the external interface and functionality of all instrument drivers for a particular instrument type. The IVI Foundation defines some functionality as required and some as optional. The IVI Foundation also allows instrument drivers to define additional functions specific to a particular instrument. You can create an IVI driver without complying with a class specification, and class specifications do not exist for

all instrument types. Nevertheless, you gain the following benefits if you create an instrument driver according to a class specification:

- The class specification makes most of the design decisions for you.
- LabWindows/CVI has class templates that comply with the class specifications. You can use the class templates with the **Tools»Create IVI Instrument Driver** command. The command invokes a wizard that generates the skeleton `.c`, `.h`, `.fp`, and `.sub` files for an instrument driver. When you use the command with a class template, it adds to the instrument driver files all the attributes and high-level functions that the class specification defines. Refer to Chapter 3, *Programming Guidelines for Instrument Drivers*, for more information.
- By following a class specification, your instrument driver has an external interface that is very familiar to users who have used other instrument drivers that comply with the class definition.
- Specific drivers that comply with a class specification can be called from the IVI class drivers to create hardware-independent test programs.

Initialization Functions

The initialization functions establish the software connection to the instrument. The initialization functions can optionally perform an instrument identification query and reset operation. They perform any necessary actions to place the instrument in its default power-on state or other specific state. An extended initialization function allows users to configure certain IVI attributes at initialization time.

Configuration Functions

The configuration functions are a collection of software routines that configure the instrument to perform a particular operation. Configuration functions change the state of instrument settings. Numerous configuration functions can exist, depending on the particular instrument. The configuration functions group multiple calls to attribute functions and handle any order dependencies that might exist in setting attributes.

Action/Status Functions

The action/status category contains two types of functions: action and status. Action functions, such as `Initiate` and `Abort`, cause the instrument to initiate or terminate test and measurement operations. These operations can include arming the triggering system or generating a stimulus. These functions are different from the configuration functions because they do not change the instrument settings; instead, these functions order the instrument to carry out an action based on its current configuration. Status functions obtain the current status of the instrument or the status of pending operations. The specific routines in this category and the actual operations performed by those routines are left up to the instrument driver developer.

Measurement Functions

Measurement functions capture single-point and multi-point measurement data. Functions that initiate and transfer data to and from the instrument appear within the Measurement category. If a driver has a Measurements category, low-level action functions do not appear under a separate Action/Status category. Instead, low-level action and measurement functions, such as `Initiate`, `Fetch`, `Abort`, and `Send Software Trigger`, appear in a Low-Level Measurements sub-category below the Measurements category. If the instrument operation includes more than scalar measurements, this category might be more appropriately named. For example, oscilloscopes define a Waveform Acquisition category that includes both waveform and measurement functions. The instrument driver developer determines the specific routines in the Measurements category and the actual operations performed by those routines.

Utility Functions

Utility functions perform operations that are auxiliary to the operation of the instrument. These utility functions include functions required by the IVI Foundation, such as `Reset`, `Self-Test`, `Error Query`, and `Revision Query`. Other utility functions include functions for reading/writing to the instrument and calibration functions. Table 1-1, *Common Utility Functions*, shows examples of utility functions.

Table 1-1. Common Utility Functions

Function Name	Description
`Reset`	This function places the instrument in a default state.
`Revision Query`	This function returns the revision of the instrument driver and the firmware revision of the instrument.
`Error Query`	This function queries the status of the instrument and returns the instrument-specific error information.
`Error Message`	This function translates the error return value from an instrument driver function to a user-readable string.
`Get/Clear Error Info`	These functions allow you to get and clear the extra error information that IVI drivers provide.
`Get Next Coercion Record`	You can configure the IVI engine to track how the instrument driver coerces user settings. This function gets the next coercion record from the IVI engine.

Table 1-1. Common Utility Functions (Continued)

Function Name	Description
`Lock/Unlock Session`	These functions allow you to protect a sequence of calls to an IVI instrument driver from interference by other execution threads.
`Write/Read Instrument Data`	These instrument data functions allow the end user to perform instrument I/O directly.

Close Function

All LabWindows/CVI instrument drivers have a close function that terminates the software connection to the instrument and deallocates system resources.

Attribute Functions

The attribute functions set or query the value of particular instrument settings or attributes. Attribute functions use the IVI engine to manage instrument attribute values and states properly. The attribute functions provide low-level access to the individual instrument settings. Users normally call the configuration functions rather than the attribute functions. Refer to Chapter 2, *IVI Architecture Overview*, for details on the attribute functions. Non-IVI drivers do not have attribute functions.

Callback Functions

In IVI drivers, callback functions contain the source code for querying and modifying instrument settings, checking the status of the instrument, and other operations. The IVI engine invokes the callback functions at appropriate times.

There are two types of callback functions: *attribute* callbacks and *session* callbacks. Attribute callbacks, such as the read and write callbacks, apply to particular instrument settings or software attributes. Session callbacks apply to the instrument as a whole.

Non-IVI drivers do not have callback functions.

Refer to Chapter 2, *IVI Architecture Overview*, for detailed information on callback functions.

Achieving Interchangeability

Using the ANSI C IVI architecture, interchangeability is achieved through the class driver. IVI class drivers export functions defined in an IVI Foundation class specification. Test programs call the class-driver functions rather than the specific-driver functions. The class driver dynamically loads an IVI-specific driver at run time and serves as a pass-through layer

to the specific driver. The class driver can act as a pass through layer only if the specific driver also exports functions defined by the IVI Foundation class specification.

To allow users to swap instruments without recompiling or re-linking, the application program must identify and load the specific driver at run-time without directly referencing the driver in the program. To achieve interchangeability, users pass a logical name to the `Initialize` function of the class driver. The logical name matches a logical name in the IVI configuration file. In the IVI configuration store file, the logical name refers to an IVI-specific driver and a specific physical instrument. The configuration file provides the class driver with the necessary path information for loading the specific driver.

Users access the IVI configuration file by using the IVI configuration utility available in Measurement and Automation Explorer (MAX).

Figure 1-3 shows how users achieve interchangeability when using a class driver together with a class-compliant specific driver.

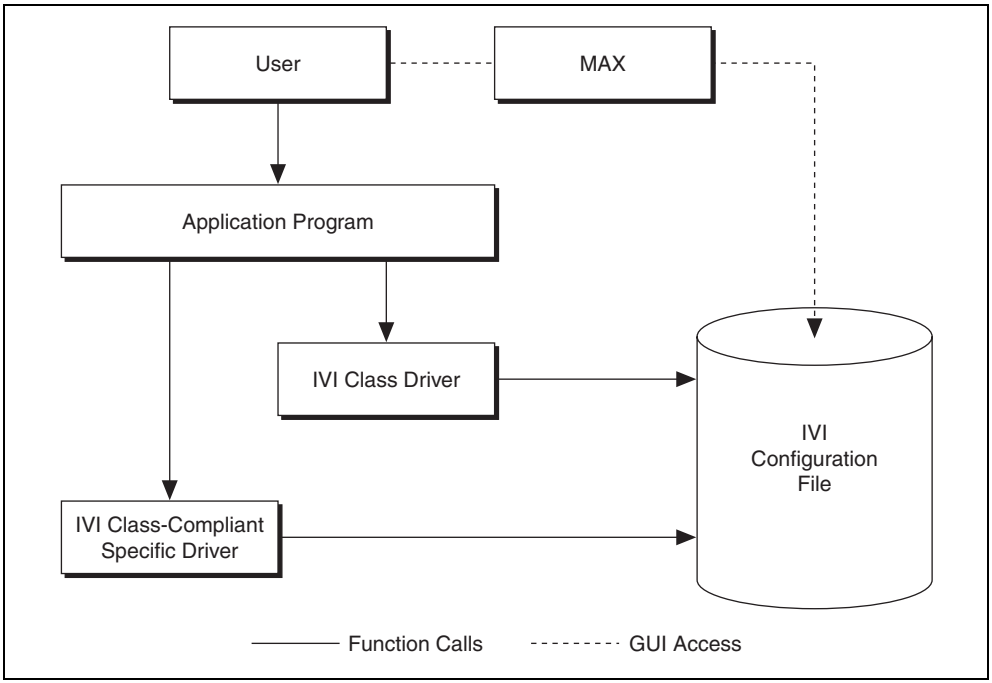

Figure 1-3. Achieving Interchangeability with IVI

2

IVI Architecture Overview

This chapter contains a general overview of the concepts of IVI instrument drivers and the IVI engine. It also contains detailed descriptions of the inherent IVI attributes.

What is IVI?

IVI drivers comply with the IVI Foundation specifications. Throughout this manual, the term IVI is used to refer to NI's implementation of IVI instrument drivers, an architecture for these drivers, a component engine that makes the architecture work, and a library that is an API to the engine. Following are the major features of the IVI driver architecture from NI.

- A standard, well-defined structure for the external interface to an instrument driver that leverages VXI*plug&play* standards.
- A standard, well-defined structure for the internal implementation of an instrument driver.
- An attribute model for representing the settings of an instrument.
- A standard set of callback functions the instrument driver can define and install to implement an instrument attribute.
- An optional state-caching mechanism that tracks the state of instrument settings to prevent unnecessary instrument I/O, thereby increasing the performance of application programs.
- A standard interface for enabling and disabling the validation of parameters the user passes to instrument driver functions.
- A standard interface for enabling and disabling queries of the instrument's status after an operation.
- A standard interface for using an instrument driver in simulation mode.
- The ability to safely access the same instrument driver session from multiple execution threads in one application.
- The ability of an application program thread to lock an instrument driver session so that a thread can execute a critical section of code without other threads in the same program interfering with the state of the instrument.
- The ability of an instrument driver to report extensive error information.

- The definition of standard classes for common types of instruments. A standard class definition specifies the high-level functions, attributes, and attribute values common to a wide variety of instruments of the same type. Instrument drivers that comply with a standard class definition provide users with a familiar interface from one specific instrument to another of the same type.
- Class drivers that work on all instruments of the same type.
- LabWindows/CVI wizards that help you create and modify IVI instrument drivers. The wizards are particularly powerful when you use them with the templates for a standard instrument class specification.
- Multiple platform capability. The IVI engine is available for Windows 2000/NT/Me/9*x*. On Windows 2000/NT/Me/9*x*, the IVI engine is in the form of a 32-bit DLL. The IVI engine requires that you install VISA, but it is not dependent on any other library.

Introduction to IVI Instrument Drivers

This section contains a brief introduction to IVI instrument drivers, how they work, and how you use them in application programs. This chapter also contains a detailed discussion of the IVI attribute model, callbacks, state-caching, and the responsibilities of high-level instrument driver functions. At the end of the chapter, you will find detailed descriptions of inherent IVI attributes. Refer to Chapter 3, *Programming Guidelines for Instrument Drivers*, and Chapter 4, *Attribute Editor*, for information on the LabWindows/CVI wizards that help you create and modify IVI instrument drivers. Refer to Chapter 3, *Programming Guidelines for Instrument Drivers*, for detailed information on instrument driver source code. Refer to the *LabWindows/CVI Help* for descriptions of the functions in the IVI engine API and for information about the IVI error reporting capabilities.

Instrument drivers are high-level function libraries for controlling specific GPIB, VXI, serial instruments, or other devices. With an instrument driver, you easily can control an instrument without knowing the low-level command syntax or I/O protocol. IVI instrument drivers apply an attribute-based approach to instrument control to deliver better run-time performance and more flexible instrument driver operation.

An IVI instrument driver specifies each readable or writable setting on your instrument, such as the vertical voltage range on an oscilloscope, as an attribute. The IVI engine works in conjunction with IVI instrument drivers to manage the reading and writing of instrument attributes. The IVI engine tracks the values of attributes in memory and controls when instrument drivers send new settings to instruments and read settings from instruments. By managing instrument attributes and mandating a standard structure for the internal

implementation of instrument drivers, the IVI engine adds many features to instrument drivers, including the following:

- **State-caching**—You can avoid sending redundant commands to the instrument as the IVI engine skips duplicate attribute value settings.
- **Configurable range-checking**—Range-checking verifies that a value you specify for an attribute is within the valid range for the attribute. You can disable this feature for faster execution speed.
- **Configurable status query**—The status query feature automatically checks the status register of the instrument after each operation. You can disable this feature for faster execution speed.
- **Simulation**—You can develop application code that uses an instrument driver even when the instrument is not available. When in simulation mode, the instrument driver range-checks input parameters and generates simulated data for output parameters.

Enable or disable these features by setting special attribute values in the instrument driver. For example, you can set the `FL45_ATTR_RANGE_CHECK` or `FL45_ATTR_SIMULATE` attributes of the Fluke 45 digital multimeter driver to `VI_TRUE` to enable range-checking or simulation. These types of attributes are called *inherent* IVI attributes because every IVI instrument driver has them and because these attributes control how the instrument driver works rather than representing particular instrument settings.

How IVI Instrument Drivers Work

This section presents a simplified view of how the IVI engine and driver work together. The actual process involves additional steps that implement other powerful features. The rest of this chapter discusses the entire process in detail.

The key to the IVI architecture is the manner in which the IVI engine controls the reading and writing of attributes to and from the instrument. The instrument driver contains *callback functions* that read and write instrument settings and *range tables* that specify the valid range for each instrument attribute. The IVI engine accesses the range tables and invokes the callbacks at the appropriate times. For example, suppose the instrument driver exports a `Scope_ConfigureChannel` function that configures the vertical subsystem of an oscilloscope. Suppose that you pass 5.0 for the vertical range parameter to the function. The driver and the IVI engine work together to execute the following steps:

1. The `Scope_ConfigureChannel` function calls the `Ivi_SetAttributeViReal64` function in the IVI engine to set the value of the vertical range to 5.0 volts.
2. If the `IVI_ATTR_RANGE_CHECK` inherent attribute is `VI_TRUE`, the IVI engine uses the range table for the vertical range attribute to determine if 5.0 volts is a valid value. If 5.0 is outside the valid range for your particular oscilloscope, the `Ivi_SetAttributeViReal64` function returns an error code. In some cases,

the IVI engine uses the range table to coerce the value you request to a value that is valid for the instrument, regardless of whether you enable range-checking.

3. If the `IVI_ATTR_CACHE` inherent attribute is `VI_TRUE`, the IVI engine compares the vertical range value you are requesting, 5.0, to the value currently in the engine's cache for the attribute. If the cache value equals 5.0, `Ivi_SetAttributeViReal64` returns the success completion code immediately. Because the vertical range is already set to 5.0 volts, sending a command to the instrument to set the vertical range to 5.0 volts is redundant.
4. If the `IVI_ATTR_SIMULATE` inherent attribute is `VI_TRUE`, `Ivi_SetAttributeViReal64` returns the success completion code immediately.
5. The IVI engine invokes the `Scope_VerticalRangeWriteCallback` function in the instrument driver. The write callback function sends a command string to the oscilloscope to set the vertical range to 5.0. The IVI engine updates the cache value for the vertical range attribute to 5.0.
6. If the `IVI_ATTR_QUERY_INSTR_STATUS` inherent attribute is `VI_TRUE` and the IVI engine invoked `Scope_VerticalRangeWriteCallback` or the write callback for any other attribute, `Scope_ConfigureChannel` calls the check status callback in the instrument driver. The check status callback reads the status register of the oscilloscope to check if an error condition occurred.

Notice that steps 1 through 5 repeat for each parameter you pass to `Scope_ConfigureChannel`. Figure 2-1 illustrates this process.

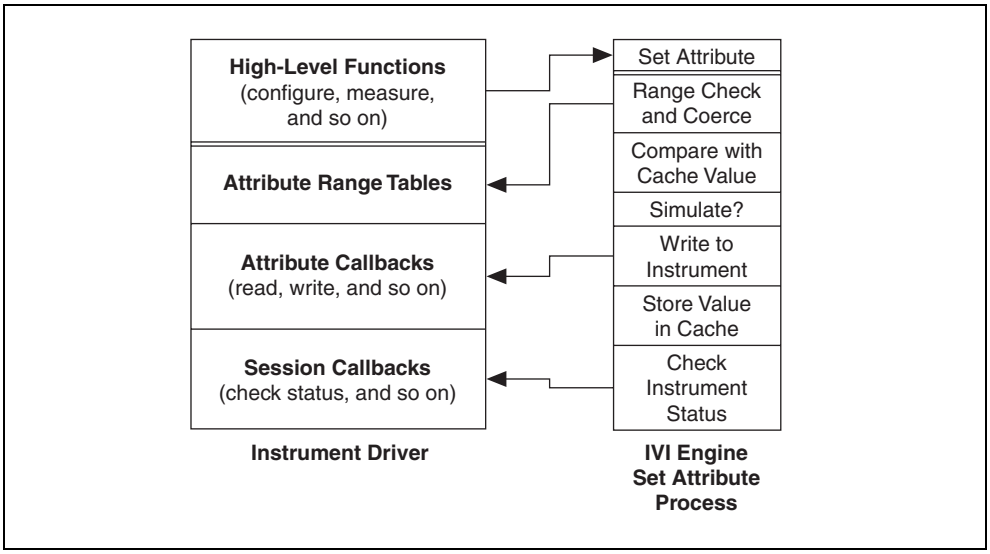

Figure 2-1. IVI Driver Operation Diagram

Programming with IVI Instrument Drivers

Each IVI instrument driver presents a set of high-level functions for controlling an instrument. The high-level functions are sufficient for most applications. Although each IVI instrument driver exports `SetAttribute` and `GetAttribute` functions, you typically do not use them in application programs. When you call the configuration functions in the instrument driver, you specify configuration settings. The configuration functions pass these settings to `SetAttribute` function calls. In many cases, it is necessary to send settings to the instrument in a particular order. The high-level configuration functions handle these order dependencies for you.

To use an instrument driver for a particular instrument, you must first initialize an IVI instrument driver *session*. When you initialize an IVI session, the IVI engine creates data structures to store all the information for the session. Each IVI instrument driver exports two functions for initializing an IVI session: *`Prefix`*`_init` and *`Prefix`*`_InitWithOptions`, where *`Prefix`* is the instrument prefix for the driver. Both functions require that you identify the physical device. The functions give you the option of performing ID query and reset operations on the device. The functions also initialize the device for correct operation of subsequent instrument driver function calls.

The *`Prefix`*`_InitWithOptions` function has a string parameter in which you can set the value of inherent attributes such as `IVI_ATTR_RANGE_CHECK`, `IVI_ATTR_SIMULATE`, and `IVI_ATTR_QUERY_INSTR_STATUS`. Set these attributes through the *`Prefix`*`_InitWithOptions` function so that your settings are in effect during the initialization procedure. It is particularly important to use the *`Prefix`*`_InitWithOptions` function if you want to enable simulation. The initialization procedure usually attempts to interact with the physical instrument. If the instrument is not available, you must enable simulation in your call to *`Prefix`*`_InitWithOptions` to prevent the initialization from failing.

The *`Prefix`*`_init` and *`Prefix`*`_InitWithOptions` functions return an IVI session handle that you pass to subsequent calls to instrument driver functions. If you want to use the same instrument driver for a separate physical device, you must call *`Prefix`*`_init` or *`Prefix`*`_InitWithOptions` again to initialize a different IVI session.

Do not open multiple IVI sessions to the same physical device because two separate cached states may not be identical and one may be out of sync with the instrument. If you want to use the same instrument in different execution threads, you can do so by sharing the same session handle across multiple threads in the same program. IVI instrument drivers functions are multithread safe. In some cases, however, you might have to lock a session around a sequence of calls to an IVI instrument driver. For example, if you call a configuration function and then take a reading in one thread, you must prevent calls to instrument driver functions in other threads from altering the configuration between the two function calls in the first thread. Each IVI instrument driver exports the *`Prefix`*`_LockSession` and *`Prefix`*`_UnlockSession` functions for this purpose.

Driver Functions and Attribute Model

The IVI Foundation requires that each IVI-C instrument driver contain the following additional inherent functions:

*Prefix*_InitWithOptions
*Prefix*_SetAttribute<type>
*Prefix*_GetAttribute<type>
*Prefix*_GetNextCoercionRecord
*Prefix*_LockSession
*Prefix*_UnlockSession
*Prefix*_GetErrorInfo
*Prefix*_ClearErrorInfo

IVI drivers that use the IVI engine require that each IVI driver contain the following functions:

*Prefix*_IviInit
*Prefix*_IviClose

The following functions are recommended to implement in IVI drivers:

*Prefix*_GetNextInterchangeWarning
*Prefix*_CheckAttribute<type>
*Prefix*_ReadInstrData (Message-based instruments only)
*Prefix*_WriteInstrData (Message-based instruments only)

Refer to the *LabWindows/CVI Help* for a description of all functions that VXI*plug&play* and IVI require.

VXI*plug&play* drivers also contain high-level functions that encapsulate multiple instrument interactions. The drivers usually group high-level functions into categories such as configuration functions and measurement functions. A configuration function modifies one or more settings on the instrument. A measurement function usually sends a command to the instrument requesting data and then reads the data from the instrument. Drivers often contain application-level functions, which are very high-level functions that typically call one or more configuration functions and a measurement function.

IVI drivers contain the same types of high-level functions. A key difference between IVI and non-IVI drivers is how they implement the high-level functions. Non-IVI drivers use direct instrument I/O to query and modify instrument settings. IVI drivers query and modify instrument settings through attributes. Thus, a high-level function in an IVI driver might consist of a set of calls to IVI Library functions such as `Ivi_GetAttributeViInt32`, `Ivi_SetAttributeViInt32`, and `Ivi_SetAttributeViReal64`. The IVI engine provides the mechanism for the management of instrument driver attributes.

Types of Attributes

Some attributes are common to all IVI instrument drivers. These attributes are called *inherent attributes*. The specification for an instrument class defines attributes that are common among all instruments of one type. These attributes are called *class-defined* attributes. A specific instrument driver defines attributes that are specific to a particular instrument model or family of models. These attributes are called *instrument-specific* attributes. Each specific instrument driver API exports the inherent IVI attributes and most or all of its instrument-specific attributes. A specific instrument driver that complies with a standard class definition also exports the class-defined attributes.

An instrument driver can use attributes for more than modeling instrument states. For instance, the driver can use attributes to represent values that the driver recalculates dynamically each time the user queries its value. The driver also can use attributes to maintain internal data it wants to attach to each IVI session the user creates. Instrument drivers can choose whether to export such attributes to the user. Attributes that drivers export to users are called *public* attributes. Attributes that drivers use only internally are called *private* or *hidden* attributes.

The instrument driver must assign one of the following VISA data types to each attribute: `ViInt32`, `ViReal64`, `ViString`, `ViBoolean`, `ViSession`, or `ViAddr`. Only hidden attributes can be of type `ViAddr`.

Refer to the *Inherent IVI Attributes* section in this chapter for detailed descriptions of each inherent attribute.

Get/Set/Check Functions

When a high-level function in an instrument driver queries or modifies the current setting of an attribute, it does so by calling one of the `Ivi_GetAttribute` or `Ivi_SetAttribute` functions. The IVI Library contains six `Ivi_GetAttribute` functions and six `Ivi_SetAttribute` functions, one for each possible attribute data type. These are called *typesafe* functions.

The IVI engine also exports six typesafe `Ivi_CheckAttribute` functions. Instrument drivers can call these functions to verify that a particular value is valid for an attribute.

Instrument drivers export *`Prefix`*`_GetAttribute`, *`Prefix`*`_SetAttribute`, and *`Prefix`*`_CheckAttribute` functions for each of the five data types that instrument driver public attributes can have. Exporting these functions for each data type allows users to bypass the high-level functions in instrument drivers and directly query and modify the values of instrument attributes. The *`Prefix`*`_GetAttribute`, *`Prefix`*`_SetAttribute`, and *`Prefix`*`_CheckAttribute` functions are merely wrappers around calls to the `Ivi_GetAttribute`, `Ivi_SetAttribute`, and `Ivi_CheckAttribute` functions.

Each `Ivi_Get/Set/CheckAttribute` function has an **optionFlags** parameter. The *`Prefix`*`_Get/Set/CheckAttribute` functions that instrument drivers export do not have this parameter. They always pass the `IVI_VAL_DIRECT_USER_CALL` flag to the `Ivi_Get/Set/CheckAttribute` function. For more information on the **optionFlags** parameter, refer to the function descriptions for the `Ivi_Get/Set/CheckAttribute` functions in the *LabWindows/CVI Help*.

The *LabWindows/CVI Help* contains one consolidated function description for all the `Ivi_GetAttribute` functions, except `Ivi_GetAttributeViString`, one consolidated function description for all the `Ivi_SetAttribute` functions, and one consolidated function description for all the `Ivi_CheckAttribute` functions. The function descriptions contain detailed information on the purpose of each function.

Callbacks

The IVI engine contains a sophisticated attribute state-caching mechanism. Each specific instrument driver, on the other hand, can obtain settings from the instrument, validate new settings, and send new settings to the instrument. Thus, a function call to set or get an attribute value executes code, both in the IVI engine and in the specific instrument driver.

The most efficient way to partition this work is through a callback mechanism. The specific instrument driver can query and modify instrument attribute values into a set of callback functions for each attribute. The driver installs the callbacks for each attribute by passing the addresses of the callback functions to the IVI engine. Besides enabling state-caching, this scheme also allows the IVI engine to play an important role in implementing the range-checking, status-checking, and simulation options in instrument drivers.

All function calls to get or set an attribute value first go through the IVI engine. The IVI engine uses this opportunity to determine whether state-caching, range-checking, and simulation are enabled. It also determines whether the cached value of the attribute is valid, that is, whether the attribute reflects the current state of the instrument. Depending on these factors, the IVI engine invokes one or more callback functions. After the callbacks return, the IVI engine can take additional actions such as storing a new value in the cache.

The IVI engine allows an instrument driver to install six callback functions for each attribute: read, write, check, coerce, compare, and range table. Refer to the *Attribute Callback Functions* section in this chapter for a description of the six attribute callbacks.

The IVI engine also allows an instrument driver to install two callback functions that are global to an entire IVI session: an operation complete callback and a check status callback. Refer to the *Session Callback Functions* section in this chapter for a description of the session callbacks.

The driver can choose which callbacks to install. The IVI engine does not require any of the callbacks.

Creating and Declaring Attributes

An instrument driver creates the specific and class attributes it uses by calling the `Ivi_AddAttribute` functions. The IVI Library provides a separate `Ivi_AddAttribute` function for each of the six data types, for example, `Ivi_AddAttributeViInt32` and `Ivi_AddAttributeViBoolean`. In the include file for the driver, the driver must declare constant names for all the public attributes. In the source file for the driver, the driver must declare constant names for all the hidden attributes.

The IVI engine creates the inherent IVI attributes for each session. The include file for the driver, however, must define constant names for all the attributes that are not hidden from the user. In this way, the driver include file presents a unified view of all attributes that the user can use in conjunction with the instrument driver.

Attribute IDs

Each attribute in an instrument driver must have a distinct integer ID. You must define a constant name for each attribute in the include file or the source code for the instrument driver. The constant name must begin with *`PREFIX`*`_ATTR_`, where *`PREFIX`* is the instrument prefix.

The include file for a specific instrument driver must define constant names for all the public attributes including attributes that the IVI Library defines, attributes that the instrument class defines, and attributes that are specific to the particular instrument.

Note The **Tools»Create IVI Instrument Driver** and **Tools»Edit Instrument Attributes** commands create the correct attribute constant definitions for you.

Inherent IVI Attributes

For each inherent IVI attribute, use the same constant name that appears in `ivi.h` but replace the IVI prefix with the specific instrument prefix. For example, `ivi.h` defines `IVI_ATTR_CACHE`, and the Fluke 45 include file, `fl45.h`, contains the following definition:

```
#define FL45_ATTR_CACHE          IVI_ATTR_CACHE
```

Class Attributes

For each class attribute, use the same constant name that appears in the class include file but replace the class prefix with the specific instrument prefix. For example, the DMM class include file, `ividmm.h`, defines `IVIDMM_ATTR_RESOLUTION`, and `fl45.h` contains the following definition:

```
#define FL45_ATTR_RESOLUTION    IVIDMM_ATTR_RESOLUTION
```

Instrument-Specific Attributes

For each instrument-specific attribute that the user can access, define a constant name in the instrument driver include file and assign a value that is an offset from `IVI_SPECIFIC_PUBLIC_ATTR_BASE`. For example, `fl45.h` contains the following definition:

```
#define FL45_ATTR_HOLD_THRESHOLD    \
             (IVI_SPECIFIC_PUBLIC_ATTR_BASE + 3)
```

For each instrument-specific attribute that is hidden from the user, define a constant name in the driver source file and assign a value that is an offset from `IVI_SPECIFIC_PRIVATE_ATTR_BASE`. For example, `hp34401.c` contains the following definition:

```
#define HP34401_ATTR_TRIGGER_TYPE    \
              (IVI_SPECIFIC_PRIVATE_ATTR_BASE + 1)
```

Attribute Flags

Each attribute has a set of flags that you can use to specify various types of behavior. You set the flags as bits in a `ViInt32` value you specify when you create the attribute using one of the `Ivi_AddAttribute` functions. You can query and modify the flags for an attribute using `Ivi_GetAttributeFlags` and `Ivi_SetAttributeFlags`.

To set multiple flags, bitwise-OR them together. For example, if you want an attribute to be read-only and never cached, use the following flags:

```
IVI_VAL_NOT_USER_WRITABLE|IVI_VAL_NEVER_CACHE
```

Table 2-1 lists the IVI attribute flags. A detailed discussion of each flag follows the table.

Table 2-1. IVI Attribute Flags

Bit	Value	Flag
0	0x0001	IVI_VAL_NOT_SUPPORTED
1	0x0002	IVI_VAL_NOT_READABLE
2	0x0004	IVI_VAL_NOT_WRITABLE
3	0x0008	IVI_VAL_NOT_USER_READABLE
4	0x0010	IVI_VAL_NOT_USER_WRITABLE
5	0x0020	IVI_VAL_NEVER_CACHE
6	0x0040	IVI_VAL_ALWAYS_CACHE
9	0x0200	IVI_VAL_FLUSH_ON_WRITE

Table 2-1. IVI Attribute Flags (Continued)

Bit	Value	Flag
10	0x0400	IVI_VAL_MULTI_CHANNEL
11	0x0800	IVI_VAL_COERCEABLE_ONLY_BY_INSTR
12	0x1000	IVI_VAL_WAIT_FOR_OPC_BEFORE_READS
13	0x2000	IVI_VAL_WAIT_FOR_OPC_AFTER_WRITES
14	0x4000	IVI_VAL_USE_CALLBACKS_FOR_SIMULATION
15	0x8000	IVI_VAL_DONT_CHECK_STATUS

IVI_VAL_HIDDEN is 0x0018, the combination of IVI_VAL_NOT_USER_READABLE and IVI_VAL_NOT_USER_WRITABLE. Use the IVI_VAL_HIDDEN flag when you create attributes that you do not want the user to access.

IVI_VAL_NOT_SUPPORTED—Indicates that the class driver defines the attribute, but the specific driver does not implement it.

IVI_VAL_NOT_READABLE—Indicates that neither users nor instrument drivers can query the value of the attribute. Only the IVI engine can query the value of the attribute.

IVI_VAL_NOT_WRITABLE—Indicates that neither users nor instrument drivers can modify the value of the attribute. Only the IVI engine can modify the value of the attribute.

IVI_VAL_NOT_USER_READABLE—Indicates that users cannot query the value of the attribute. Only the IVI engine and instrument drivers can query the value of the attribute.

IVI_VAL_NOT_USER_WRITABLE—Indicates that users cannot modify the value of the attribute. Only the IVI engine and instrument drivers can modify the value of the attribute.

IVI_VAL_NEVER_CACHE—Directs the IVI engine never to use the cache value of the attribute, regardless of the state of the IVI_ATTR_CACHE attribute. The IVI engine always calls the read and write callbacks for the attribute, if present.

IVI_VAL_ALWAYS_CACHE—Directs the IVI engine to use the cache value of the attribute, if it is valid, regardless of the state of the IVI_ATTR_CACHE attribute.

IVI_VAL_MULTI_CHANNEL—Indicates that the attribute has a separate value for each channel. You cannot modify this flag using Ivi_SetAttributeFlags.

IVI_VAL_COERCEABLE_ONLY_BY_INSTR—Indicates that the instrument coerces values in a way that the instrument driver cannot anticipate in software. Do *not* use this flag unless the instrument's coercion algorithm is undocumented or too complicated to encapsulate in a

range table or a coerce callback. When you query the value of an attribute for which this flag is set, the IVI engine ignores the cache value unless it obtained the cache value from the instrument. Thus, after you call an `Ivi_SetAttribute` function, the IVI engine invokes the read callback the next time you call an `Ivi_GetAttribute` function. When you set this flag, the IVI engine makes two assumptions that allow it to retain most of the benefits of state-caching:

- The instrument always coerces the same value in the same way.
- If you send the instrument a value that you obtained from the instrument, the instrument does not coerce the value.

Based on these two assumptions, the IVI engine does not invoke the write callback for the attribute when you call an `Ivi_SetAttribute` function with the same value you just sent to, or received from, the instrument. If one or both of these assumption are not valid, use the `IVI_VAL_NEVER_CACHE` flag instead.

`IVI_VAL_WAIT_FOR_OPC_BEFORE_READS`—Directs the IVI engine to call the operation complete callback for the session before calling the read callback for the attribute.

`IVI_VAL_WAIT_FOR_OPC_AFTER_WRITES`—Directs the IVI engine to call the operation complete callback for the session after calling the write callback for the attribute.

`IVI_VAL_USE_CALLBACKS_FOR_SIMULATION`—Directs the IVI engine to invoke the read and write callbacks for the attribute even when in simulation mode.

`IVI_VAL_DONT_CHECK_STATUS`—By default, when a user calls one of the *`Prefix`*`_GetAttribute` or *`Prefix`*`_SetAttribute` functions in an instrument driver and the `IVI_ATTR_QUERY_INSTR_STATUS` attribute is enabled, the IVI engine calls the check status callback for the session after calling the read or write callback for the attribute. This flag directs the IVI engine never to call the check status callback for the attribute.

Range Tables

For each `ViInt32` or `ViReal64` attribute, you can specify a range table that describes the valid values for the attribute. The IVI engine uses the table to validate and coerce values for the attribute. The write callback for the attribute also can use the table to associate each value with a command string to send to the instrument. Similarly, the read callback can use the table to convert a response string from the instrument into an attribute value.

Range Table Structures

The typedefs for IviRangeTable and IviRangeTableEntry in ivi.h describe structures you use to define range tables as follows:

```
typedef struct  /* describes one range table entry */
   {
      ViReal64    discreteOrMinValue;
      ViReal64    maxValue;
      ViReal64    coercedValue;
      ViString    cmdString;      /* optional */
      ViInt32     cmdValue;       /* optional */
   } IviRangeTableEntry;

typedef struct  /* describes the entire range table */
   {
      ViInt32     type;   /* discrete, ranged, or coerced */
      ViBoolean   hasMin;
      ViBoolean   hasMax;
      ViString    customInfo;
      IviRangeTableEntry  *rangeValues;
   } IviRangeTable;
```

The rangeValues field contains a pointer to an array of IviRangeTableEntry structures. The array must contain a termination entry, which is an entry in which the cmdString field contains IVI_RANGE_TABLE_END_STRING. The ivi.h include file defines IVI_RANGE_TABLE_END_STRING as ((ViString)(-1)). The ivi.h include file also defines the IVI_RANGE_TABLE_LAST_ENTRY macro, which you can use to represent an entire termination entry.

Three types of range tables exist. The type determines how you interpret the discreteOrMinValue, maxValue, and coercedValue fields in the IviRangeTableEntry structure. You indicate the type in the type field of the IviRangeTable structure. The three types are as follows:

- IVI_VAL_DISCRETE—In a discrete range table, each entry defines a discrete value. The discreteOrMinValue contains the discrete value. The maxValue and coercedValue fields are not used.
- IVI_VAL_RANGED—In a ranged range table, each entry defines a range with a minimum and a maximum value. The discreteOrMinValue field holds the minimum value. The maxValue field holds the maximum value. The coercedValue field is not used. If the attribute has only one continuous valid range and you do not assign different command strings or command values to subsets of the range, the range table contains only one entry other than the terminating entry.

- IVI_VAL_COERCED—In a coerced range table, each entry defines a discrete value that represents a range of values, which is useful when an instrument supports a set of ranges, and you must specify those ranges to the instrument using one discrete value. The discreteOrMinValue holds the minimum value of the range. The maxValue holds the maximum value. The coercedValue holds the discrete value that represents the range.

The discreteOrMinValue, maxValue, or coercedValue fields are always of type ViReal64, even for ViInt32 attributes.

The IVI Library exports functions that access entries in the range tables. Examples of these functions include Ivi_GetViInt32EntryFromValue and Ivi_GetViInt32EntryFromCmdValue. Refer to the *LabWindows/CVI Help* for more details on the use of these and other range table functions.

You can use the cmdString field to store the command string that the write callback uses to set the instrument to the value that the range table entry defines. The read callback also can use the cmdString field to convert a response string from the instrument into an attribute value. If you do not want to associate a command string with each value, you can set the cmdString field to VI_NULL.

For a register-based instrument, you can use the cmdValue field to store the register value. The write callback uses this register value to set the instrument to the value that the range table entry defines. For a message-based instrument, you can use the cmdValue field to store an integer value that the attribute write callback formats into an instrument command string. You can use the customInfo field to store the format string for the instrument command.

The hasMin and hasMax fields indicate whether, as a whole, the table contains a relevant minimum value and maximum value. The Ivi_GetAttrMinMaxViInt32 and Ivi_GetAttrMinMaxViReal64 functions use these fields to determine whether they can calculate the minimum and maximum values that the instrument implements for an attribute. For coerced range tables, these functions use the coercedValue field to calculate the minimum and maximum values that the instrument actually implements. In discrete range tables that contain values that represent non-numeric settings, assign VI_FALSE to the hasMin and hasMax fields. For example, the measurement function attribute of a DMM does not have a relevant minimum or maximum value.

If you use the **Tools»Edit Instrument Attributes** command in the source window, you can view and modify your range tables in a dialog box. The **Edit Instrument Attributes** command requires that the name of the array of IviRangeTableEntry structures always be the name of the IviRangeTable structure followed by Entries.

Discrete Range Table Example

The following is an example of a discrete range table.

```
static IviRangeTableEntry functionTableEntries[] = {
    /* discrete value */                /* cmdString */
    {FL45_VAL_DC_VOLTS,           0,  0, "VDC",   0},
    {FL45_VAL_AC_VOLTS,           0,  0, "VAC",   0},
    {FL45_VAL_AC_PLUS_DC_VOLTS,   0,  0, "VACDC", 0},
    {FL45_VAL_DC_CURRENT,         0,  0, "ADC",   0},
    {FL45_VAL_AC_CURRENT,         0,  0, "AAC",   0},
    {FL45_VAL_AC_PLUS_DC_CURRENT, 0,  0, "AACDC", 0},
    {FL45_VAL_2_WIRE_RES,         0,  0, "OHMS",  0},
    {FL45_VAL_FREQ,               0,  0, "FREQ",  0},
    {FL45_VAL_CONTINUITY,         0,  0, "CONT",  0},
    {FL45_VAL_DIODE,              0,  0, "DIODE", 0},
    {IVI_RANGE_TABLE_LAST_ENTRY}
};
static IviRangeTable  functionTable = {
    IVI_VAL_DISCRETE,  /* type */
    VI_FALSE,          /* hasMin */
    VI_FALSE,          /* hasMax */
    VI_NULL,           /* customInfo */
    functionTableEntries,
};
```

This range table lists all the possible values for a DMM measurement function attribute. It also lists the command strings the driver uses to set the instrument to each possible value and convert instrument response strings to attribute values.

Coerced Range Table Example

The following is an example of a coerced range table.

```
static IviRangeTableEntry resolutionTableEntries [] = {
    /* min   max  coerced cmdString       */
    {0.0, 4.5, 4.5,   "F",      0},
    {4.5, 5.5, 5.5,   "M",      0},
    {5.5, 6.5, 6.5,   "S",      0},
    {IVI_RANGE_TABLE_LAST_ENTRY}
};
static IviRangeTable  resolutionTable = {
    IVI_VAL_COERCED,   /* type */
    VI_TRUE,           /* hasMin */
    VI_TRUE,           /* hasMax */
```

```
    VI_NULL,            /* customInfo */
    resolutionTableEntries
};
```

This range table lists all the possible ranges for a DMM resolution attribute, in terms of digits of precision. For each range, the range table specifies a coerced value. In this case, the coerced value is the highest value in the range. The table also lists the command string the driver uses to set the instrument to each possible coerced value and convert instrument response strings to coerced values.

Ranged Range Table Example

The following is an example of a ranged range table.

```
static IviRangeTableEntry  triggerDelayTableEntries [] = {
     /* min        max    */
    {1.0e-6,  100.0,   0, VI_NULL,      0},
    {IVI_RANGE_TABLE_LAST_ENTRY}
};
static IviRangeTable  triggerDelayTable = {
    IVI_VAL_RANGED,     /* type */
    VI_TRUE,            /* hasMin */
    VI_TRUE,            /* hasMax */
    VI_NULL,            /* customInfo */
    triggerDelayTableEntries
};
```

This range table declares the minimum and maximum value for a trigger delay attribute.

Static and Dynamic Range Tables

You can pass the address of a range table to `Ivi_AddAttributeViInt32` and `Ivi_AddAttributeViReal64` to associate a single range table with an attribute. This is called a *static* range table.

Some cases exist in which the set of valid values for one attribute depends on the current setting of another attribute. In such cases, you can define multiple static range tables for the attribute. Instead of specifying one range table when you call `Ivi_AddAttributeViInt32` or `Ivi_AddAttributeViReal64`, call `Ivi_SetAttrRangeTableCallback` to install a range table callback. Each time the IVI engine invokes the range table callback, the callback must obtain the value of the second attribute and then return a pointer to the appropriate range table.

You can obtain the address of the current range table for an attribute by calling the `Ivi_GetAttrRangeTable` function. `Ivi_GetAttrRangeTable` invokes the range table callback if the attribute has one. Otherwise, `Ivi_GetAttrRangeTable` returns the address

of the range table you specify when you call `Ivi_AddAttributeViInt32` or `Ivi_AddAttributeViReal64`. You can call `Ivi_GetStoredRangeTablePtr` to bypass the range table callback and get the address of the range table you specify when you call `Ivi_AddAttributeViInt32` or `Ivi_AddAttributeViReal64`. You can replace this range table with a different one by calling `Ivi_SetStoredRangeTablePtr`.

In certain instances, the set of valid values for an attribute varies so much that you must create a large number of static range tables for the attribute. A better approach in this case is to modify the contents of a single range table depending on the current settings of other attributes. If you want to modify the contents of a range table dynamically, you must create a *dynamic* range table using `Ivi_RangeTableNew`. You also must install a range table callback using `Ivi_SetAttrRangeTableCallback`. In the range table callback, you modify the contents of the range table and then return its address. To allow for multithreading and multiple sessions to the same instrument type, you must create a separate dynamic range table for each IVI session. Pass the address of the dynamic range table to `Ivi_AddAttributeViInt32` or `Ivi_AddAttributeViReal64` when you create the attribute. Your range table callback can then use the `Ivi_GetStoredRangeTablePtr` function to obtain the address of the dynamic range table for the session before modifying its contents.

Default Check and Coerce Callbacks

The IVI Library supplies default check and coerce callback functions that use the range tables. When you install a static range table callback for an attribute, the IVI engine automatically installs the default check callback. For coerced range tables, the IVI engine also installs the coerce callback. When you install a range table callback for an attribute, the IVI engine automatically installs the default check and coerce callbacks. The following are the names of the default callbacks that use range tables:

```
Ivi_DefaultCheckCallbackViInt32
Ivi_DefaultCoerceCallbackViInt32
Ivi_DefaultCheckCallbackViReal64
Ivi_DefaultCoerceCallbackViReal64
```

You can invoke the default callbacks from your own check and coerce callbacks. To add functionality to one of the default callbacks, install a callback that performs the additional functionality before or after calling the default callback.

Comparison Precision

Because of the imprecision inherent in the computer representation of floating-point numbers, it is not always possible to determine if two `ViReal64` values are equal by comparing them based on strict equality. When attempting to find `ViReal64` values in range tables, the IVI engine performs comparisons using 14 decimal digits of precision.

Some instruments represent floating-point numbers differently than computers do. Consequently, comparisons between instrument and computer floating-point numbers can be less precise than comparisons between two computer floating-point numbers. The IVI engine makes a comparison between an instrument and a computer floating-point number whenever it compares an attribute cache value it obtained from the instrument against a new value to which you attempt to set the attribute. If the values are equal within the degree of precision that you specify for the attribute, the IVI engine does not invoke the write callback.

Call `Ivi_AddAttributeViReal64` to specify the degree of precision for an attribute. You can specify from 1 to 14 digits of precision. The more digits of precision, the closer the two values must be for the IVI engine to consider them equal. To obtain or modify the degree of precision for an attribute, call `Ivi_GetAttrComparePrecision` or `Ivi_SetAttrComparePrecision`.

The IVI engine uses the compare precision when the binary representations of two `ViReal64` values are not exactly equal. It uses the following logic, where `a` and `b` are the values you want to compare, and `d` is the number of digits of precision.

```
if a == 0
    if |b| < 10^-(d-1)
        then a == b.
else /* a != 0 */
    if |a - b| / |a| < 10^-(d-1)
        then a == b
```

IVI State-Caching Mechanism

When you enable the state-caching mechanism, the IVI engine maintains a software copy of the current instrument setting for each attribute. If the IVI engine determines that the cache value accurately reflects the state of the instrument, it considers the cache value to be valid. If state-caching is disabled or the IVI engine determines that the cache value does not accurately reflect the state of the instrument, it considers the cache value to be invalid.

When you call one of the `Ivi_GetAttribute` functions to query the current setting for an attribute and the cache value for that attribute is invalid, the IVI engine asks the driver to perform instrument I/O to obtain the setting. The IVI engine does this by invoking the read callback for the attribute. After the read callback obtains the current setting, the IVI engine stores the value in the cache, marks the cache as valid, and returns the value to the caller. If the cache value is already valid, the IVI engine returns the cache value to the caller immediately.

When you call one of the `Ivi_SetAttribute` functions to specify a new setting for an attribute and the cache value for the attribute is invalid or different than the value you specify, the IVI engine invokes the write callback for the attribute, which asks the driver to send the new setting for the instrument. If the write callback reports that it successfully modified the instrument setting, the IVI engine stores the new value in the cache and marks the cache as valid.

Initial Instrument State

The IVI state-caching mechanism makes no assumptions about the initial state of an instrument. When the driver creates attributes during the initialization of an IVI session, the IVI engine marks the attribute cache values as invalid. Thus, the first call to an `Ivi_GetAttribute` or `Ivi_SetAttribute` function for an instrument attribute causes the driver to perform instrument I/O.

It is the user's responsibility to set the instrument to a known state. Typically, the user does this by calling the instrument driver high-level configuration functions.

Special Cases

Instrument command sets do not always map perfectly onto the state-caching model. In such cases, you must take special actions to ensure that the state-caching mechanism works properly given the particular instrument's behavior. The following sections describe four possible situations.

Changing the Value of One Attribute Invalidates Another

In many cases, changing the value of one attribute causes the setting of another attribute in the instrument to change. For example, changing the measurement function in a DMM can change the range setting. In this case, the specific driver must indicate to the IVI engine that setting the first attribute invalidates the second attribute. The specific driver can notify the IVI engine of this relationship by calling the `Ivi_AddAttributeInvalidation` function. Whenever a call to an `Ivi_SetAttribute` function changes the value of the first attribute, the IVI engine marks the cache value of the second attribute as invalid. For each IVI session, the IVI engine maintains a list of invalidation relationships.

The specific driver also can call the `Ivi_InvalidateAttribute` function to invalidate the cache value of an attribute directly.

In some cases, you might think you can determine the new value of the second attribute and set its cache value. It is better to invalidate the second attribute. Any assumption you make about the new state of the second attribute depends on instrument behavior that might change in future revisions of the instrument. As the *Initial Instrument State* section in this chapter explains, it is the user's responsibility to set the instrument to a known state.

Two Attributes Invalidate Each Other

In rare cases, changing the value of one instrument setting can affect another instrument setting, and changing the value of the second instrument setting can affect the first. For example, changing the measurement range in a DMM commonly affects the resolution setting. If changing the resolution setting can change the measurement range, the invalidation relationship is two-way.

The proper way to handle this situation is to impose a one-way invalidation model in the instrument driver. Identify one attribute as dominant and the other as subordinate. Call `Ivi_AddAttributeInvalidation` to notify the IVI engine that changing the value of the dominant attribute invalidates the subordinate attribute. Range check and coerce values for the subordinate attribute based on the current setting of the dominant attribute. In the example, select the measurement range as the dominant attribute. Range check and coerce the resolution attribute so that you cannot set the resolution to a value that would cause the instrument to modify the measurement range setting.

Setting or Getting the Values of Two Attributes in One Command

Some instruments have command sets that force the driver to set or get two attributes at once. The driver cannot set or get the value of one attribute without also setting or getting the value of another. In this case, the read callback for each coupled attribute must record the value it obtained for the other attribute. The read callback can record the value it obtained for the other attribute by calling the appropriate `Ivi_SetAttribute` function and passing `IVI_VAL_SET_CACHE_ONLY` as the **optionFlags** parameter.

The write callbacks can be more difficult to handle. Generally, one of the coupled attributes is dominant. If so, the write callback for the dominant attribute must calculate the default value of the subordinate attribute, include the value in the command string it sends to the instrument, and call the appropriate `Ivi_SetAttribute` function with the `IVI_VAL_SET_CACHE_ONLY` flag to cache the new value of the subordinate attribute. The write callback for the subordinate attribute must call the `Ivi_GetAttribute` function to obtain the current value of the dominant attribute and use its value in the command string. When a high-level driver function wants to set the two attributes, it must set the dominant attribute first. Handling such order dependencies while minimizing instrument I/O is one of the benefits that the high-level driver functions provide to application programs.

Instrument Coerces Values

In some cases, an instrument accepts a range of values for an attribute but coerces them into discrete settings. For example, a DMM might have three maximum reading ranges, 10.0, 100.0, and 1,000.0, but accept any value from 1.0 to 1,000.0. If you set the maximum reading range to 50.0, the instrument coerces the value to 100.0. In responding to a query, the instrument returns 100.0. If, after you set the attribute to 50.0, the IVI engine stores 50.0 in

the cache, the cache does not accurately reflect the state of the instrument. Instead, the IVI engine must store 100.0 in the cache.

Thus, state-caching requires the driver to coerce the value in software before sending it to the instrument. Requiring coercion is especially important for drivers that comply with a standard class definition. To handle any specific driver within a class, the class definition must allow for a continuous range of values. Each specific driver must coerce the range of values into the discrete settings the instrument uses.

The driver can coerce the range of values by using a coerce callback. The easiest way to use a coerce callback is to create a coerced range table for the attribute. The IVI engine automatically installs a default coerce callback for attributes that have coerced range tables. Refer to the *Range Tables* and *Coerce Callback* sections in this chapter for more information.

In rare instances, an instrument might coerce a value using an algorithm that is undocumented or too complicated to encapsulate in a range table or a coerce callback. You can let the instrument coerce the value and still retain much of the benefits of state-caching by using the `IVI_VAL_COERCEABLE_ONLY_BY_INSTR` flag. Refer to the documentation for this flag in the *Attribute Flags* section of this chapter.

Enabling and Disabling State-Caching

The user can enable or disable the state-caching mechanism for an entire IVI session by setting the `IVI_ATTR_CACHE` attribute. Nevertheless, a specific instrument driver can override the user's choice on an attribute-by-attribute basis. The driver can set the `IVI_VAL_NEVER_CACHE` flag for an attribute to prevent the IVI engine from using the cache. The driver can set the `IVI_VAL_ALWAYS_CACHE` flag for an attribute to force the IVI engine to always use the cache value, if valid, regardless of the state of `IVI_ATTR_CACHE`.

Attribute Callback Functions

For each attribute, an instrument driver can install up to six callback functions. The IVI engine invokes these functions in the context of the state-caching mechanism. Each of the six callbacks performs a specific task. The six callbacks types are read, write, check, coerce, compare, and range table. A driver can install the read and write callbacks when it creates the attribute using one of the `Ivi_AddAttribute` functions. Also, the IVI Library contains functions that install each of the first five callback types for each of the six attribute data types, for example, `Ivi_SetAttrReadCallbackViInt32` and `Ivi_SetAttrCheckCallbackViReal64`. The IVI Library contains only one function, `Ivi_SetAttrRangeTableCallback`, which installs a range table callback.

Whether a driver installs a read and write callback for an attribute depends on what the driver uses the attribute for.

- Attributes that represent instrument settings always have read and write callbacks.
- Attributes that store internal driver data or software-only options generally do not have read or write callbacks. Instead, the driver uses the state-caching mechanism to store the values. The driver must set the `IVI_VAL_ALWAYS_CACHE` flag on such attributes.
- Attributes that represent values that the driver recalculates upon each query have read callbacks but no write callbacks. The read callback performs the recalculation of the value. To ensure that the IVI engine always invokes the read callback, the driver must set the `IVI_VAL_NEVER_CACHE` flag on such attributes.

In discussing the various callback types, the following sections assume that the attributes represent instrument settings.

Read Callback

The read callback function for an attribute obtains the current setting for the attribute from the instrument. Typically, this involves sending a query command to the instrument, reading the response from the instrument, and interpreting the response. The IVI engine invokes the read callback function when you request the current value of the attribute and the attribute cache value is invalid.

If the instrument expresses the setting in units that are different from the units the driver uses, the read callback must translate the value it receives from the instrument. For example, oscilloscopes typically implement the `VERTICAL_RANGE` attribute in terms of volts-per-division, whereas oscilloscope instrument drivers use values that represent the overall voltage range. In this case, the read callback must translate volts-per-division values into overall voltage range values.

If you do not want the IVI Library to invoke a read callback, specify `VI_NULL` for the **readCallback** parameter to the `Ivi_AddAttribute` function.

Write Callback

The write callback function for an attribute is responsible for sending a new attribute setting to the instrument. The IVI engine invokes the write callback function when you specify a new value for the attribute and the cache value is invalid or is not equal to the new value.

If the instrument expresses the setting in units that are different from the units the driver uses, the write callback must translate the value before sending the value to the instrument. For example, oscilloscopes typically implement the `VERTICAL_RANGE` attribute in terms of volts-per-division, whereas oscilloscope instrument drivers use values that represent the overall voltage range. In this case, the read callback translates voltage range values into overall volts-per-division values.

For example, if the instrument uses volts-per-division but the driver uses `VERTICAL_RANGE`, the read callback translates the `VERTICAL_RANGE` value into a volts-per-division value before it formats the instrument command string.

If you do not want the IVI Library to invoke a write callback, specify `VI_NULL` for the **writeCallback** parameter to the `Ivi_AddAttribute` function.

Check Callback

The check callback function for an attribute validates new values to which you attempt to set the attribute.

The IVI engine supplies default check callbacks for `ViInt32` and `ViReal64` attributes. The default check callbacks use the range table or range table callback for the attribute to validate the value. The IVI engine automatically installs one of the default check callbacks when you create a `ViInt32` or `ViReal64` attribute with a range table. The IVI engine also installs the default check callback when you install a range table callback for the attribute and the attribute does not already have a check callback.

You can invoke the default check callback from your callback. To add functionality to one of the default check callbacks, install a check callback that performs the additional functionality before or after calling `Ivi_DefaultCheckCallbackViInt32` or `Ivi_DefaultCheckCallbackViReal64`.

Coerce Callback

The IVI engine invokes the coerce callback function when you set an attribute to a new value. The IVI engine invokes the coerce callback after the engine invokes the check callback. The coerce callback converts the value you specify into the value to send to the instrument.

In general, two cases exist in which an instrument driver must coerce attribute values. In these cases, the instrument defines a set of discrete values for an attribute, and it is possible to map a range of values onto each member of the discrete set. For example, a DMM might accept 10.0, 100.0, or 1,000.0 as the maximum reading voltage. If a user specifies 50.0 as the maximum voltage, the correct action is to set the DMM to 100.0.

In the first case, the instrument itself coerces values in this manner. It accepts values from 1.0 to 1,000.0 and coerces them to 10.0, 100.0, or 1,000.0. For the IVI state-caching mechanism to work properly, the cache value for the attribute must reflect the coerced value in the instrument. In order for the cached value and the value in the instrument to be the same, the instrument driver must coerce the value before sending the value to the instrument. In the example, the instrument driver must coerce 50.0 to 100.0 and send 100.0 to the instrument. The IVI engine caches the coerced value, which in this example is 100.0.

In the second case, the instrument does not coerce values. Instead, it accepts only the values in the discrete set. Although you can write the instrument driver so that it accepts only the discrete values, doing so is not feasible if you want the driver to comply with a standard class definition. In the previous example, the DMM in question might accept only 10.0, 100.0, or 1,000.0. However, another DMM might accept 10.0, 50.0, 100.0, 500.0, or 1,000.0. A third might accept a continuous range of values and coerce them to still another discrete set. Standard class definitions handle such cases by allowing users to specify a continuous set of values and requiring the instrument drivers to coerce them.

Note A third case exists that seems to require value coercion but does not. In this case, the instrument expresses a setting in units that are different from the units a class definition uses. For example, oscilloscopes typically implement the `VERTICAL_RANGE` attribute in terms of volts-per-division, whereas the oscilloscope class definition specifies values that represent the overall voltage range. The driver must translate between volts-per-division and overall voltage range before sending a new value to the instrument and after receiving the current setting from the instrument. Thus, it is best to do the translations in the read and write callbacks for the attribute.

You can implement coercion through a coerced range table. Specify a coerced range table when you call `Ivi_AddAttributeViInt32` or `Ivi_AddAttributeViReal64`. When you do so, the IVI engine automatically installs a default coerce callback that uses the range table. It also installs the default coerce callback when you install a range table callback for the attribute, and the attribute does not already have a coerce callback.

You can invoke the default coerce callback from your own callback. To add functionality to one of the default coerce callbacks, install a coerce callback that performs the additional functionality before or after calling `Ivi_DefaultCoerceCallbackViInt32` or `Ivi_DefaultCoerceCallbackViReal64`.

The IVI engine also supplies a default coerce callback for `ViBoolean` attributes. The callback coerces all non-zero values to `VI_TRUE (1)`. The IVI engine always installs the callback when you create a `ViBoolean` attribute.

Generally, `ViString`, `ViSession`, and `ViAddr` attributes do not have coerce callbacks. When you add one of these types of attributes, its coerce callback is `VI_NULL`.

Compare Callback

The IVI engine invokes the compare callback function for an attribute only when comparing cache values it obtains from the instrument against new values to which you attempt to set the attribute. The IVI engine invokes the compare callback after it invokes the check callback and the coerce callback. If the compare callback determines that the two values are equal, the IVI engine does not call the write callback for the attribute. If the attribute does not have a compare callback, the IVI engine makes the comparison based on strict equality.

When you create a `ViReal64` attribute, the IVI engine automatically installs a default compare callback. The default compare callback uses the degree of precision you pass to `Ivi_AddAttributeViReal64`. The IVI engine installs the default compare callback for `ViReal64` attributes rather than comparing based on strict equality because of differences between computer and instrument floating-point representations. Refer to the *Comparison Precision* section in this chapter for more information on the compare callback function.

Typically, compare callbacks are necessary only for `ViReal64` attributes.

Range Table Callback

Use the range table callback to dynamically determine which static range table you want to use for an attribute. You also can use the range table callback to modify the contents of a dynamic range table that you create with `Ivi_RangeTableNew`. Refer to the *Static and Dynamic Range Tables* section in this chapter for more information on this topic.

The IVI engine invokes the range table callback only when the `Ivi_GetAttrRangeTable` function executes. If the attribute has a range table callback, `Ivi_GetAttrRangeTable` returns the range table pointer that the callback returns. Otherwise, it returns the range table pointer you associate with the attribute when you call `Ivi_AddAttributeViInt32`, `Ivi_AddAttributeViReal64`, or `Ivi_SetStoredRangeTablePtr`.

The IVI engine calls `Ivi_GetAttrRangeTable` from the default check and coerce callbacks for `ViInt32` and `ViReal64` attributes. If you install your own check or coerce callback function, you can call `Ivi_GetAttrRangeTable` from your callback.

Session Callback Functions

The IVI engine allows an instrument driver to install two callback functions that are global to an entire IVI session: operation complete and check status.

Operation Complete Callback

The operation complete callback waits until the instrument finishes processing all pending operations. Many instruments cannot accept a command while processing a previous one. If an instrument takes a long time to process an attribute setting, the driver must avoid sending another command until the instrument is ready. By setting `IVI_VAL_WAIT_FOR_OPC_AFTER_WRITES` for the attribute, the driver causes the IVI engine to invoke the operation complete callback after invoking the write callback for the attribute.

High-level instrument driver functions can call the operation complete callback directly. For example, a high-level measurement function might use the operation complete callback

to wait until the instrument has completed an acquisition before attempting to retrieve the data from the instrument.

The IVI engine invokes the operation complete callback in the following two cases:

- Before invoking the read callback for attributes for which the `IVI_VAL_WAIT_FOR_OPC_BEFORE_READS` flag is set
- After invoking the write callback for attributes for which the `IVI_VAL_WAIT_FOR_OPC_AFTER_WRITES` flag is set

The operation complete callback is optional. By default, an IVI session does not have an operation complete callback. The driver can install an operation complete callback by calling `Ivi_SetAttributeViAddr` with the `IVI_ATTR_OPC_CALLBACK` attribute.

The operation complete callback must have the following prototype:

```
ViStatus _VI_FUNC <function name>(ViSession vi, ViSession io);
```

Check Status Callback

Most instruments have status registers and an error queue. The status register indicates whether one or more errors are in the queue. Typically, the check status callback queries the status registers to determine if the instrument encounters an error. If it has, the driver returns the `IVI_ERROR_INSTR_SPECIFIC` error code. The user then calls the *`Prefix`*`_error_query` function. The *`Prefix`*`_error_query` function extracts instrument-specific error information from the instrument's error queue and returns it to the user.

The IVI engine invokes the check status callback in the `Ivi_SetAttribute` and `Ivi_GetAttribute` functions. It does so only when a *`Prefix`*`_SetAttribute` function passes the `IVI_VAL_DIRECT_USER_CALL` flag in **optionFlags** parameter of the corresponding `Ivi_SetAttribute` function.

The high-level functions in an instrument driver also invoke the check status callback. They do so at the end of functions that make one or more `Ivi_SetAttribute` or `Ivi_GetAttribute` calls or that perform direct instrument I/O.

The user can disable the check status callback by setting the `IVI_ATTR_QUERY_INSTR_STATUS` attribute to `VI_FALSE`. The driver can disable the check status callback for a particular attribute by setting the `IVI_VAL_DONT_CHECK_STATUS` flag.

The check status callback is optional. By default, an IVI session does not have a check status callback. The driver can install a check status callback by calling `Ivi_SetAttributeViAddr` with the `IVI_ATTR_CHECK_STATUS_CALLBACK` attribute.

The check status callback must have the following prototype:

```
ViStatus _VI_FUNC <function name> (ViSession vi, ViSession io);
```

Instruments without Error Queues

Some instruments have status registers but no error queue. All the error information is in the status registers. The act of reading the status registers clears them. When the check status callback queries the registers, it destroys the error information.

In this case, the check status callback must queue the error information in software so that the *Prefix*_error_query function can return it. The IVI library contains functions to manage the software error queue for a session. The check status callbacks calls Ivi_QueueInstrSpecificError to add the error information to the queue.

The *Prefix*_error_query function first calls Ivi_InstrSpecificErrorQueueSize to determine if the software queue is empty. If the queue is empty, *Prefix*_error_query calls the check status callback and then checks the software queue size again. In either case, if there is an error in the queue, *Prefix*_error_query calls Ivi_DequeueInstrSpecificError to extract the error information and then returns the error information to the user.

Channels

Many instruments have multiple channels. Some attributes apply to the instrument as a whole, while other attributes apply to each specific channel. For a few instruments, some attributes apply to only a subset of the available channels.

Each instrument driver that supports multiple channels must declare names for the channels. Each name is in the form of a string and is called a *channel string*. The driver calls the Ivi_BuildChannelTable function to declare the channel strings during the initialization of the IVI session. If, for some reason, the driver wants to declare additional channels later, it can call Ivi_AddToChannelTable. The driver can replace the list of channel strings by calling Ivi_BuildChannelTable again.

Instrument drivers that do not support multiple channels also must call Ivi_BuildChannelTable. By convention, all such drivers declare one channel with the name 1.

An attribute that applies separately to each channel is called a *channel-based* attribute. The driver marks channel-based attributes by setting the IVI_VAL_MULTI_CHANNEL flag for the attribute when it calls the appropriate Ivi_AddAttribute function. The driver marks channel-based attributes even for attributes that apply to only a subset of channels. To declare

a subset of channels to which an attribute applies, the driver calls `Ivi_RestrictAttrToChannels` after calling `Ivi_BuildChannelTable`. To determine if an attribute applies to a particular channel, call `Ivi_ValidateAttrForChannel`.

Virtual Channel Names

Each instrument driver specifies its own set of valid channel strings. If an application program opens an IVI session through a class instrument driver, the program expects to work with any specific instrument driver that complies with the class. Each specific driver, however, can have a different set of channel strings. To allow the application program to use one set of channel names for all possible specific drivers, IVI has the concept of a *virtual channel name*.

The user can specify a mapping between specific driver channel strings and virtual channel names in the `ivi.ini` configuration file by using MAX. The application program can then reference the virtual channel names rather than specific driver channel strings.

Refer to the *Configuration Entries* section in this chapter for more information on the `ivi.ini` configuration file.

Passing Channel Names to IVI Functions

Many IVI Library functions, including the `Ivi_Get/Check/SetAttribute` functions, have a **channelName** parameter. When you call one of these functions for a channel-based attribute, you must pass a valid channel string or virtual channel name for the **channelName** parameter. When you call one of these functions on an attribute that is not channel-based, you must pass `VI_NULL` or an empty string.

Coercing and Validating Channel Names

If you pass a virtual channel name to one of the `Ivi_Get/Check/SetAttribute` functions, the IVI engine converts the virtual channel name into a specific driver channel string before invoking a read, write, check, coerce, compare, or range table callback function.

If a user-callable instrument driver function uses channel strings directly, it must call the `Ivi_CoerceChannelName` to validate the channel name parameter it receives. If the user passes a virtual channel name, `Ivi_CoerceChannelName` converts it to specific driver channel string.

High-Level Driver Functions

Most of the discussion in this chapter focuses on the IVI attribute model, callbacks, and state-caching mechanism. These concepts are important for the low-level implementation of instrument drivers. Most users, however, think in terms of actions, such as measure or configure channel, rather than setting individual attributes. To provide a user-friendly API,

instrument drivers must provide high-level functions that set and/or get the values of multiple instrument attributes. Depending on the instrument, it is often necessary to set related attributes in a particular order. The high-level functions handle these order dependencies. Examples of high-level functions are `FL45_ConfigureMeasurement`, `FL45_ConfigureTrigger`, and `FL45_Read`.

IVI provides standardized interfaces for implementing range checking, status checking, simulation, and multithread safety. Users can enable or disable range checking, status checking, and simulation. Users also can use one instrument session in multiple execution threads. The IVI engine does as much as it can to implement these user capabilities. The high-level functions in each driver also must help implement these capabilities. The next four sections in this chapter explain what the IVI engine does to implement the user capabilities and what the high-level driver functions must do.

Chapter 3, *Programming Guidelines for Instrument Drivers*, contains guidelines and example code that illustrates how high-level functions implement the user capabilities. Also, if you use the **Tools»Create IVI Instrument Driver** command to generate an instrument driver, the resulting instrument driver source code contains skeleton code for high-level functions. The skeleton code follows the guidelines in Chapter 3, *Programming Guidelines for Instrument Drivers*.

Range Checking

IVI provides users with the ability to enable or disable range checking. Range checking is most useful during debugging. After users validate their programs, they can disable range checking to maximize performance. By default, range checking is enabled. The user disables range checking by setting the `IVI_ATTR_RANGE_CHECK` attribute to `VI_FALSE`, by setting the `RangeCheck` tag in the **optionsString** parameter of `Prefix_InitWithOptions` to `VI_FALSE`, or by specifying the `RangeCheck` entry to be false in the `ivi.ini` configuration file.

Refer to the *Configuration Entries* section in this chapter for more information on the `ivi.ini` configuration file.

The IVI engine honors the `IVI_ATTR_RANGE_CHECK` attribute when you call one of the `Ivi_SetAttribute` functions. The IVI engine calls the check callback for the attribute only if `IVI_ATTR_RANGE_CHECK` is `VI_TRUE`. If a high-level function passes all its configuration parameters to `Ivi_SetAttribute` functions, it does not have to do any range checking on its own.

Sometimes, however, a high-level function must implement range checking for certain parameters. In that case, the high-level function must range check only if `IVI_ATTR_RANGE_CHECK` is `VI_TRUE`. The IVI Library contains the `Ivi_RangeChecking`

function so that the high-level function can quickly determine the state of the IVI_ATTR_RANGE_CHECK attribute.

Status Checking

Most instruments support the ability to query the instrument's status. The instrument returns an indication of whether it has encountered any errors. IVI instrument drivers have the ability to check the instrument status after every function that interacts with the instrument. IVI provides users with the ability to enable or disable status checking. Status checking is most useful during debugging. After users validate their programs, they can disable status checking to maximize performance. By default, status checking is enabled. The user disables status checking by setting the IVI_ATTR_QUERY_INSTR_STATUS attribute to VI_FALSE, by setting the QueryInstrStatus tag in the **optionsString** parameter of *Prefix*_InitWithOptions to VI_FALSE, or by specifying the QueryInstrStatus entry to be false in the ivi.ini configuration file.

Refer to the *Configuration Entries* section in this chapter for more information on the ivi.ini configuration file.

An instrument driver defines a check status callback to encapsulate the code that queries the instrument status and interprets the response. Refer to the *Check Status Callback* section in this chapter for more information.

The IVI engine invokes the check status callback only when a user calls one of the *Prefix*_SetAttribute or *Prefix*_GetAttribute functions that an instrument driver exports. The instrument driver marks these calls by passing the IVI_VAL_DIRECT_USER_CALL flag to the Ivi_SetAttribute or Ivi_GetAttribute function. In this case, the IVI engine invokes the check status callback after it invokes the read or write callback, but only if the IVI_ATTR_QUERY_INSTR_STATUS attribute is VI_TRUE.

When other instrument driver functions call the Ivi_SetAttribute or Ivi_GetAttribute functions, the IVI engine does *not* invoke the check status callback. Because high-level functions often make multiple calls to the Ivi_SetAttribute and Ivi_GetAttribute functions, invoking the check status callback each time would be very wasteful. Consequently, the high-level functions must invoke the check status callback before returning. They must do so only if the IVI_ATTR_QUERY_INSTR_STATUS attribute is VI_TRUE and only if instrument I/O has actually occurred. Instrument I/O occurs when a high-level function performs direct instrument I/O or when calls to the Ivi_SetAttribute functions invoke write callbacks because the cache values are invalid or not equal to the requested values.

The IVI Library contains the Ivi_QueryInstrStatus function that instrument drivers can call to determine whether the IVI_ATTR_QUERY_INSTR_STATUS attribute is VI_TRUE. The IVI Library also contains the Ivi_NeedToCheckStatus function that instrument drivers

can call to determine whether any instrument interaction has occurred since the last time the driver or the engine invoked the check status callback. To help drivers maintain this information, the IVI Library contains the `Ivi_SetNeedToCheckStatus` function. After an instrument driver performs status checking, it must call `Ivi_SetNeedToCheckStatus` with `VI_FALSE`. Before performing direct instrument I/O, the instrument driver must call `Ivi_SetNeedToCheckStatus` with `VI_TRUE`. If you use the **Tools»Create IVI Instrument Driver** command to generate a driver based on a class definition, the initial instrument driver source code contains a *`Prefix`*`_CheckStatus` function that handles these requirements. The skeleton code for the high-level functions calls *`Prefix`*`_CheckStatus`.

Simulation

IVI provides users with the ability simulate an instrument. Simulation is useful when the instrument is not available, as when the user develops a test program concurrently with the development of the test system hardware. By default, simulation is disabled. The user enables simulation by setting the `IVI_ATTR_SIMULATE` attribute to `VI_TRUE`, by setting the `Simulate` tag in the **optionsString** parameter of the *`Prefix`*`_InitWithOptions` to `VI_TRUE`, or by specifying the `Simulate` entry to be false in the `ivi.ini` configuration file.

Refer to the *Configuration Entries* section in this chapter for more information on the `ivi.ini` configuration file.

The IVI engine handles simulation for attributes automatically. For all attributes, range checking and coercion still occur. When simulation is enabled and the `IVI_VAL_USE_CALLBACKS_FOR_SIMULATION` flag is not set for the attribute, the IVI engine refrains from calling read and write callbacks. Instead, the IVI engine merely records the values you set and returns these values when you query the attribute. If, when you query the attribute, you have not yet set it to a value, the IVI engine returns the default value the driver passes to the appropriate `Ivi_AddAttribute` function.

If a high-level function consists of nothing but `Ivi_SetAttribute` or `Ivi_GetAttribute` calls and does not return any values other than the status code, the high-level function does not need to take any action to support simulation. If the high-level function performs direct instrument I/O, the high-level function must refrain from doing so when `IVI_ATTR_SIMULATE` is `VI_TRUE`. Also, if the high-level function returns values, it must simulate values if both `IVI_ATTR_SIMULATE` and `IVI_ATTR_USE_SPECIFIC_SIMULATION` are `VI_TRUE`. The IVI Library contains the `Ivi_Simulating` and `Ivi_UseSpecificSimulation` functions that the high-level function can call to quickly determine the state of the `IVI_ATTR_SIMULATE` and `IVI_ATTR_USE_SPECIFIC_SIMULATION` attributes.

Note Class drivers use `IVI_ATTR_USE_SPECIFIC_SIMULATION` to determine if simulation is performed within the specific driver or through an advanced simulation driver.

Caution The instrument driver must ensure that no instrument I/O occurs when simulation is enabled. The instrument driver, therefore, must be careful not to perform instrument I/O in callbacks or internal functions that might execute even when simulation is enabled. I/O that must not occur during simulation includes check callbacks and coerce callbacks.

Multithread Safety

Users can use IVI drivers in multithreaded applications. Multiple execution threads can use the same IVI session without interfering with each other.

To make this work, IVI provides a way to lock and unlock an IVI session. The IVI Library contains the `Ivi_LockSession` and `Ivi_UnlockSession` functions. Instrument drivers export these functions as *`Prefix`*`_LockSession` and *`Prefix`*`_UnlockSession`. The IVI engine, instrument drivers, and user applications must use the lock and unlock capabilities to enable safe multithreaded access to an IVI session.

Most IVI Library functions lock the IVI session on entry and unlock it on exit. Some IVI functions do not lock the session because they do not take an IVI session as a parameter. Others, such as `Ivi_RangeChecking` and `Ivi_Simulating`, do not lock the session so that they can execute as fast as possible. An instrument driver must not call these optimized functions unless the driver has already locked the session. The descriptions for the optimized functions note that restriction.

In general, the user-callable functions in an instrument driver must lock the IVI session on entry and unlock it on exit. The *`Prefix`*`_init` and *`Prefix`*`_InitWithOptions` functions do not lock the session. When these functions execute, the user does not yet have the IVI session handle to use in other execution threads. The *`Prefix`*`_close` function locks the session on entry but must unlock it before calling `Ivi_Dispose`.

By locking the session, the instrument driver functions ensure that no other execution thread can act on the session. In very high-level functions, this is sufficient. For instance, in a function that configures the instrument and takes a measurement, locking the session ensures that no other thread can disturb the instrument configuration before the function completes the measurement. In this case, the user application does not have to lock the session to ensure multithread safety.

In other cases, however, the user application must use the *`Prefix`*`_LockSession` and *`Prefix`*`_UnlockSession` to protect the state of the instrument from other threads. If, for instance, the application program calls a configure function and then a read function, the application must lock the session before the configure and unlock it after the read. Otherwise,

another thread could change the configuration between the time the program configures the instrument and the time the instrument takes the reading.

Notice that application programs have to use the `Prefix_LockSession` and `Prefix_UnlockSession` functions *only* if two or more threads use the same IVI session. In contrast, instrument drivers and the IVI Library always lock and unlock the session in case the application that uses them is multithreaded.

Cautions Multiple processes cannot use the same IVI session.

IVI does *not* prevent the same process or different processes from opening multiple sessions to the same physical device. The current version of IVI does not provide any capabilities to coordinate access to the same physical device from multiple IVI sessions. Do not open multiple IVI sessions to the same physical resource.

Configuration Entries

When a user opens an IVI session through a class driver, the class driver must determine which specific instrument driver to use. The IVI engine uses a special configuration file for this purpose. The name of the file is always `ivi.ini`. You configure the `ivi.ini` with MAX.

The configuration file allows the user to swap instruments without modifying, recompiling, or relinking the application program. The configuration file also allows the user to set the initial values for inherent IVI attributes, such as `IVI_ATTR_CACHE` and `IVI_ATTR_RANGE_CHECK`, without modifying the application program.

By default, the IVI engine looks for `ivi.ini` in the `niivi` directory under the VXI*plug&play* framework directory. The IVI Library contains the `Ivi_SetIviIniDir` function, which allows programs to specify a different location.

In some cases, the user might not want to rely on a configuration file. The IVI Library contains functions that create run-time configuration entries just like the configuration entries the IVI engine reads from the configuration file. The library also contains a function that writes the run-time configuration entries to a file.

Inherent IVI Attributes

Inherent IVI attributes are attributes that the IVI engine defines for all IVI sessions. The inherent IVI attributes are grouped into categories. The following list shows the IVI attributes in their categories.

Note Refer to the *IVI Driver Toolset Release Notes* for the latest information regarding inherent IVI attributes.

Table 2-2. Inherent IVI Attributes

Category or Attribute	Defined Constant
Session I/O	
VISA Resource Manager Session	IVI_ATTR_VISA_RM_SESSION
Instrument I/O Session	IVI_ATTR_IO_SESSION
Session Callbacks	
Check Status Callback	IVI_ATTR_CHECK_STATUS_CALLBACK
Operation Complete Callback	IVI_ATTR_OPC_CALLBACK
User Options	
Range Check	IVI_ATTR_RANGE_CHECK
Query Instrument Status	IVI_ATTR_QUERY_INSTR_STATUS
Cache	IVI_ATTR_CACHE
Simulate	IVI_ATTR_SIMULATE
Record Value Coercions	IVI_ATTR_RECORD_COERCIONS
Driver Setup	IVI_ATTR_DRIVER_SETUP
Class Drivers Only	
Interchangeability Check	IVI_ATTR_INTERCHANGE_CHECK
Spy	IVI_ATTR_SPY
Session Info	
Specific Driver Prefix	IVI_ATTR_SPECIFIC_PREFIX
Specific Driver Module Pathname	IVI_ATTR_MODULE_PATHNAME
Resource Descriptor	IVI_ATTR_RESOURCE_DESCRIPTOR
Logical Name	IVI_ATTR_LOGICAL_NAME
Class Driver Prefix	IVI_ATTR_CLASS_PREFIX
Instrument Capabilities	
Number of Channels	IVI_ATTR_NUM_CHANNELS
Class Group Capabilities	IVI_ATTR_GROUP_CAPABILITIES
Version Info	
Driver Major Version	IVI_ATTR_DRIVER_MAJOR_VERSION
Driver Minor Version	IVI_ATTR_DRIVER_MINOR_VERSION
Driver Revision	IVI_ATTR_DRIVER_REVISION
Class Major Version	IVI_ATTR_CLASS_MAJOR_VERSION
Class Minor Version	IVI_ATTR_CLASS_MINOR_VERSION
Class Revision	IVI_ATTR_CLASS_REVISION
Engine Major Version	IVI_ATTR_ENGINE_MAJOR_VERSION
Engine Minor Version	IVI_ATTR_ENGINE_MINOR_VERSION

Table 2-2. Inherent IVI Attributes (Continued)

Category or Attribute	Defined Constant
Engine Revision	IVI_ATTR_ENGINE_REVISION
Error Info	
Primary Error	IVI_ATTR_PRIMARY_ERROR
Secondary Error	IVI_ATTR_SECONDARY_ERROR
Error Elaboration	IVI_ATTR_ERROR_ELABORATION

- Session I/O category—Contains attributes you use to perform instrument I/O in a specific instrument driver.
- User Options category—Contains attributes that the user can set to affect the behavior of instrument drivers and the IVI engine.
- Session Info category—Contains attributes that provide information about the instrument driver that created the session and the physical resource it is using.
- Instrument Capabilities category—Contains attributes that describe various capabilities of the instrument and the driver, which is information that the IVI engine requires. Other attributes are useful for application programs.
- Version Info category—Contains attributes that provide version information about the instrument driver and the IVI engine.
- Error Info category—Contains attributes for reporting and retrieving error information.

Inherent Attribute Reference

This section contains detailed descriptions of the inherent IVI attributes. The attributes are arranged alphabetically. The description of each attribute indicates restrictions on its use. Specific instrument driver include files must not export any inherent attributes that are marked as hidden from the user.

IVI_ATTR_CACHE

Data Type: `ViBoolean`
Restrictions: None

Specifies whether to cache the value of attributes. When the user enables caching, the IVI engine keeps track of the current instrument settings so that it can avoid sending redundant commands to the instrument. Caching can significantly increase execution speed.

The user specifies the value of `IVI_ATTR_CACHE`. For a particular attribute, however, the driver can override the value of `IVI_ATTR_CACHE` by setting the `IVI_VAL_NEVER_CACHE` or `IVI_VAL_ALWAYS_CACHE` flag for the attribute.

The default value is `VI_TRUE`. If the user opens an instrument session by passing a logical name, the user can override the default value by using MAX to specify a value in the `ivi.ini` configuration file. Instrument drivers provide a *`Prefix`*`_InitWithOptions` function that

users can call to override both the default value and any value specified in the `ivi.ini` configuration file.

IVI_ATTR_CHECK_STATUS_CALLBACK

Data Type: `ViAddr`
Restrictions: Hidden from user

Specifies the check status callback for the session. The check status callback queries the instrument status.

If the user enables the `IVI_ATTR_QUERY_INSTR_STATUS` attribute, the specific driver calls the check status callback at the end of each user-callable function that interacts with the instrument. The IVI engine invokes the check status callback when the user calls one of the *`Prefix`*`_SetAttribute` or *`Prefix`*`_GetAttribute` functions that the driver provides.

The default value is `VI_NULL`. Leave the value as `VI_NULL` if you do not want a check status callback.

IVI_ATTR_CLASS_MAJOR_VERSION

Data Type: `ViInt32`
Restrictions: Not writable by user

The major version number of the class instrument driver.

The class driver sets the value of this attribute.

If the user opens an instrument session through a specific driver, the IVI engine generates an error when you attempt to set or get this attribute.

IVI_ATTR_CLASS_MINOR_VERSION

Data Type: `ViInt32`
Restrictions: Not writable by user

The minor version number of the class instrument driver.

The class driver sets the value of this attribute.

If the user opens an instrument session through a specific driver, the IVI engine generates an error when you attempt to set or get this attribute.

IVI_ATTR_CLASS_PREFIX

Data Type: `ViString`
Restrictions: Read-only

The prefix for the class instrument driver. The maximum character length of the prefix is 31 characters.

The prefix begins with a two-character vendor code defined in the VXI*plug&play* specification, *VPP-9: Instrument Vendor Abbreviations*, followed by characters that uniquely identify the driver. VXI*plug&play* specifications are available at `www.vxipnp.org`.

The name of each user-callable function in the class driver begins with this prefix. For example, if a class driver has a user-callable function named `IviDmm_init`, `IviDmm` is the prefix for that driver.

If the user opens an instrument session through a specific driver, the IVI engine generates an error when you attempt to set or get this attribute.

IVI_ATTR_CLASS_REVISION

Data Type: `ViString`
Restrictions: Not writable by user

A string that contains additional version information about the class instrument driver.

The class driver sets the value of this attribute.

If the user opens an instrument session through a specific driver, the IVI engine generates an error when you attempt to set or get this attribute.

IVI_ATTR_DRIVER_MAJOR_VERSION

Data Type: `ViInt32`
Restrictions: Not writable by user

The major version number of the specific instrument driver.

The specific driver sets the value of this attribute.

IVI_ATTR_DRIVER_MINOR_VERSION

Data Type: `ViInt32`
Restrictions: Not writable by user

The minor version number of the specific instrument driver.

The specific driver sets the value of this attribute.

IVI_ATTR_DRIVER_REVISION

Data Type: `ViString`
Restrictions: Not writable by user

A string that contains additional version information about the specific instrument driver.

The specific driver sets the value of this attribute.

IVI_ATTR_DRIVER_SETUP

Data Type: `ViString`
Restrictions: Read-only, hidden from user

Some cases exist where the user must specify instrument driver options at initialization time. An example is when specifying a particular instrument model from among a family of instruments that the driver supports. Specification of driver options at initialization time is useful when using simulation. The user can specify driver-specific options through the `DriverSetup` keyword in the **optionsString** parameter to the *`Prefix`*`_InitWithOptions` function. If the user opens an instrument session by passing a logical name, the user also can specify the options in the `ivi.ini` configuration file.

The default value is an empty string.

IVI_ATTR_ENGINE_MAJOR_VERSION

Data Type: `ViInt32`
Restrictions: Read-only

The major version number of the IVI engine.

The IVI engine sets the value of this attribute.

IVI_ATTR_ENGINE_MINOR_VERSION

Data Type: `ViInt32`
Restrictions: Read-only

The minor version number of the IVI engine.

The IVI engine sets the value of this attribute.

IVI_ATTR_ENGINE_REVISION

Data Type: `ViString`
Restrictions: Read-only

A string that contains additional version information about the IVI engine.

The IVI engine sets the value of this attribute.

IVI_ATTR_ERROR_ELABORATION

Data Type: `ViString`
Restrictions: None

An optional string that gives additional information about the primary error condition.

IVI_ATTR_FUNCTION_CAPABILITIES

Data Type: `ViString`
Restrictions: Not writable by user

A comma-separated string that identifies the class functions that the specific instrument driver implements.

IVI_ATTR_GROUP_CAPABILITIES

Data Type: `ViString`
Restrictions: Not writable by user

A comma-separated string that identifies the instrument class and the class-extension groups that the specific instrument driver implements.

IVI_ATTR_INTERCHANGE_CHECK

Data Type: `ViBoolean`
Restrictions: None

Specifies whether the class driver performs interchangeability checking. Each instrument class specification defines the rules for interchangeability checking for that class.

The user specifies the value of `IVI_ATTR_INTERCHANGE_CHECK`.

The default value is `VI_TRUE`. If the user opens an instrument session by passing a logical name, the user can override the default value by using MAX to specify a value in the `ivi.ini` configuration file. Instrument drivers provide a *`Prefix`*`_InitWithOptions` function that users can call to override both the default value and any value specified in the `ivi.ini` configuration file.

If the user opens an instrument session through a specific driver, the IVI engine generates an error when you attempt to set or get this attribute.

IVI_ATTR_IO_SESSION

Data Type: `ViSession`
Restrictions: Not writable by user

Specifies the I/O session that the specific driver uses to communicate with the instrument.

If a specific driver uses VISA instrument I/O, the driver passes the value of the `IVI_ATTR_VISA_RM_SESSION` attribute to the `viOpen` function and sets the `IVI_ATTR_IO_SESSION` attribute to the VISA session handle that `viOpen` returns.

The IVI engine passes the value of `IVI_ATTR_IO_SESSION` to the read and write callbacks the specific driver installs for its attributes. The `Ivi_IOSession` function provides convenient access to the value of this attribute.

IVI_ATTR_LOGICAL_NAME

Data Type: `ViString`
Restrictions: Read-only

When opening an IVI session through a class driver, the user passes a logical name to the class driver initialization function. The `ivi.ini` configuration file must contain an entry for the logical name. The logical name entry refers to a *virtual instrument* section in the configuration file. The virtual instrument section specifies a physical device and a specific instrument driver. By assigning the name of a different virtual instrument section to the logical name in the configuration file, the user can swap one instrument for another without changing source code

or recompiling or relinking the application program. This attribute indicates the logical name the user specified when opening the current IVI session.

IVI_ATTR_MODULE_PATHNAME

Data Type: `ViString`
Restrictions: Read-only

If the user opens the IVI session through a class driver, this attribute indicates the pathname the class driver uses to find the specific driver module file.

If the user opens an instrument session through a specific driver, the IVI engine generates an error when you attempt to set or get this attribute.

IVI_ATTR_NUM_CHANNELS

Data Type: `ViInt32`
Restrictions: Read-only

Indicates the number of channels that the specific instrument driver supports.
The specific driver declares the strings it uses to identify the channels by calling the `Ivi_BuildChannelTable` function during initialization of the IVI session. The driver does not set this attribute directly.

For each attribute for which the `IVI_VAL_MULTI_CHANNEL` flag is set, the IVI engine maintains a separate cache value for each channel.

IVI_ATTR_OPC_CALLBACK

Data Type: `ViAddr`
Restrictions: Hidden from user

Specifies the operation complete callback for the session. The operation complete callback waits until all pending instrument operations are complete.

If the `IVI_VAL_WAIT_FOR_OPC_AFTER_WRITES` flag is set for the attribute, the IVI engine invokes the operation complete callback after invoking the write callback.

If the `IVI_VAL_WAIT_FOR_OPC_BEFORE_READS` flag is set for the attribute, the IVI engine invokes the operation complete callback before invoking the read callback.

The default value is `VI_NULL`. Leave the value as `VI_NULL` if you do not want an operation complete callback.

IVI_ATTR_PRIMARY_ERROR

Data Type: `ViInt32`
Restrictions: None

A code that describes the first error that occurred since the last call to `Ivi_GetErrorInfo` on the session. The value follows the VXI*plug&play* completion code conventions. A negative value indicates an error condition. A positive value indicates a warning condition and specifies that no error occurred. A zero indicates that no error or warning occurred. The error and warning values can be status codes defined by IVI, VISA, class drivers, or specific drivers.

IVI_ATTR_QUERY_INSTR_STATUS

Data Type: `ViBoolean`
Restrictions: None

Specifies whether the instrument driver queries the instrument status after each user operation. The driver does so by calling the check status callback at the end of each user-callable function that interacts with the instrument. The IVI engine also invokes the check status callback when the user calls one of the *`Prefix`*`_SetAttribute` or *`Prefix`*`_GetAttribute` functions that the instrument driver provides. Querying the instrument status is very useful for debugging. After validating the program, the user can set this attribute to `VI_FALSE` to disable status checking and maximize performance.

The user specifies the value of `IVI_ATTR_QUERY_INSTR_STATUS`. The driver, however, can prevent the IVI engine from invoking the check status callback on a particular attribute by setting the `IVI_VAL_DONT_CHECK_STATUS` flag for the attribute.

The default value is `VI_TRUE`. If the user opens an instrument session by passing a logical name, the user can override the default value by using MAX to specify a value in the `ivi.ini` configuration file. Instrument drivers provide a *`Prefix`*`_InitWithOptions` function that users can call to override both the default value and any value specified in the `ivi.ini` configuration file.

The `Ivi_QueryInstrStatus` function provides convenient access to the value of this attribute.

IVI_ATTR_RANGE_CHECK

Data Type: `ViBoolean`
Restrictions: None

Specifies whether to validate attribute values and function parameters. If enabled, the instrument driver validates the parameter values that users pass to driver functions, and the IVI engine validates values that the driver or users pass to `SetAttribute` functions. The

IVI engine uses the range table, range table callback, or check callback for each attribute to validate its values. Range-checking parameters is useful for debugging. After validating the program, the user can set this attribute to `VI_FALSE` to disable range checking and maximize performance.

The user specifies the value of `IVI_ATTR_RANGE_CHECK`.

The default value is `VI_TRUE`. If the user opens an instrument session by passing a logical name, the user can override the default value by using MAX to specify a value in the `ivi.ini` configuration file. Instrument drivers provide a *`Prefix`*`_InitWithOptions` function that users can call to override both the default value and any value specified in the `ivi.ini` configuration file.

The `Ivi_RangeChecking` function provides convenient access to the value of this attribute.

IVI_ATTR_RECORD_COERCIONS

Data Type: `ViBoolean`
Restrictions: None

Specifies whether the IVI engine keeps a list of the value coercions it makes for `ViInt32` and `ViReal64` attributes.

If the driver provides a coerced range table, a range table callback that returns a coerced range table, or a coerce callback for an attribute, the IVI engine can coerce the values you specify for the attribute to canonical values the instrument accepts.

If the `IVI_ATTR_RECORD_COERCIONS` attribute is enabled, the IVI engine maintains a record of each coercion. The user calls the *`Prefix`*`_GetNextCoercionRecord` function in the specific driver to extract and delete the oldest coercion record from the list.

The user specifies the value of `IVI_ATTR_RECORD_COERCIONS`.

The default value is `VI_FALSE`. If the user opens an instrument session by passing a logical name, the user can override the default value by using MAX to specify a value in the `ivi.ini` configuration file. Instrument drivers provide a *`Prefix`*`_InitWithOptions` function that users can call to override both the default value and any value specified in the `ivi.ini` configuration file.

IVI_ATTR_RESOURCE_DESCRIPTOR

Data Type: `ViString`
Restrictions: Read-only

If the user opens the IVI session through a class driver, this attribute indicates the resource descriptor the class driver uses to identify the physical device to the specific driver.

If the user opens an instrument session through a specific driver, the IVI engine generates an `IVI_ERROR_NOT_CREATED_BY_CLASS` error when you attempt to set or get this attribute.

IVI_ATTR_SECONDARY_ERROR

Data Type: `ViInt32`
Restrictions: None

An optional code that provides additional information that concerns the primary error condition. The error and warning values can be status codes defined by IVI, VISA, class drivers, or specific drivers. Zero indicates no additional information.

IVI_ATTR_SIMULATE

Data Type: `ViBoolean`
Restrictions: None

Specifies whether to simulate instrument driver I/O operations. If simulation is enabled, specific instrument driver functions perform range checking and call `Ivi_GetAttribute` and `Ivi_SetAttribute` functions, but they do not perform instrument I/O. The IVI engine does not invoke the read and write callbacks for attributes, except when the `IVI_VAL_USE_CALLBACKS_FOR_SIMULATION` flag is set for an attribute.

For output parameters that represent instrument data, the instrument driver functions return hardcoded values. If the user opens the session through a class driver, the class driver loads a special simulation driver to generate output data in a more sophisticated manner unless the user sets the `IVI_ATTR_USE_SPECIFIC_SIMULATION` attribute to `VI_TRUE`.

The user sets the value of `IVI_ATTR_SIMULATE`.

The default value is `VI_FALSE`. If the user opens an instrument session by passing a logical name, the user can override the default value by using MAX to specify a value in the `ivi.ini` configuration file. Instrument drivers provide a *`Prefix`*`_InitWithOptions` function that users can call to override both the default value and any value specified in the `ivi.ini` configuration file.

The `Ivi_Simulating` function provides convenient access to the value of this attribute.

IVI_ATTR_SPECIFIC_PREFIX

Data Type: `ViString`
Restrictions: Read-only

The prefix for the specific instrument driver. The maximum character length for the prefix is eight characters.

The name of each user-callable function in the specific driver begins with this prefix. For example, if the Fluke 45 driver has a user-callable function named `FL45_init`, `FL45` is the prefix for that driver.

IVI_ATTR_SPY

Data Type: `ViBoolean`
Restrictions: None

Specifies whether the class driver uses the NI Spy utility to record calls to class driver functions.

The user specifies the value of `IVI_ATTR_SPY`.

The default value is `VI_FALSE`. If the user opens an instrument session by passing a logical name, the user can override the default value by using MAX to specify a value in the `ivi.ini` configuration file. Instrument drivers provide a *`Prefix`*`_InitWithOptions` function that users can call to override both the default value and any value specified in the `ivi.ini` configuration file.

If the user opens an instrument session through a specific driver, the IVI engine generates an `IVI_ERROR_NOT_CREATED_BY_CLASS` error whenever you attempt to set or get this attribute.

IVI_ATTR_VISA_RM_SESSION

Data Type: `ViSession`
Restrictions: Read-only, hidden from user

If a specific driver uses VISA instrument I/O, it passes the value of this attribute to the `viOpen` function during initialization. The `viOpen` function returns an instrument I/O session, which the driver stores in the `IVI_ATTR_IO_SESSION` attribute.

3

Programming Guidelines for Instrument Drivers

This chapter contains general procedures and guidelines for creating IVI drivers. If you write instrument drivers for general distribution, these guidelines help ensure that your driver behaves correctly, has a standard look and feel, and works on multiple platforms and operating systems.

The first part of this chapter describes how to use the Instrument Driver Development Wizard to generate instrument driver files. When you use the wizard with the built-in, predefined instrument driver templates, the wizard generates files in which most of the instrument driver design decisions have already been made and much of the coding has already been done for you.

The chapter also describes how to customize the driver for your particular instrument once you have generated the skeleton driver from the template. Each section shows you how to handle some common situations that you might encounter. The guidelines are well-suited for GPIB, VXI, and RS-232 instruments; however, you can apply this information to instruments that use other I/O interfaces.

The remainder of this chapter describes many of the details for developing instrument drivers, such as defining driver functions, attributes, and range tables.

General Guidelines

The following general guidelines help you to develop an instrument driver. Follow these guidelines whether you are developing instrument drivers for personal use or for general distribution to other users.

- Use the Instrument Driver Development Wizard to create your driver. Select **Tools» Create IVI Instrument Driver** in LabWindows/CVI to initiate the wizard. The wizard uses standard instrument templates for oscilloscopes, digital multimeters, function generators/arbitrary waveform generators, switches, and power supplies to define easy-to-use functions and attributes for these types of instruments. The wizard also allows you to base your instrument driver on an existing driver.
- If the wizard does not have a predefined template that fits your instrument type, you can still use the wizard to build a VXI*plug&play*-style driver. The wizard automatically generates skeleton versions of the instrument driver files and sets up the internal structure

of a driver for you. Before completing your driver, define the external interface to the driver. A useful instrument driver is more than a group of functions. It is a tool to help users develop application programs. Therefore, design an instrument driver with the user in mind.

- Use the Attribute Editor to add and modify attributes and to navigate through your source code. Select **Tools»Edit Instrument Attributes** to initiate the Attribute Editor.
- Follow the instructions in the *Writing an Instrument Driver* section. If the step refers you to another section for more information, read the information before you complete the task outlined in the step.

Writing an Instrument Driver

You can develop the pieces of an instrument driver in several different sequences. For more information about how to perform the individual steps in the procedure, refer to the *LabWindows/CVI Help*. You can use the Instrument Driver Development Wizard and the Attribute Editor to automate developing your instrument driver.

When you use the wizard to build a driver for an oscilloscope, digital multimeter, function generator, switch, or power supply, you can select a predefined template that defines common functions and attributes for these types of instruments. Refer to the *Selecting an Instrument Driver Template* section for more information. The wizard generates a function panel (`.fp`) file, a source (`.c`) file, an include (`.h`) file, and a `.sub` file for you.

To write the driver for your specific instrument, complete the following steps:

1. Name the instrument driver. Refer to the *Naming the Driver* section in this chapter for more information.
2. Define the attributes. The wizard automates this task when you use a template.
3. Define the instrument functions and function classes. The wizard automates this task when you use a template.
4. Create a function tree for the instrument driver, adding help information to the top level of the tree. The wizard automates this task when you use a template.
5. For each function in the driver, complete the following steps:
   a. Define the parameters to the function, including variable types and limits, and error codes. The wizard automates this task when you use a template.
   b. Create the function panel for the function. Include help information for the panel and for each control. The wizard automates this task when you use a template.
   c. Write the code to perform the function.
   d. Test the source code.

6. Create the include file for the final instrument source code, including function declarations and constant definitions. The wizard automates this task when you use a template.
7. Operate the completed driver using function panels.
8. Document the driver.

Naming the Driver

The instrument drivers you create join the large set of LabWindows/CVI instrument drivers. Give unique and meaningful names to the driver and its routines to avoid conflicts with the other instrument drivers and routines. Create both a descriptive name and assign an instrument prefix. Insert the prefix before each function name in the driver and use the prefix to name the component files (`.c`, `.h`, `.fp`, and so on) of the driver.

The descriptive name should be a short description of the driver. This name appears in the **Instrument** menu list. Examples of such descriptive names are Tektronix 3000 Series Oscilloscopes and HPE1411B Digital Multimeter.

Create the instrument prefix by using a two-character vendor code followed by characters that uniquely identify the driver. Vendors register their two-character codes with the VXI*plug&play* Alliance. The VXI*plug&play* specification, *VPP-9: Instrument Vendor Abbreviations*, contains a list of the two-character codes. For example, suppose you write an instrument driver for the Agilent 34401A digital multimeter. Since AG is the vendor code for Agilent, a unique prefix is ag34401a. The files that comprise the instrument driver would be `ag34401a.c`, `ag34401a.h`, `ag34401a.fp`, `ag34401a.sub`, and `ag34401a.doc`. Furthermore, the driver function names each have the prefix `ag34401a` added to them, for example, `ag34401a_ConfigureTrigger`. Since the prefix is appended to all functions, attribute IDs, and value definitions, the prefix should be relatively short.

Note The instrument prefix *must* have 31 characters or fewer. LabWindows/CVI adds an underscore (_) separator to the 31-character prefix before appending the function name to the prefix.

Using the Instrument Driver Development Wizard

The Instrument Driver Development Wizard automates the creation of the source, include, and function panel files for controlling an instrument. You create an instrument driver from an IVI instrument class template, an existing driver for a similar instrument, or the core IVI driver template.

Before using the wizard, complete the following worksheet with the appropriate information for your instrument. You need this information to complete the wizard.

Driver Information

Instrument Driver Name: _____

Prefix: _____ (two-character vendor code followed by non-whitespace characters that uniquely identify the driver)

Target Directory: _____

Developer Information

Name: _____

Company: _____

Phone: _____

Fax: _____

Standard Functions (if supported by the instrument)

Default Setup Command: _____

ID Query Command: _____ (`*IDN?` if it uses SCPI commands)

ID Query Response: _____

Reset Command: _____

Reset Delay: _____ (time required for reset to execute and return)

Self-Test Command: _____

Self-Test Response Format: _____ (self-test code and/or message)

Self-Test Delay: _____ (time to allow self-test to execute and return)

Error-Query Command: _____

Error-Query Response Format: _____ (error code and/or message)

Revision Query Command: _____

Revision Query Response Format: _____

Test Information

VISA Resource Descriptor: _____ (for example, `GPIB::5::INSTR`)

To launch the wizard, select **Tools»Create IVI Instrument Driver** in the LabWindows/CVI Project window. Follow the instructions on each panel of the wizard using the information from the worksheet. The number and types of panels that appear vary according to the selections you make.

Selecting an Instrument Driver Template

After you click **Next** on the initial wizard panel, you can copy an existing driver or create a new driver, as shown in Figure 3-1.

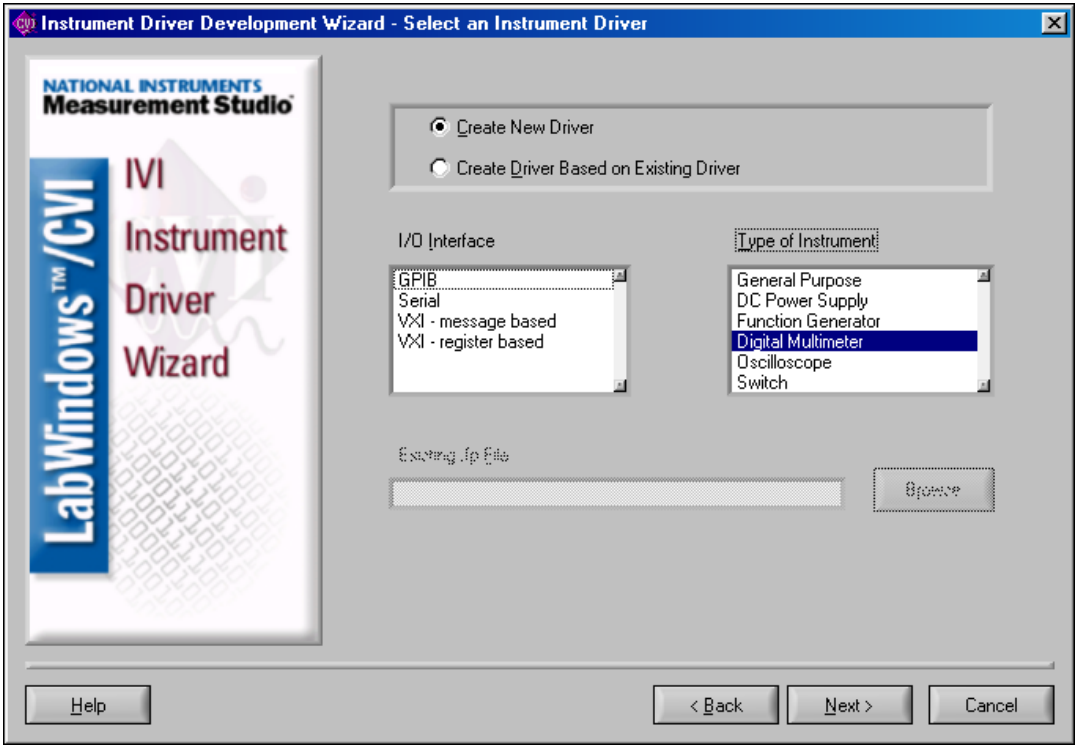

Figure 3-1. Instrument Driver Wizard Selection Panel

If you choose to copy an existing driver, you must specify the pathname of the `.fp` file for the existing driver. The wizard copies the `.fp`, `.c`, `.h`, and `.sub` files of the existing driver to the target directory that you specify later in the wizard. The wizard uses the instrument prefix that you specify later in the wizard as the new base filename.

The wizard builds new drivers based on predefined instrument templates. If you choose to create a new driver, you must first select the type of I/O interface you want to use to communicate with the instrument. You must then select from a list of predefined instrument templates. The list of templates can vary depending on the I/O interface you select.

Table 3-1 lists the templates that the Instrument Driver Development Wizard uses. Select a template that best matches the capabilities of your instrument. Most predefined instrument templates define a complete driver architecture with functions and attributes for a particular type of instrument or device. An exception is the General Purpose template. The General Purpose template appears in the list if you choose GPIB, Serial, VXI message based, or VXI register based as your I/O interface. When you select the General Purpose template, the wizard generates skeleton driver files that can use the IVI engine for managing attributes and that follow the basic VXI*plug&play* guidelines for instrument drivers. The skeleton driver files define no functions or attributes other than the ones that the IVI Foundation requires. Use the General Purpose template only when no predefined instrument templates apply to your instrument.

Table 3-1. Instrument Driver Class Templates

Template	Description
General Purpose	Use only for instrument types for which there is no class template. The template contains all the inherent functions that the IVI Foundation requires. It also has utility routines that implement typical low-level driver operations.
Digital Multimeter	Controls basic operations such as setting the measurement function, range, and resolution. It also includes advanced features such as configuring the trigger count and sample count and taking multipoint measurements.
Function Generator	Controls basic operations such as outputting standard waveforms. It also includes the ability to generate arbitrary waveforms and configure the modulation.
Oscilloscope	Controls basic operations such as acquiring waveforms using edge triggering and transferring waveform data from the instrument. It also includes features such as configuring advanced acquisition types and trigger modes and performing waveform measurements.
DC Power Supply	Controls basic operations such as outputting DC power and configuring the over-voltage and over-current protection. It also includes features such as monitoring the output voltage and current.
Switch	Controls basic channel connect and disconnect operations. It also includes features such as monitoring the output voltage and current.

Note The instrument class templates incorporate all the features of the general purpose template as well as the features in Table 3-1.

Running the Preliminary I/O Tests from the Wizard

If you select GPIB, Serial, or VXI message based as your I/O interface, you can run preliminary I/O tests to verify that the information you have provided is accurate before the wizard generates any code. Enter the appropriate information in the Test dialog box and click **Run Tests**. The wizard launches a separate application that sends commands to the instrument for ID query, self-test, reset, and so on, and parses the instrument response based on information you provide in the wizard. After the tests execute, the wizard generates a report that describes the success or failure of each operation. If any failures occur, you can click **Back** to return to the appropriate wizard panel, update the information, and try the tests again.

After the tests execute successfully, click **Next** to generate the `.fp`, `.c`, `.h`, and `.sub` files for your instrument driver.

After the wizard displays the newly created files, you can launch the Attribute Editor, which NI recommends you use to complete your instrument driver.

Reviewing the Generated Driver Files

The wizard generates all the required files for an instrument driver. If you use the General Purpose template, you must design a function hierarchy with function definitions and attributes on your own. You must build function panels and write the source code to implement these functions. Refer to the **Using LabWindows/CVI»Function Tree Editor**, **Function Panel Editor**, and **Adding Help Information** topics of the *LabWindows/CVI Help* for instructions on building function panels.

If you use a predefined instrument template to generate your driver files, your instrument driver files have predefined functions, function panels, and attributes. You must complete the source code by adding the appropriate command strings for your instrument and the code for parsing response strings.

Generated Function Panels

Your instrument driver function panel file displays all the instrument's capabilities in a function tree. The wizard builds all the function panels automatically and includes online help for the function classes, functions, and parameters.

.sub file

The `.sub` file contains information about instrument attributes and their possible values. This information appears to the user through the `GetAttribute` and `SetAttribute` function panels. To view this information, open one of the `SetAttribute` functions from the Configuration Functions class in your driver. Click the **Attribute** control to display the

Select Attribute Constant dialog box, as shown in Figure 3-2. You can select each attribute to view the possible values for that attribute in the lower half of the dialog box.

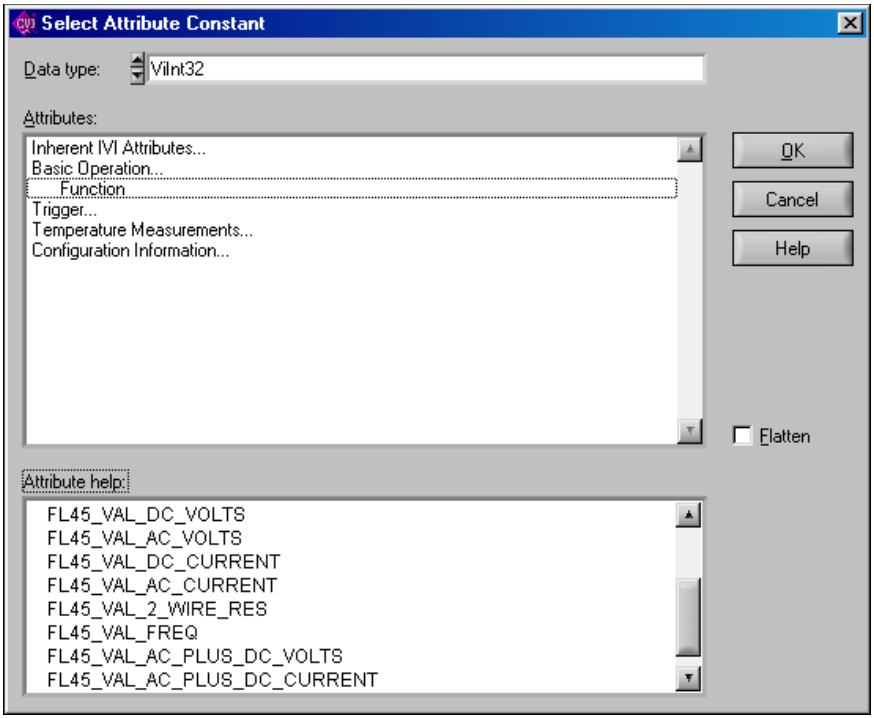

Figure 3-2. Select Attribute Constant Dialog Box

Where possible, the include file for the driver defines constants with intuitive names for each of the attributes and their possible values.

Add information to the `.sub` file by selecting **Tools»Edit Instrument Attributes**, which launches the Attribute Editor. When you apply the changes that you make in the Attribute Editor, the Attribute Editor updates the `.c`, `.h`, and `.sub` files for the instrument driver. For more information, refer to Chapter 4, *Attribute Editor*.

Source File

When you use the wizard with a predefined instrument template, the wizard groups the functions in the driver source file into the following categories:

- **Initialize Functions**—The wizard completes these functions for you automatically.
- **Configure Functions**—These functions set instrument attributes or groups of instrument attributes. For example, an oscilloscope has a `ConfigureVertical` function that sets the vertical range, offset, coupling, and probe attenuation for a particular channel. The

wizard completes the code for the Configure functions automatically. The code for each Configure function consists of calls to an `Ivi_SetAttribute` function for each configuration parameter.

- **Data Functions**—These functions query the instrument for data. You must complete the source code for these functions to send the appropriate query commands to the instrument, then parse and scale the data the instrument returns.
- **Action/Status Functions**—These functions either initiate an action on the instrument or check the status of a particular operation. An Action function might initiate an action by sending a trigger to the instrument. A Status function might check whether an acquisition is in progress for an oscilloscope. You must complete the source code for these functions.

Note Some predefined instrument templates, such as the DMM template, combine the Data Functions and Action/Status functions under the term Measurement Functions to present a more intuitive function hierarchy to the user.

- **Utility Functions**—The wizard completes these functions for you automatically.
- **Attribute Callback Functions**—These functions contain the code to read, write, or check attribute values. For example, the code to query and modify an oscilloscope's vertical offset setting must be in the `VerticalOffsetReadCallback` and `VerticalOffsetWriteCallback` functions. You must complete the source code for each of these functions by inserting the appropriate command to set or query the attribute value on your instrument.
- **Session Callback Functions**—These functions contain the code to perform various actions such as checking the status of the instrument. You must complete the source code for each of these functions.
- **Close Function**—The wizard completes this function for you automatically.

Include File

The include file contains function prototypes and defined constants for your instrument driver. The defined constants provide unique constant names for the attributes and attribute values your driver uses. Modify the include file only if you add, delete, or modify functions in your driver or if you add new attribute values.

Extended Functions and Attributes

The generated driver files can include functions and attributes that your instrument does not support. The instrument templates define the fundamental capabilities of each instrument type and extended capabilities that not all instruments support. For example, the DMM template includes functions and attributes for making multipoint measurements with scanning DMMs. If you are writing a driver for a DMM that does not support scanning, you must delete the functions and attributes from your driver.

Customizing Wizard-Generated Driver Files

After you create the instrument driver files, you can customize them to match the requirements of your instrument. To customize the instrument driver files, modify the existing attributes and functions, delete the attributes and functions that the instrument does not use, or add new attributes and functions.

Refer to the *Instrument Driver Attributes* and *User-Callable Functions* sections in this chapter for more information and guidelines regarding attributes and functions.

Attribute Editor

The Attribute Editor is a tool for viewing, modifying, and navigating through your instrument driver files. The Attribute Editor provides a visual display of the attributes in your driver. You can use the Attribute Editor to add, delete, or modify attributes. You can add or delete the callback function names for each attribute. You can modify the help information for attributes, and you can create or modify attribute range tables. Refer to Chapter 4, *Attribute Editor*, for information on how to use this tool to complete your instrument driver source code.

Modifying Existing Attributes and Functions

When you use a template or modify an existing driver, the majority of your effort is spent modifying the existing attributes and functions. If you develop your driver from a template, the code contains examples with instructions to help you customize the driver.

Complete the following steps to modify an attribute:

1. Edit the range table with the Range Tables dialog box in the Attribute Editor. Verify that the table represents the range of values your instrument accepts.
2. If you add new entries to the range table, complete the actual value and help text information.
3. If you delete range table entries that use defined constants that the driver does not otherwise reference, manually delete the constants from the header file.
4. Edit the attribute with the Attribute Editor. Verify that the attribute help information is accurate for your instrument.
5. Modify the implementation of the attribute callbacks.
6. Delete any modification instructions.
7. Test the attribute.

Complete the following steps to modify an instrument driver function:

1. Edit the function panel help.
2. Edit the function panel. Verify that all control help accurately describes your instrument.

3. Modify the function code to work with your instrument.
4. Delete any modification instructions.
5. Test the function.

Deleting the Attributes and Functions

In many cases, the template or existing driver contains attributes and functions that your instrument does not use. You can delete attributes and functions from your instrument driver.

Complete the following steps to delete an attribute:

1. Use the Range Tables dialog box in the Attribute Editor to delete the range tables for the attribute.
2. If you delete range table entries that use defined constants that the driver does not otherwise reference, manually delete the constants from the header file.
3. Use the Attribute Editor to delete the attribute callbacks and the defined constant for the attribute ID.
4. Apply the changes in the source file.

Caution *Never* manually remove range tables from the source file. Always use the Range Tables dialog box to delete range tables. Information regarding range tables resides in multiple instrument driver files. Manually removing range tables can cause the instrument driver to become unsynchronized.

Complete the following steps to delete an instrument driver function:

1. Delete the function prototype from the header file.
2. Delete the function from the source file.
3. Delete the function node from the function tree.

Adding New Attributes and Functions

Your instrument probably requires attributes or functions that the wizard does not create. You can add these attributes and functions to your instrument driver.

Complete the following steps to add a new attribute:

1. Create the attribute with the Attribute Editor.
2. Create a range table for the attribute from the Edit Attribute dialog box. Complete the actual value and help text information for each entry in the range table.
3. Edit the help for the attribute.
4. Place the new attribute in the appropriate position in the attribute hierarchy in the Edit Driver Attributes dialog box.

5. Apply the changes in the source.
6. Use the **Go To Callback Source** button to find the callback definitions in the source file. Create the function body for each callback.
7. Test the new attribute.

Complete the following steps to add a new instrument driver function:

1. Insert the new function in the appropriate position in the function tree.
2. Edit the function panel help.
3. Edit the function panel. Create all function panel controls and edit all control help.
4. Declare the new function in the instrument driver header file.
5. Insert the function code in the instrument driver source file.
6. Test the new function.

General Modifications

If you generate the driver files from a wizard template, comments at the beginning of the source file provide additional instructions for customizing the driver files. Read these comments and perform the corresponding modifications.

If you create the driver files from a class template, the source file also contains instructions for modifying the driver for that type of instrument.

Instrument Driver Attributes

This section contains additional explanations and guidelines for implementing instrument driver attributes. This section also contains extensive examples that illustrate common approaches for implementing attributes.

Attribute ID Values

Each attribute in your instrument driver must have a macro that defines the ID value for the attribute. Your instrument driver contains the following three types of attributes:

- Attributes that the IVI engine defines
- Attributes that the instrument class defines
- Attributes that only your instrument driver defines

Redefine the IVI engine and instrument class attributes using your instrument driver macro prefix. The following example shows how to redefine attribute IDs for IVI engine attributes. The example uses `FL45` as the macro prefix.

```
#define FL45_ATTR_RANGE_CHECK        IVI_ATTR_RANGE_CHECK
```

```
#define FL45_ATTR_QUERY_INSTR_STATUS IVI_ATTR_QUERY_INSTR_STATUS
#define FL45_ATTR_CACHE              IVI_ATTR_CACHE
```

The following example shows how to redefine attribute IDs for instrument class attributes.

```
#define FL45_ATTR_FUNCTION           IVIDMM_ATTR_FUNCTION
#define FL45_ATTR_RANGE              IVIDMM_ATTR_RANGE
```

The IVI engine and the instrument class headers have unique values for the attributes IDs that they define. Each new attribute that your instrument driver creates must have a unique ID value. The IVI engine header file defines attribute ID bases for this purpose. Use the `IVI_SPECIFIC_PUBLIC_ATTR_BASE` macro to define public attributes in the instrument driver header file. The following example shows how to define public attribute ID values.

```
#define FL45_ATTR_ID_QUERY_RESPONSE \
                                (IVI_SPECIFIC_PUBLIC_ATTR_BASE + 0L)
#define FL45_ATTR_HOLD_THRESHOLD    \
                                (IVI_SPECIFIC_PUBLIC_ATTR_BASE + 1L)
#define FL45_ATTR_HOLD_ENABLE       \
                                (IVI_SPECIFIC_PUBLIC_ATTR_BASE + 2L)
```

The header file defines each attribute ID value as `IVI_SPECIFIC_PUBLIC_ATTR_BASE` plus an offset. For each public attribute that you create, increment the offset.

Use the `IVI_SPECIFIC_PRIVATE_ATTR_BASE` macro to define the ID values for hidden attributes. Place the ID definitions for hidden attributes in the instrument driver source file. The following example shows how to define private attribute ID values.

```
#define FL45_ATTR_OPC_TIMEOUT       \
                                (IVI_SPECIFIC_PRIVATE_ATTR_BASE + 1L)
```

The source file defines each attribute ID value as `IVI_SPECIFIC_PRIVATE_ATTR_BASE` plus an offset. For each hidden attribute you create, increment the offset.

Attribute Value Definitions

Define values for public attributes in your instrument driver header file. Typically, the instrument class defines values for the attributes that it defines. Redefine the class values using the macro prefix for your instrument driver. The following example shows how to redefine class attribute values for use with your instrument driver. The example uses `FL45` as the macro prefix.

```
#define FL45_VAL_DC_VOLTS            IVIDMM_VAL_DC_VOLTS
#define FL45_VAL_AC_VOLTS            IVIDMM_VAL_AC_VOLTS
#define FL45_VAL_DC_CURRENT          IVIDMM_VAL_DC_CURRENT
#define FL45_VAL_AC_CURRENT          IVIDMM_VAL_AC_CURRENT
```

If you add additional values that the instrument class does not define for an attribute, ensure that the values are unique. Where possible, the instrument class defines extended value bases for each attribute. You can add new attribute values starting at these bases. The following example shows how to add instrument-specific values for the measurement function attribute of a DMM.

```
#define FL45_VAL_NEW_FUNCTION    \
                            (IVIDMM_VAL_FUNC_SPECIFIC_EXT_BASE + 1)
```

The example adds an offset to the `IVIDMM_VAL_FUNC_SPECIFIC_EXT_BASE` macro to create a unique attribute value. For each new value that you add, increment the offset.

Simulation

When the user enables simulation, your driver must not perform any instrument I/O. For attributes, you typically perform instrument I/O only in the read and write callbacks. By default, the IVI engine does not invoke the read and write callbacks during simulation. Therefore, you generally do not have to worry about simulation when you implement the callbacks for your attributes. However, you must prevent instrument I/O in attribute callbacks when simulating in the following situations:

- You perform instrument I/O in a callback other than the read or write callback. This condition does not include calling the set and get attribute functions.
- You perform instrument I/O in a read or write callback and the `IVI_VAL_USE_CALLBACKS_FOR_SIMULATION` flag is set for the attribute.

The following example shows how to use the `Ivi_Simulating` function to determine whether to simulate instrument I/O.

```
if (!Ivi_Simulating(vi)) /* call only when the session is locked */
    {
        /* Perform instrument I/O here */
    }
```

Data Types

Use the IVI engine to create attributes with only a subset of the data types you can use with instrument drivers. Table 3-2 shows what data types you can use for attributes.

Table 3-2. Data Types You Can Use for Attributes

Attribute Access	Data Type

Public and hidden attributes	`ViInt32`
	`ViReal64`
	`ViString`
	`ViBoolean`
	`ViSession`

Hidden attributes only	`ViAddr`

Note Create `ViAddr` attributes only for internal use within the instrument driver.

Callbacks

This section describes special requirements to consider when you implement the attribute callbacks.

Read and Coerce Callbacks for ViString Attributes

In general, the read and coerce callbacks have a reference parameter in which they return the result of the operation to the IVI engine. Use an alternative mechanism in read and coerce callbacks for `ViString` attributes. For these attributes, use the `Ivi_SetValInStringCallback` function to return the string that the callback reads or coerces.

The following example shows how to return a string to the IVI engine from the read callback for a `ViString` attribute.

```
static ViStatus _VI_FUNC exampleAttrIdQueryResponse_ReadCallback
                          (ViSession vi, ViSession io,
                          ViConstString channelName,
                          ViAttr attributeId,
                          const ViConstString cacheValue)
{
   ViStatus error = VI_SUCCESS;
   ViChar   rdBuffer[BUFFER_SIZE];
   ViUInt32 retCnt;

   viCheckErr( viPrintf (io, "*IDN?"));
   viCheckErr( viRead (io, rdBuffer, BUFFER_SIZE-1, &retCnt));
   rdBuffer[retCnt] = 0;
```

```
    checkErr( Ivi_SetValInStringCallback (vi, attributeId, rdBuffer));
Error:
    return error;
}
```

Use the same technique to return the result of the coerce callback to the IVI engine for `ViString` attributes.

Write Callbacks

Write callbacks receive the new setting for the attribute in the **value** input parameter. You can assume that the IVI engine has already checked and coerced the value. Thus, the write callback needs only to write the value to the instrument.

For message-based instruments, the write callback must build complete and independent command strings. Each command string must be valid regardless of any other command strings the driver might send beforehand or afterwards. For the instrument to interpret the command string correctly, you must begin all commands with the complete header information and separate the individual commands with the appropriate termination character.

Reading Strings From the Instrument

The following two examples show the proper techniques for reading data into string variables when you use the `viScanf` and `viRead` functions. NI recommends using the `viScanf` and `viPrintf` functions instead of the `viRead` and `viWrite` functions for most instrument I/O.

Using viScanf

You typically use the `viScanf` function to read data from an instrument and parse the data in one step. When you use the `viScanf` function to read data from the instrument into a string variable, you must guard against writing past the end of the string variable. Use the following technique to read data from the instrument and place it in a string variable.

```
ViChar     rdBuffer[BUFFER_SIZE];
ViInt32    rdBufferSize = sizeof(rdBuffer);

viCheckErr( viPrintf (io, "FUNC?;"));
viCheckErr( viScanf (io, "FUNC %#s", &rdBufferSize, rdBuffer));

checkErr( Ivi_GetViInt32EntryFromString (rdBuffer,
                          &attrFunctionRangeTable, value, VI_NULL,
                          VI_NULL, VI_NULL, VI_NULL));
```

The `viScanf` statement in the example shows how to use the `#` modifier. The format string instructs VISA to place the data that follows the literal `FUNC` in the **rdBuffer** string variable. The `#` modifier specifies the size of the buffer into which the `viScanf` function places the data. When VISA parses the `#` modifier, it consumes the first parameter that follows the

format string. This parameter is the address of a variable that contains the size of the **rdBuffer**. The example shows how to declare and initialize the **rdBufferSize** variable. After the `viScanf` function executes, the **rdBufferSize** variable contains the number of bytes, including the ASCII NUL byte, that the `viScanf` function wrote into the **rdBuffer** variable.

Drivers that the Instrument Driver Development Wizard create contain the `BUFFER_SIZE` macro, which has the value 512. The example illustrates how you can use this macro to declare the number of bytes in a local string variable. See the *NI-VISA Programmer Reference Help* for further details on formatting specifiers used by the `viScanf` function.

Using viRead

If you use the `viRead` function to read character data, you must account for the fact that the `ViRead` function does not null-terminate character data. You must null-terminate the character data explicitly.

Often, instruments return a carriage return or a line feed at the end of strings. Functions such as `Scan` or `Ivi_GetViInt32EntryFromString` use these characters for termination. However, the templates do not assume that instruments have this behavior.

The following example illustrates how to use the `viRead` function.

```
ViChar      rdBuffer[BUFFER_SIZE];
ViUInt32    retCnt;

viCheckErr( viWrite (io, "*IDN?", 5, VI_NULL));
viCheckErr( viRead (io, rdBuffer, BUFFER_SIZE-1, &retCnt));
rdBuffer[retCnt] = 0;

checkErr( Ivi_SetValInStringCallback (vi, attributeId, rdBuffer));
```

The example declares a **rdBuffer** character array with a size of `BUFFER_SIZE`. The example passes `BUFFER_SIZE-1` to the `viRead` function as the maximum number of bytes to store in **rdBuffer**. The `viRead` function terminates when it reads `BUFFER_SIZE-1` bytes or encounters a termination character. The example uses `BUFFER_SIZE-1` so that if the maximum number of bytes are read, enough room remains in the array to append the ASCII NUL byte. After the function executes, the **retCnt** variable contains the number of bytes the `viRead` function stored in the **rdBuffer** character array. The next line shows how to use the **retCnt** variable to null-terminate the string in **rdBuffer**.

Caution *Never* pass a buffer that is not null-terminated to `Ivi_SetValInStringCallback`.

Range Table Callbacks

Range table callbacks return the address of a range table in the **rangeTablePtr** reference parameter. These callbacks must guard against returning bad range table addresses to the IVI engine. If a range table callback cannot determine which range table to use or encounters an error, the callback must return `VI_NULL` as the range table address. The following example shows how to structure the shell of a range table callback.

```
static ViStatus _VI_FUNC exampleAttrRange_RangeTableCallback
                          (ViSession vi, ViConstString channelName,
                          ViAttr attributeId,
                          IviRangeTablePtr *rangeTablePtr)
{
   ViStatus         error = VI_SUCCESS;
   IviRangeTablePtr tblPtr = VI_NULL;
   ViInt32          function;

   /*
      NOTE: Insert code here to select the correct range table. Set
      the tblPtr local variable to the address of the range table.
   */
Error:
   *rangeTablePtr = tblPtr;
   return error;
}
```

The example creates a local `IviRangeTablePtr` variable called **tblPtr** and initializes it to `VI_NULL`. The comment shows where to insert the code that determines the correct range table and sets the value of the **tblPtr** variable. In the error block, the example shows how to set the **rangeTablePtr** output parameter to the address that the **tblPtr** variable contains. If the example encounters an error before the **tblPtr** variable is set, this approach guarantees that the **rangeTablePtr** reference parameter returns `VI_NULL`. Otherwise, the function returns the appropriate range table address.

Range Tables

Create range tables to describe the possible values for an attribute. You typically use static range tables for this purpose. If the user initializes multiple instruments with the driver, each IVI session shares the static range tables. Therefore, the driver must not programmatically change the values of a static range table because doing so changes the range table for all sessions.

If you must modify a range table programmatically, you must dynamically allocate the range table with the `Ivi_RangeTableNew` function and then store the range table address with the IVI session. The *Attributes with a Changing Valid Range* section in this chapter shows an example of this technique.

Attribute Examples

The section shows techniques you can use to implement source code for your instrument driver attributes. The section covers three common attribute types. For each attribute type, an example shows how to implement the range table and the write and read callbacks.

Attributes that Represent Discrete Settings

A typical type of attribute is one that represents a set of discrete instrument settings. The attribute that controls the measurement function of a DMM is an example of this type of attribute. The following example shows a range table for a measurement function attribute.

```
static IviRangeTableEntry attrFunctionRangeTableEntries[] =
    {
        {EXAMPLE_VAL_DC_VOLTS,   0, 0, "DCV CMD",   0},
        {EXAMPLE_VAL_AC_VOLTS,   0, 0, "ACV CMD",   0},
        {EXAMPLE_VAL_DC_CURRENT, 0, 0, "DCA CMD",   0},
        {EXAMPLE_VAL_AC_CURRENT, 0, 0, "ACA CMD",   0},
        {EXAMPLE_VAL_2_WIRE_RES, 0, 0, "2WRES CMD", 0},
        {IVI_RANGE_TABLE_LAST_ENTRY}
    };

static IviRangeTable attrFunctionRangeTable =
    {
        IVI_VAL_DISCRETE,
        VI_FALSE,
        VI_FALSE,
        VI_NULL,
        attrFunctionRangeTableEntries
    };
```

The range table type is `IVI_VAL_DISCRETE`. The range table defines the possible discrete settings and the corresponding command strings for the attribute. The following example shows a typical write callback that uses the discrete range table.

```
static ViStatus _VI_FUNC exampleAttrFunction_WriteCallback
                         (ViSession vi, ViSession io,
                         ViConstString channelName,
                         ViAttr attributeId, ViInt32 value)
{
    ViStatus error = VI_SUCCESS;
    ViString cmd;

    checkErr( Ivi_GetViInt32EntryFromValue (value,
                         &attrFunctionRangeTable, VI_NULL, VI_NULL,
                         VI_NULL, VI_NULL, &cmd, VI_NULL));
    viCheckErr( viPrintf (io, ":FUNC %s;", cmd));
```

```
Error:
   return error;
}
```

The write callback assumes that the value it receives in the **value** parameter is valid. The write callback for the attribute performs the following operations:

1. Uses the `Ivi_GetViInt32EntryFromValue` function to search the range table for the command string that corresponds to the value it receives in the **value** parameter.
2. Formats and writes the command string to the instrument.

The example shows how to build the complete command string from a command header (`:FUNC`), the command string found in the range table, and a termination character (`;`).

The following example shows a read callback for the attribute.

```
static ViStatus _VI_FUNC exampleAttrFunction_ReadCallback
                          (ViSession vi, ViSession io,
                          ViConstString channelName,
                          ViAttr attributeId, ViInt32 *value)
{
   ViStatus error = VI_SUCCESS;
   ViChar   rdBuffer[BUFFER_SIZE];
   ViInt32  rdBufferSize = sizeof(rdBuffer);

   viCheckErr( viPrintf (io, "FUNC?;"));
   viCheckErr( viScanf (io, "FUNC %#s", &rdBufferSize, rdBuffer));

   checkErr( Ivi_GetViInt32EntryFromString (rdBuffer,
                          &attrFunctionRangeTable, value, VI_NULL,
                          VI_NULL, VI_NULL, VI_NULL));

Error:
   return error;
}
```

The read callback performs the following operations:

1. Sends a command string (`FUNC?`) that instructs the instrument to return the current measurement function setting.
2. Parses the response, discarding any header information.
3. Finds the corresponding value in the range table and sets the **value** output parameter.

You can structure the read and the write callbacks so that they use the same range table.

Attributes with discrete settings are often `ViInt32` attributes. You can use this technique for `ViReal64` attributes as well. However, `ViReal64` attributes usually represent a continuous range of instrument settings or a continuous range that the instrument coerces to a group of

discrete settings. Refer to the *Attributes that Represent a Continuous Range* and *Attributes that Represent a Continuous Range with Discrete Settings* sections in this chapter for more information on these types of attributes.

Attributes that Represent a Continuous Range

Another common type of attribute represents a continuous range of valid instrument settings. The data type for this kind of attribute is usually `ViReal64` but also can be `ViInt32`. The attribute that controls the trigger delay of a DMM is an example of this type of attribute.

The following example shows a range table for an attribute with a continuous range.

```
static IviRangeTableEntry attrTriggerDelayRangeTableEntries[] =
    {
        {0.0, 10.0, 0, "", 0},
        {IVI_RANGE_TABLE_LAST_ENTRY}
        };
static IviRangeTable attrTriggerDelayRangeTable =
    {
        IVI_VAL_RANGED,
        VI_TRUE,
        VI_TRUE,
        VI_NULL,
        attrTriggerDelayRangeTableEntries
    };
```

The range table type is `IVI_VAL_RANGED`. The range table defines the range of values the instrument accepts for the attribute. The following example shows the write callback.

```
static ViStatus _VI_FUNC exampleAttrTriggerDelay_WriteCallback
                          (ViSession vi, ViSession io,
                          ViConstString channelName,
                          ViAttr attributeId, ViReal64 value)
{
    ViStatus error = VI_SUCCESS;
    viCheckErr (viPrintf (io, ":TRIG:DEL %Lf;", value));
Error:
    return error;
}
```

The write callback assumes that the value it receives in the **value** parameter is valid. The write callback uses this value to format and write the command string to the instrument. The example shows how to build the complete command string from a command header (`:TRIG:DEL`), the value held in the **value** parameter, and a termination character (`;`).

The following example shows the read callback.

```
static ViStatus _VI_FUNC exampleAttrTriggerDelay_ReadCallback
                          (ViSession vi, ViSession io,
                          ViConstString channelName,
                          ViAttr attributeId, ViReal64 *value)
{
   ViStatus error = VI_SUCCESS;

   viCheckErr( viPrintf (io, "TRIG:DEL?"));
   viCheckErr( viScanf  (io, "%Lf", value));

Error:
   return error;
}
```

The read callback performs the following operations:

1. Sends a command string (`TRIG:DEL?`) that instructs the instrument to return the current trigger delay setting.
2. Parses the response and sets the **value** output parameter.

Attributes that Represent a Continuous Range with Discrete Settings

Another common attribute type represents a continuous range, but the instrument only uses a set of discrete values that are within the range. Instruments can implement these attributes in two ways.

- The instrument accepts only the discrete values that fall within the range.
- The instrument accepts any value within the range but coerces the value internally.

For either case, this example shows how to implement this type of attribute in your driver.

An example of this type of attribute is the measurement resolution attribute for a DMM. The example below shows a coerced range table.

```
static IviRangeTableEntry attrResolutionRangeTableEntries[] =
   {
      {0.0, 3.5, 3.5, "LOW",  0},
      {3.5, 4.5, 4.5, "MID",  0},
      {4.5, 5.5, 5.5, "HIGH", 0},
      {IVI_RANGE_TABLE_LAST_ENTRY}
   };

static IviRangeTable attrResolutionRangeTable =
   {
      IVI_VAL_COERCED,
      VI_TRUE,
```

```
        VI_TRUE,
        VI_NULL,
        attrResolutionRangeTableEntries
    };
```

The range table type is `IVI_VAL_COERCED`. The range table defines the possible ranges, the coerced values that the instrument uses for each range, and the corresponding response strings. The following example shows the write callback.

```
static ViStatus _VI_FUNC exampleAttrResolution_WriteCallback
                          (ViSession vi, ViSession io,
                          ViConstString channelName,
                          ViAttr attributeId, ViReal64 value)
{
    ViStatus error = VI_SUCCESS;

    viCheckErr (viPrintf (io, ":RES %Lf;", value));

Error:

    return error;
}
```

The write callback assumes that the value it receives in the **value** parameter is valid and is a coerced value that the instrument accepts. The write callback uses this value to format and write the command string to the instrument. This example shows how to build the complete command string from a command header (`:RES`), the value held in the **value** parameter, and a termination character (`;`).

The following example shows the read callback.

```
static ViStatus _VI_FUNC exampleAttrResolution_ReadCallback
                          (ViSession vi, ViSession io,
                          ViConstString channelName,
                          ViAttr attributeId, ViReal64 *value)
{
    ViStatus error = VI_SUCCESS;
    ViChar   rdBuffer[BUFFER_SIZE];
    ViInt32  rdBufferSize = sizeof(rdBuffer);

    viCheckErr (viPrintf (io, ":RES?;"));
    viCheckErr (viScanf (io, "%#s", &rdBufferSize, rdBuffer));

    checkErr( Ivi_GetViReal64EntryFromString (rdBuffer,
                          &attrResolutionRangeTable, value, VI_NULL,
                          VI_NULL, VI_NULL, VI_NULL));

Error:
    return error;
}
```

The read callback performs the following operations:

1. Sends a command string (`RES?`) that instructs the instrument to return the current resolution setting.
2. Reads the response.
3. Finds the corresponding value in the range table and sets **value** output parameter.

This example shows how to use a range table to convert the instrument response to a corresponding value that the function can return. In many cases, the function can return the actual value that the instrument sends. In this case, you can modify the `viScanf` statement to read the string, parse the response, and set the **value** output parameter in one step. The following example shows an alternative way to use the `viScanf` function.

```
viCheckErr( viScanf (io, "%Lf", value));
```

Attributes with a Changing Valid Range

The previous three examples use a single static range table to describe the valid values for the attribute. In many cases, the valid value for an attribute depends on other instrument settings. For these cases, a single range table is not adequate. There are two approaches for this kind of attribute.

- Use multiple static range tables.
- Use a dynamic range table.

A major advantage of these approaches is that the IVI engine still performs the checking and coercion operations for the attribute. In both of these approaches, the driver installs a range table callback, and the IVI engine uses the range table callback to obtain the correct range table.

In some cases, it is not possible to describe the valid values and coercion rules for an attribute with range tables. Refer to the *Check, Coerce, and Compare Callbacks* section in this chapter for information on how to structure check and coerce callbacks.

Using Multiple Static Range Tables

If the number of static range tables necessary to describe the valid values for the attribute is fairly small, complete the following steps:

1. Create a static range table for each configuration.
2. Install a range table callback that selects the correct range table for the current configuration.

The measurement range attribute of a DMM is a common example of this type of attribute. The valid values for the measurement range attribute often depend on the current setting of the measurement function attribute. The following example shows typical range tables for the

measurement range attribute. For simplicity, this example shows the range tables that correspond only to the volts DC and resistance settings for the measurement function.

```
static IviRangeTableEntry VDCRangeTableEntries[] =
    {
        {  0.0,    0.1,    0.1, "R1", 0},
        {  0.1,    1.0,    1.0, "R2", 0},
        {  1.0,   10.0,   10.0, "R3", 0},
        { 10.0,  100.0,  100.0, "R4", 0},
        {100.0, 1000.0, 1000.0, "R5", 0},
        {IVI_RANGE_TABLE_LAST_ENTRY}
    };

static IviRangeTable VDCRangeTable =
    {
        IVI_VAL_COERCED,
        VI_TRUE,
        VI_TRUE,
        VI_NULL,
        VDCRangeTableEntries,
    };

static IviRangeTableEntry ohmsRangeTableEntries[] =
    {
        {     0.0,     100.0,     100.0, "R1", 0},
        {   100.0,    1000.0,    1000.0, "R2", 0},
        {  1000.0,   10000.0,   10000.0, "R3", 0},
        { 10000.0,  100000.0,  100000.0, "R4", 0},
        {100000.0, 1000000.0, 1000000.0, "R5", 0},
        {IVI_RANGE_TABLE_LAST_ENTRY}
    };

static IviRangeTable ohmsRangeTable =
    {
        IVI_VAL_COERCED,
        VI_TRUE,
        VI_TRUE,
        VI_NULL,
        ohmsRangeTableEntries,
    };
```

The driver uses the `VDCRangeTable` when the measurement function is set to volts DC and the `ohmsRangeTable` when the measurement function is set to resistance. The following example shows a range table callback that selects the correct range table.

```
static ViStatus _VI_FUNC exampleAttrRange_RangeTableCallback
                          (ViSession vi, ViConstString channelName,
                          ViAttr attributeId,
                          IviRangeTablePtr *rangeTablePtr)
{
   ViStatus         error = VI_SUCCESS;
   IviRangeTablePtr tblPtr = VI_NULL;
   ViInt32          function;

   checkErr( Ivi_GetAttributeViInt32 (vi, VI_NULL,
                          EXAMPLE_ATTR_FUNCTION, 0, &function));

   switch (function)
      {
      case EXAMPLE_VAL_DC_VOLTS:
         tblPtr = &VDCRangeTable;
         break;
      case EXAMPLE_VAL_2_WIRE_RES:
         tblPtr = &ohmsRangeTable;
         break;
      default:
         viCheckErr (IVI_ERROR_INVALID_CONFIGURATION);
         break;
}

Error:
   *rangeTablePtr = tblPtr;
   return error;
}
```

The range table callback performs the following operations:

1. Gets the current value of the measurement function attribute.
2. Uses a `switch` statement to select the correct range table.
3. Sets the **rangeTablePtr** output parameter to the address of the range table.

Install the range table callback for the attribute after you create the attribute with the `Ivi_AddAttribute` function as follows:

```
checkErr( Ivi_AddAttributeViReal64 (vi, EXAMPLE_ATTR_RANGE,
                          "EXAMPLE_ATTR_RANGE", 1.0, 0,
                          exampleAttrRange_ReadCallback,
                          exampleAttrRange_WriteCallback, VI_NULL,
                          0));
```

```
checkErr( Ivi_SetAttrRangeTableCallback (vi, EXAMPLE_ATTR_RANGE,
                          exampleAttrRange_RangeTableCallback));
```

The following example illustrates how to get the correct range table in a write callback by declaring a variable of type `IviRangeTablePtr` and passing its address to the `Ivi_GetAttrRangeTable` function.

```
static ViStatus _VI_FUNC exampleAttrRange_WriteCallback
                          (ViSession vi, ViSession io,
                          ViConstString channelName,
                          ViAttr attributeId, ViReal64 value)
{
   ViStatus         error = VI_SUCCESS;
   ViString         cmd;
   IviRangeTablePtr rangeTablePtr;

   checkErr( Ivi_GetAttrRangeTable (vi, "", EXAMPLE_ATTR_RANGE,
                          &rangeTablePtr));

   checkErr( Ivi_GetViInt32EntryFromValue (value, rangeTablePtr,
                          VI_NULL, VI_NULL, VI_NULL, VI_NULL, &cmd,
                          VI_NULL));

   viCheckErr (viPrintf (io, ":RANG: %s;", cmd));

Error:
   return error;
}
```

Use the same technique to get the range table in the other callbacks for the attribute.

Using Dynamic Range Tables

If the number of range tables is excessively large or if the values in the range table are the result of a calculation, complete the following steps:

1. Dynamically allocate a range table with the `Ivi_RangeTableNew` function.
2. Pass the range table pointer to the `Ivi_AddAttribute` function call that creates the attribute.
3. Install a range table callback for the attribute. The callback gets the address of the dynamic range table with the `Ivi_GetStoredRangeTablePtr` function, sets the values in the range table for the current instrument configuration, and returns the range table address in the **rangeTablePtr** output parameter.

The following example shows how to create a range table dynamically, pass the address of the range table to the `Ivi_AddAttribute` function, and install a range table callback.

```
IviRangeTablePtr tablePtr = VI_NULL;

checkErr( Ivi_RangeTableNew (vi, 10, 1, VI_TRUE, VI_TRUE, &tablePtr));

checkErr( Ivi_AddAttributeViReal64 (vi,
       EXAMPLE_ATTR_DYNAMIC_RANGE_TABLE,
       "EXAMPLE_ATTR_DYNAMIC_RANGE_TABLE", 0, 0,
       exampleAttrDynamicRangeTable_ReadCallback,
       exampleAttrDynamicRangeTable_WriteCallback, tablePtr, 0));

checkErr( Ivi_SetAttrRangeTableCallback (vi,
       EXAMPLE_ATTR_DYNAMIC_RANGE_TABLE,
       exampleAttrDynamicRangeTable_RangeTableCallback));
```

The following example shows how to manipulate dynamic range tables in a range table callback.

```
static ViStatus _VI_FUNC
             exampleAttrDynamicRangeTable_RangeTableCallback
             (ViSession vi, ViConstString channelName,
             ViAttr attributeId, IviRangeTablePtr *rangeTablePtr)
{
   ViStatus         error = VI_SUCCESS;
   IviRangeTablePtr tblPtr = VI_NULL;

   viCheckErr( Ivi_GetStoredRangeTablePtr (vi,
                        EXAMPLE_ATTR_DYNAMIC_RANGE_TABLE,
                        &tblPtr));

   /* Set the values in the range table here. */

Error:
   *rangeTablePtr = tblPtr;
   return error;
}
```

The range table callback performs the following operations:

1. Gets the range table address you pass to the `Ivi_AddAttribute` function for the attribute.
2. Sets the entries in the range table that are appropriate for the current configuration.
3. Returns the address of the range table in the **rangeTablePtr** output parameter.

Check, Coerce, and Compare Callbacks

The IVI engine supplies default callbacks that perform the basic check, coerce, and compare operations for attributes. The default check and coerce callbacks use the range table or range table callback the driver installs for an attribute to check and coerce values for the attribute. The default compare callback uses the comparison precision the driver passes in the `Ivi_AddAttributeViReal64` function to perform the compare operation.

In some cases, it is not possible to describe the checking, coercion, or comparison rules for an attribute by using a range table or a comparison precision value. Usually, the driver must perform operations *in addition* to those that the default check, compare, or coerce callbacks perform. In these cases, you can take the approach that the following example illustrates.

```
static ViStatus _VI_FUNC exampleAttrRange_CheckCallback
                         (ViSession vi, ViConstString channelName,
                         ViAttr attributeId, ViReal64 value)
{
   ViStatus    error = VI_SUCCESS;

   /* Perform additional checking here */

   checkErr( Ivi_DefaultCheckCallbackViReal64 (vi, channelName,
                         attributeId, value));

Error:
   return error;
}
```

The callback first performs the additional operations and then calls the appropriate default callback in the IVI engine. Table 3-3 shows the attribute callbacks for which you can use this approach.

Table 3-3. Attribute Callbacks that Can Call the Appropriate Default Callback

Attribute Data Type	Default Callbacks Supported
`ViInt32`	check callback coerce callback
`ViReal64`	check callback coerce callback compare callback
`ViBoolean`	coerce callback

User-Callable Functions

Add user-callable functions to your instrument driver to control the instrument operations that you want to make available to users. All user-callable functions have a function panel interface and return error and status information. The functions that you develop can merely manipulate your instrument driver attributes, or they can perform instrument I/O directly. This section gives explanations and guidelines for implementing user-callable functions in your instrument driver.

Instrument Driver Function Structure

When you develop user-callable functions, you must develop those functions to be consistent with the state-caching, simulating, multithread safety, and error handling features of other drivers in the National Instruments IVI driver library. To make developing user-callable functions easier, LabWindows/CVI defines a standard function structure. The following example illustrates the general structure of a user-callable function.

```
/*****************************************************************
 * Function:   Fl45_StdFunction
 * Purpose:    This function shows a standard approach for error
 *             handling.
*****************************************************************/
ViStatus _VI_FUNC Fl45_ StdFunction (ViSession vi)
    {
    ViStatus error = VI_SUCCESS;

    /* Perform instrument function here */

Error:
    return error;
    }
```

This standard function includes the following features:

- The function declares a local variable with the name **error** and a type of `ViStatus`. The declaration initializes the variable to `VI_SUCCESS`. The function records error and status information in the variable.

Caution The function contains a label named `Error:`. When the function encounters an error, the function sets the **error** variable and jumps to the `Error` label. The code following the `Error` label is called the `Error` block.

- The last line of code in the function is a `return` statement that returns the value of the **error** variable. The function must have no other `return` statements.

This structure encourages a consistent clean-up and error handling strategy for the function. The function localizes all cleanup and error handling in the `Error` block. If the function encounters an error, it jumps to the `Error` block, handles the error, and performs any necessary clean-up operations such as freeing temporary data that you dynamically allocate. If the function does not encounter an error, the program flows through the `Error` block and still performs the clean-up operations.

The IVI engine defines a set of error macros that you use when you implement your instrument driver functions. These macros help you determine when an error occurs, set the appropriate error information, and jump to the `Error` block. Refer to the *LabWindows/CVI Help* for detailed information on how to use the error macros.

You fill in the standard function structure with the code that performs the operation for your instrument. You can divide the function code you write into the following steps:

1. Lock the instrument driver session.
2. Check parameters.
3. Set or get attribute values.
4. Perform instrument I/O if you are not simulating.
5. Create and return simulated data if the function has output parameters and you are simulating.
6. Check the instrument status.
7. Unlock the session.
8. Return the value of the `error` variable.

The following example illustrates all the steps just listed.

```
/*****************************************************************
 * Function:   Fl45_Measure
 * Purpose:    This function sets the measurement function and reads
 *             a measurement.
*****************************************************************/

ViStatus _VI_FUNC Fl45_Measure (ViSession vi, ViInt32 function,
                            ViReal64 *reading)
{
   ViStatus  error = VI_SUCCESS;
   ViInt32   retCnt;

   checkErr( Ivi_LockSession (vi, VI_NULL));

   if (reading == VI_NULL)

      viCheckParm( IVI_ERROR_INVALID_PARAMETER, 3,
                            "Null address for Reading.");
```

```
    viCheckParm( Ivi_SetAttributeViInt32 (vi, FL45_ATTR_FUNCTION,
                                function),2, "Function");
    if (!Ivi_Simulating(vi))
        {
        /* perform io */
        ViSession io = Ivi_IOSession();
        checkErr( Ivi_SetNeedToCheckStatus (vi, VI_TRUE));
            viCheckErr( viPrintf (io, "VAL1?;"));
            error = viScanf (io, "%lf", &reading);
            if (error == VI_ERROR_TMO)
                error = FL45_ERROR_MAX_TIME_EXCEEDED;
            viCheckErr(error);
        }
    else
        {
        /* simulate output parameters */
        *reading = rand ();
        }
    checkErr( Fl45_CheckStatus (vi));
Error:
    Ivi_UnlockSession(vi, VI_NULL);
    return error;
}
```

Note This example is not the actual measure function from the Fluke 45 instrument driver. It has been modified to illustrate all the possible features of a user-callable function.

The `Fl45_Measure` example performs the following operations:

1. Locks the IVI session.
2. Verifies that the **reading** output parameter address is non-NULL.
3. Sets the `FL45_ATTR_FUNCTION` attribute to the value of **function**.
4. Performs instrument I/O if the `Ivi_Simulating` function returns `VI_FALSE`.
5. Creates simulated data for the **reading** output parameter if the `Ivi_Simulating` function returns `VI_TRUE`.
6. Checks the instrument status.
7. Unlocks the IVI session.
8. Returns the value of **error**.

The following sections reference this example to discuss the various features of a user-callable instrument driver function.

Locking/Unlocking the Session

In general, user-callable functions lock the IVI session at the beginning of the function and unlock the IVI session before returning. The `Fl45_Measure` example shows how to use the `Ivi_LockSession` and `Ivi_UnlockSession` functions for this purpose.

Caution Do *not* lock or unlock the IVI session in the *`Prefix`*`_init`, *`Prefix`*`_InitWithOptions`, *`Prefix`*`_IviInit`, and *`Prefix`*`_IviClose` functions.

Refer to the *LabWindows/CVI Help* for detailed information on how to use the locking functions.

Parameter Checking

An important step for user-callable functions is to check the parameters of the function. You *must* consider the following four types of parameters when range checking.

- The **vi** session parameter
- Input parameters that the function passes to one of the `Ivi_SetAttribute` functions
- Input parameters that the function uses directly
- Reference parameters

You do not check the **vi** session parameter in the function. The `Ivi_LockSession` reports an appropriate error if the session is invalid.

You do not check parameters that you pass to one of the `Ivi_SetAttribute` functions. The IVI engine invokes the check callback for the attribute to check these values. Use the `viCheckParm` macro to report errors that the IVI engine returns.

You *do* check parameters that the function uses directly. If the parameter checking has a significant performance penalty, check the parameters only if the `Ivi_RangeChecking` function returns `VI_TRUE`. Use the `viCheckParm` macro to report an error.

In most cases, user-callable functions use input parameters to set attributes, and the IVI engine checks the values. Therefore, user-callable functions typically do not range check parameters explicitly.

You must verify that reference parameter addresses are not NULL. Always perform this type of checking regardless of the value that the `Ivi_RangeChecking` function returns. If the address is NULL, use the `viCheckParm` macro to report the error and set the error elaboration string to `Null address for <parameter name>.`

The `Fl45_Measure` example illustrates the following items:

- The `Ivi_LockSession` reports an appropriate error if the **vi** is invalid.
- The IVI engine checks the value of the **function** parameter when the `Ivi_SetAttributeViInt32` function executes.
- The function always checks the address held in the **reading** reference parameter. If `reading` is NULL, the function sets the error elaboration string to `Null address for Reading`.

Accessing Attributes

You typically implement user-callable functions by manipulating attributes. Many user-callable functions only call `Ivi_SetAttribute` functions. When you call one of the `Ivi_SetAttribute` or `Ivi_GetAttribute` functions, you must consider the following:

- Call the `Ivi_GetAttribute` and `Ivi_SetAttribute` functions rather than the *`Prefix`*`_GetAttribute` and *`Prefix`*`_SetAttribute` functions that your driver exports. The *`Prefix`*`_` functions are for the instrument driver user only.
- Always call the get and set attribute functions regardless of whether you are simulating. Thus, place calls to the get and set attribute functions outside of any simulation/non-simulation block. The IVI engine handles simulation for attributes, including the validation of attribute values.

The `Fl45_Measure` example shows how to set the measurement function attribute.

Performing Direct Instrument I/O

If a user-callable function performs direct I/O to the instrument, the function must not perform the I/O when simulating. Use the `Ivi_Simulating` function to determine whether simulation is enabled.

Before performing direct instrument I/O, use the `Ivi_IOSession` function to obtain the I/O session handle for the instrument and pass `VI_TRUE` to the `Ivi_SetNeedToCheckStatus` function.

The `Fl45_Measure` example illustrates how to perform direct instrument I/O in a user-callable function.

Simulating Output Parameters

When simulating, user-callable functions must return simulated data in output parameters.

The IVI engine handles simulation for you in the `Ivi_GetAttribute` functions. The IVI engine returns the state of the attribute. Therefore, if the function obtains the value for an output parameter by calling one of the `Ivi_GetAttribute` functions, the function does *not* have to create simulated data.

A user-callable function is responsible for creating the simulated data if the function uses direct instrument I/O to obtain the value for the output parameter when simulation is disabled.

The `Fl45_Measure` example illustrates how to create and to return simulated data for output parameters.

Checking the Instrument Status

In general, you call the *`Prefix`*`_CheckStatus` utility function before the `Error` block in all user-callable functions, except with the following functions:

`Prefix``_init`
`Prefix``_InitWithOptions`
`Prefix``_close`
`Prefix``_IviClose`
`Prefix``_error_query`
`Prefix``_error_message`

Other exceptions to this rule are low-level user-callable functions that perform I/O that are included as part of a higher-level function. For example, `fl45_Read` is implemented by calling the `fl45_Initiate` and `fl45_Fetch` functions. The `fl45_Read` calls the *`Prefix`*`_CheckStatus` function, but the `fl45_Initiate` and `fl45_Fetch` functions do not. The instrument driver function panel help should document which functions do not implement status checking.

The *`Prefix`*`_CheckStatus` function is a utility function that calls the check status callback. Drivers that you develop from an IVI template already have a check status utility function. The check status utility function calls the check status callback when the following conditions are true:

- The `Ivi_QueryInstrStatus` function returns `VI_TRUE`.
- The `Ivi_NeedToCheckStatus` function returns `VI_TRUE`.
- The `Ivi_Simulating` function returns `VI_FALSE`.

The IVI engine sets a flag for the instrument session whenever it calls a read or write callback. The flag indicates that instrument I/O has occurred. The `Ivi_NeedToCheckStatus` function returns the value of this flag. The *`Prefix`*`_CheckStatus` function uses the `Ivi_NeedToCheckStatus` function to determine if instrument I/O has occurred since the driver last checked the instrument status. If no instrument I/O has occurred, the check status utility function does not check the instrument status. This flag enables the check status utility function to query the instrument status only when necessary.

Therefore, whenever a user-callable function performs direct instrument I/O, it must call the function `Ivi_SetNeedToCheckStatus` before performing the I/O. This call causes the *`Prefix`*`_CheckStatus` utility function to query the instrument status the next time it executes.

The `Fl45_Measure` example shows how to use the `Ivi_SetNeedToCheckStatus` function and the *`Prefix`*`_CheckStatus` utility function to perform status checking in user-callable functions.

Functions that Only Set Attributes

Not all user-callable functions perform direct instrument I/O. Some functions only set attributes. Examples of such functions are the high-level functions that configure groups of related attributes. You must structure such functions so that they set the attributes in the correct order for the instrument.

Note The configuration functions enable the instrument driver user to set multiple attributes without having to understand the order dependencies between the attributes. The IVI state-caching mechanism prevents redundant instrument I/O from occurring in the configuration functions.

The following example shows a function that only sets attributes.

```
/*****************************************************************
*Function: fl45_Configure Measurement
*Purpose: Configures the common attributes of the DMM. These
*attributes are FL45_ATTR_FUNCTION,
*               FL45_ATTR_RANGE, and
*               FL45_ATTR_RESOLUTION_ABSOLUTE.
*****************************************************************/

ViStatus _VI_FUNC  fl45_ConfigureMeasurement (ViSession vi,
                                              ViInt32 measFunction,
                                              ViReal64 range,
                                              ViReal64 resolution)
{
        ViStatus    error = VI_SUCCESS;

        checkErr( Ivi_LockSession (vi, VI_NULL));

               /* Set attributes: */
        viCheckParm( Ivi_SetAttributeViInt32 (vi, VI_NULL,
                    FL45_ATTR_FUNCTION, 0,measFunction), 2,
                    "Measurement Function");

        /*
            For the fl45, the resolution needs to be set before the
           range since the range is dependent on the resolution.
        */
```

```
        if (range != FL45_VAL_AUTO_RANGE_ON)
                {
                checkErr( Ivi_SetAttributeViReal64 (vi, VI_NULL,
                        FL45_ATTR_WANTED_RANGE,0, range));

                viCheckParm( Ivi_SetAttributeViReal64 (vi, VI_NULL,
                           FL45_ATTR_RESOLUTION_ABSOLUTE, 0,
                           resolution), 4, "Resolution");
                }
        viCheckParm( Ivi_SetAttributeViReal64 (vi, VI_NULL,
                    FL45_ATTR_RANGE, 0, range), 3, "Range");

        checkErr( fl45_CheckStatus (vi));

Error:
        Ivi_UnlockSession(vi, VI_NULL);
        return error;
}
```

Notice the following important features of the example function:

- The function does *not* perform parameter checking.
- The function does *not* have a simulating or non-simulating block.
- The function *does* check the instrument status.

Initialization Functions

There are two special considerations regarding the initialization functions:

- Do *not* call `Ivi_LockSession` or `Ivi_UnlockSession` in the *`Prefix`*`_init`, *`Prefix`*`_InitWithOptions`, and *`Prefix`*`_IviInit` functions.
- Check the instrument status in only the *`Prefix`*`_IviInit` function.

Channel Strings

IVI drivers use channel strings to identify the channels of an instrument. Typically, the *`Prefix`*`_IviInit` function calls the `Ivi_BuildChannelTable` function to specify the valid channel strings for the instrument driver.

For a multichannel instrument, you typically use channel strings such as `1`, `2`, `3`, `4`, or `A1` through `A4` and `D0` through `D15`. If your instrument has a front panel, use the channel names from the front panel. For message-based devices, the front panel channel name is usually the

same as the component of the instrument command string that identifies the channel. If the instrument commands use different strings to identify channels, complete the following steps:

1. Use the front panel channel names as the channel strings.
2. Create a utility function that converts a front panel channel name to the appropriate command string component for the instrument.

If your instrument does not support multiple channels, you must use `1` as the only valid channel string.

Close Functions

There are four special considerations regarding the close function:

- In the *`Prefix`*`_IviClose` function, set the `IVI_ATTR_IO_SESSION` attribute to `VI_NULL` before calling the `viClose` function.
- Do not call `Ivi_LockSession` or `Ivi_UnlockSession` in the *`Prefix`*`_IviClose` function.
- In the *`Prefix`*`_close` function, call `Ivi_UnlockSession` before calling `Ivi_Dispose`.
- Do not check the instrument status.

Developing Portable Instrument Drivers

When you are developing an instrument driver, consider making the driver portable across multiple compilers and operating systems. Established guidelines exist for the development of portable instrument driver code. The primary issues in developing portable instrument driver code involve data types, the declaration of user-callable functions, and the use of scan and formatting functions.

Instrument Driver Data Types

A subset of the VISA data types exists for use in the development of LabWindows/CVI instrument drivers. Use only these data types when defining instrument driver function parameters. The data types strictly define the type and size of the parameters and therefore enhance the portability of the functions to new operating systems and programming languages. Refer to Table 3-4 for a list of VISA data types.

Table 3-4. VISA Data Types

VISA Type Name	Definition
`ViInt16`	Signed 16-bit integer
`ViInt32`	Signed 32-bit integer

Table 3-4. VISA Data Types (Continued)

VISA Type Name	Definition
`ViReal64`	64-bit floating-point number
`ViInt16[]`	An array of `ViInt16` values
`ViInt32[]`	An array of `ViInt32` values
`ViReal64[]`	An array of `ViReal64` values
`ViChar[]`	A string buffer
`ViConstString`	A read-only string
`ViRsrc`	A VISA resource descriptor (`string`)
`ViSession`	A VISA session handle
`ViStatus`	A VISA return status type
`ViBoolean`	Boolean value

Declaring Instrument Driver Functions

The VISA I/O library also defines the `_VI_FUNC` macro that you must use when prototyping the user-callable functions of an instrument driver. Table 3-5 contains the macro definition for each platform.

Table 3-5. VISA I/O Library Macros

Macro	Outside the LabWindows/CVI Environment (Windows 3.1)	Windows 2000/95/98/NT	UNIX
_VI_FUNC	_far _pascal _export	__stdcall	—

The macro resolves differences between various platforms. Use the `_VI_FUNC` macro to define the calling convention of all user-callable functions.

The following example of an instrument driver function prototype uses the VISA data types and macros.

```
ViStatus _VI_FUNC tek2430a_read_waveform (ViSession instrSession,
                          ViReal64 wvfm[],ViReal64 *xin,
                          ViReal64 *trig_off);
```

Using Scan and Fmt Functions

In devices that manipulate large arrays of data, such as oscilloscopes or arbitrary waveform generators, you transfer data from computer to instrument or from instrument to computer in a binary format to improve throughput and performance. When you transfer data in a binary format, you must manipulate arrays of binary data, typically integer arrays. Under normal circumstances, manipulating arrays of binary data is not a problem. However, the differences between operating systems and programming languages in which the drivers might be used in the future require more attention in this area. With this in mind, you must give special consideration to code segments that handle binary instrument data.

The following important rules exist for developing portable instrument driver code using `Scan` and `Fmt` functions.

- If you are using a `Scan` or `Fmt` statement to manipulate binary data that has been received from an instrument or that will be sent to an instrument, use an `o` modifier on the side of the `Scan` or `Fmt` statement that represents the binary data.

  The `o` modifier describes the byte ordering.

  Example: Intel `[o01]`
  Motorola `[o10]`

- Whenever you are scanning or formatting binary data, use an array of either type `short`, `long`, or one of the VISA data types, rather than simply `int`. The representations of shorts, longs, and the VISA data types are the same on all LabWindows/CVI platforms.
- When using a `Scan` or `Fmt` statement to scan or format data in to or out of an array of type `short`, `long`, or one of the VISA data types, use the `b` modifier to represent the width of the data. When you are scanning or formatting data in to or out of an array of type `int`, do not use the `b` modifier to represent the width of the data.

The following code example shows the correct way to scan binary data that you receive from an instrument. In the code example, the `viRead` function transfers the binary waveform information from the instrument to the `cmd` buffer. Then the `Scan` function parses the binary information and places it in the `ViInt16` array `wavefrm`. Notice the following:

- The `o` modifier is on the side of the `Scan` statement that represents the binary data that you receive from the instrument.
- The `b` modifier is used on both sides of the `Scan` function. It represents the size of the binary data and the element size of the `wavefrm` array.

```
ViInt16 wavefrm[4000];
ViUInt32 retCnt;

ViCheckErr (viRead (10, cmd, 1024, &retCnt));
Scan (cmd, "%*d[zb2o10]>%*d[b2]", 512, 512, wavefrm);
```

Error-Reporting Guidelines

One of the most important operations performed in an instrument driver is reporting the status of each operation. Each user-callable routine is a function with a return value of the type `ViStatus` that returns the appropriate error or warning value.

Table 3-6 presents a scheme for determining error values. It lists predefined error codes for instrument drivers.

Table 3-6. Suggested Error Values

Value	Meaning
0	No error occurred.
Positive values	Completion or warning codes, such as warnings for instrument driver features that are not supported by the device or I/O completion codes the VISA I/O libraries return.
Negative values	Errors that are detected in an instrument driver, such as the range-checking of function parameters or I/O errors the VISA I/O libraries report.

Refer to *LabWindows/CVI Help* for a complete list of IVI error codes that you can use in your driver.

The defined names for completion and error codes in Tables 3-7 and 3-8 are resolved in the file `vpptype.h`. The include file, `ivi.h`, includes `vpptype.h`. By including the file `ivi.h` in your instrument driver header file, you can use these defined names in your instrument driver. IVI also defines additional errors.

Table 3-7. Instrument Driver Completion and Warning Codes

Completion Code	Description	Error Number
`VI_SUCCESS`	No error: the call was successful	—
`VI_WARN_NSUP_ID_QUERY`	Identification query not supported	`0x3FFC0101L`
`VI_WARN_NSUP_RESET`	Reset not supported	`0x3FFC0102L`
`VI_WARN_NSUP_SELF_TEST`	Self-test not supported	`0x3FFC0103L`
`VI_WARN_NSUP_ERROR_QUERY`	Error query not supported	`0x3FFC0104L`
`VI_WARN_NSUP_REV_QUERY`	Revision query not supported	`0x3FFC0105L`

Table 3-8. Instrument Driver Error Codes

Status	Description	Error Numbers
VI_ERROR_FAIL_ID_QUERY	Instrument identification query failed	0xBFFC0011L
VI_ERROR_INV_RESPONSE	Error interpreting instrument response	0xBFFC0012L

VI_ERROR_INV_RESPONSE (*Error in interpreting an instrument response*) is an important error code. This error occurs when a Scan statement tries to parse data from an erroneous instrument response. In the following code, VI_ERROR_INV_RESPONSE is returned if the scan does not place data in the header and wvfm variables.

```
if (Scan (in_data, "%1027i[b1u]>%3i[b1]%1024f", header, wvfm) != 2)
                            ViCheckErr (VI_ERROR_INV_RESPONSE)
```

Refer to the **Library Reference»IVI Instrument Driver Developer Library»Error Reporting** and **Error Macros** topics in the *LabWindows/CVI Help* for additional guidelines on reporting errors.

General Programming Guidelines

The following guidelines relate to general programming practices.

- Base your instrument driver on an existing instrument driver or one of the class templates.
- Avoid declaring function names that exceed 31 characters.
- Choose an instrument prefix that uniquely identifies the instrument driver.
- Make the base filename of the instrument driver files the same as the prefix for the instrument driver and the base filename of the .fp file. For example, the filenames for a driver might be tek2430a.fp, tek2430a.sub, tek2430a.c, and tek2430a.h.
- Use only the VISA I/O library to perform instrument I/O where possible.
- Use only the VISA data types.
- Include the file, ivi.h, in the include file for your instrument driver. This file includes both vpptype.h and visa.h.
- Declare functions that do not return values as void. You must include a return value control in a panel for functions that return values.
- Avoid declaring large arrays within instrument drivers because arrays use large amounts of memory.
- Do not use the FmtOut, printf, and user interface functions.

- Avoid using Advanced Analysis Library functions in instrument drivers. Many LabWindows/CVI users do not have the Advanced Analysis Library, which is available only with the Measurement Studio LabWindows/CVI Full Development System.
- Test all the instrument drivers you create. Test them in LabWindows/CVI and in standalone applications.
- Avoid declaring global or static local variables. Instead, create attributes to store the data.
- If it is an error for the user to set an attribute when the instrument is in certain configurations, you must check for these conditions in the check callback rather than the write callback. Because the default check callback uses only the range table, you must create a custom callback for this purpose. However, you can call the `Ivi_DefaultCheckCallback` function in your custom callback.
- Use only the IVI memory allocation functions to dynamically allocate memory in your instrument driver.

Function Panels

The function panels link the user and the user-callable functions. Function panels let users interactively control the instrument and develop application programs. You should create function panels with the user in mind. Make the panels look like other instrument drivers in the LabWindows/CVI Instrument Library. Arrange controls neatly and center them on the panel. Place the instrument ID control in the lower left corner. Place the error return control in the lower right corner of every function panel.

Function Tree Hierarchy

The function tree defines the relationship between each function panel. Users think in terms of high-level application operations such as `Initialize`, `Configure`, `Measure`, and so on. Group the functions in the function tree accordingly. Make function trees from similar instruments look similar.

For example, Figure 3-3 shows the Fluke 45 instrument driver function tree.

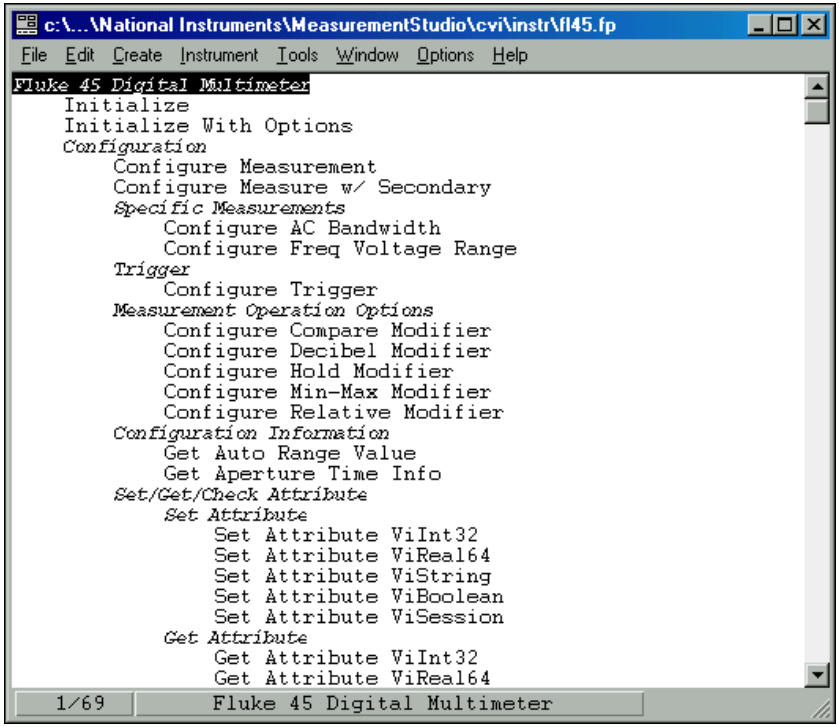

Figure 3-3. Fluke 45 Digital Multimeter Function Tree

The functions are easy to understand, and the instrument driver user can immediately incorporate them into an application program. Develop your function tree with an application in mind and place the functions in the natural order in which they will be used. Again, keep your function tree consistent with others in the LabWindows/CVI Instrument Library. Refer to the **Using LabWindows/CVI»Function Tree Editor** section of the *LabWindows/CVI Help* for more information on editing function trees.

Documentation Guidelines

Writing useful documentation is an essential step in developing instrument drivers. Proper documentation helps the user understand the instrument driver and its functions. Instrument driver documentation consists of the following resources:

- Online help from within LabWindows/CVI function trees and function panels.
- A `.doc` file you distribute with the instrument driver files. The `.doc` file is an ASCII file that contains the online help from the function panels.

Online Help

Users consult the online help of an instrument driver frequently. Include online help at every level of the instrument driver.

The following examples present the types of help information found in the Fluke 45 instrument driver. Use these example help screens as a guide when editing online help for your instrument driver.

Note You should add help text when you create or edit the function tree or function panels. Online help text is stored as part of the `.fp` file.

- Instrument driver help describes the instrument driver. Figure 3-4 shows instrument help for the Fluke 45 instrument driver.

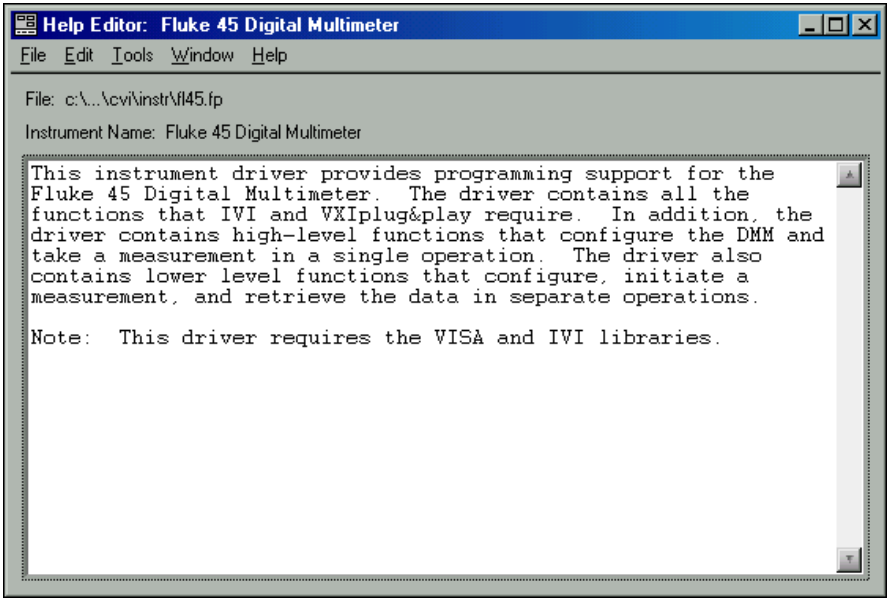

Figure 3-4. Fluke 45 Instrument Help

- Function class help briefly describes all the functions and subclasses beneath the selected function class. Figure 3-5 shows function class help from the Fluke 45 instrument driver.

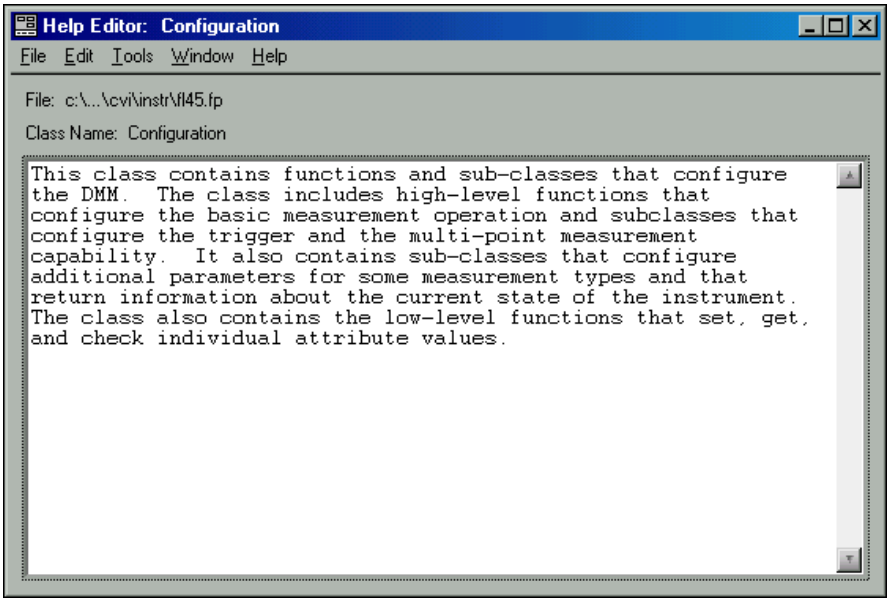

Figure 3-5. Fluke 45 Function Class Help

- Function panel help describes the function call. Figure 3-6 shows the function panel help from the Fluke 45 instrument driver.

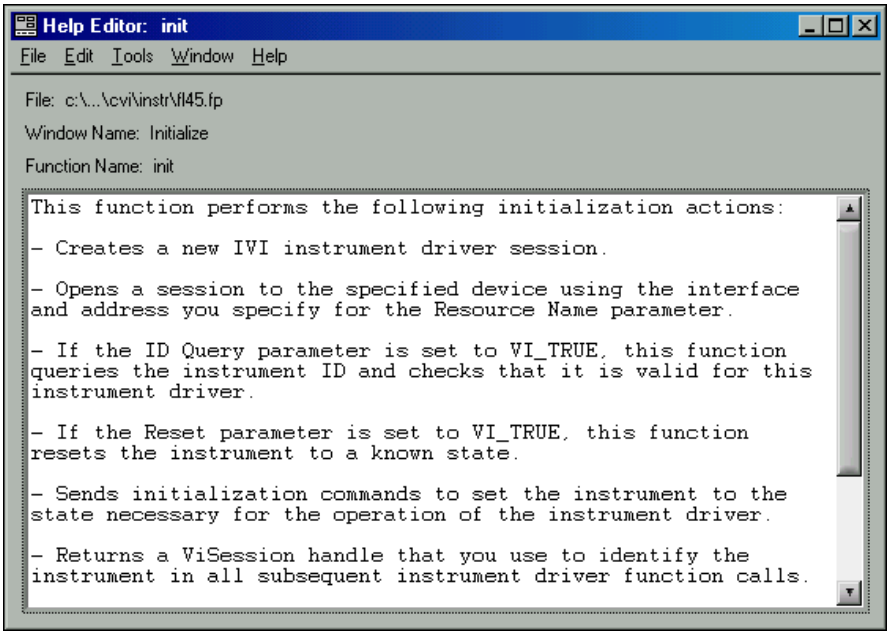

Figure 3-6. Fluke 45 Function Panel Help

- Control help contains a description of the parameter, the valid range, and the default value. Figure 3-7 shows an example of function panel control help from the Fluke 45 instrument driver.

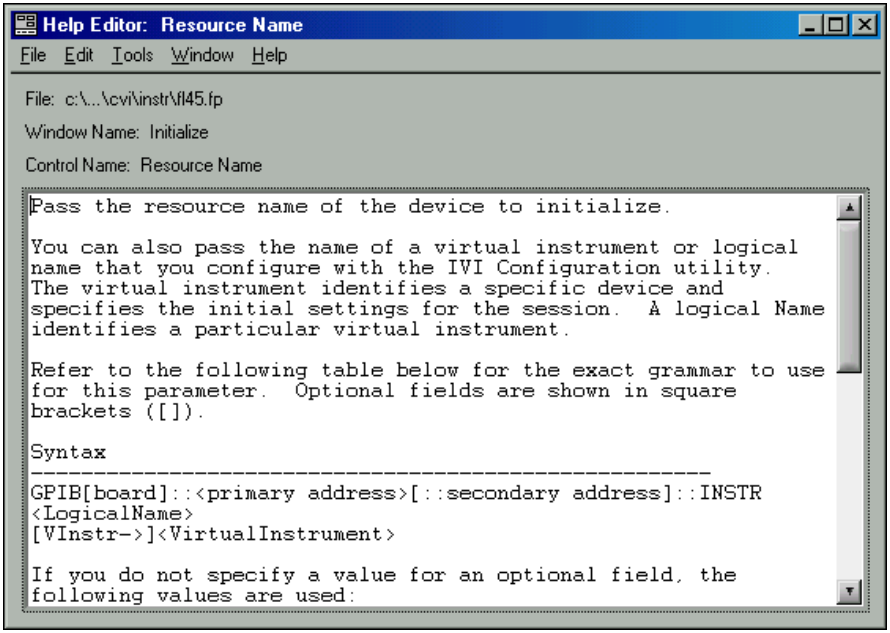

Figure 3-7. Fluke 45 Function Panel Control Help

- Status help contains a description of the parameter and the possible error values. Since the possible errors are often too numerous to document effectively, the help should contain guidelines for retrieving error messages as well as error code ranges from various sources. Figure 3-8 shows an example of status control help from the Fluke 45 instrument driver.

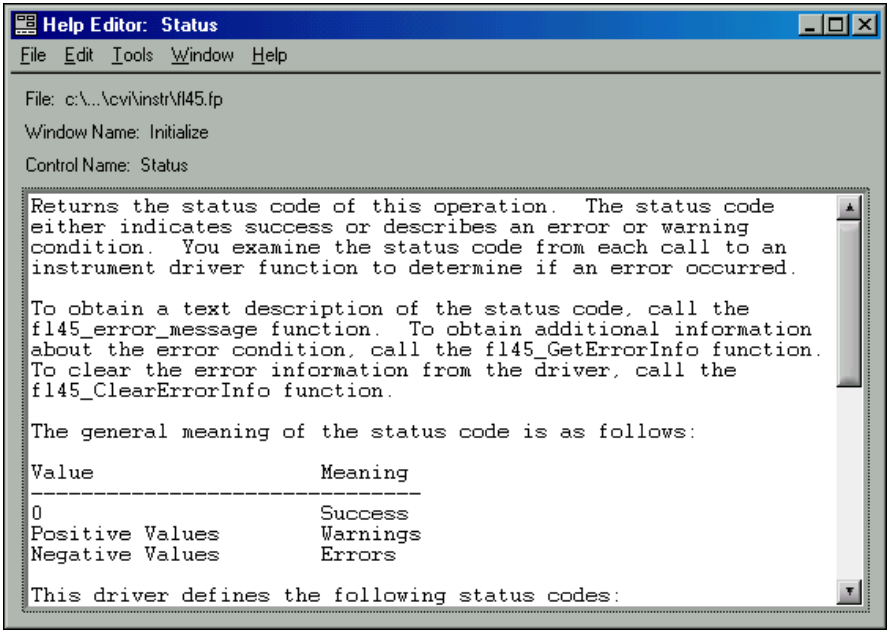

Figure 3-8. Fluke 45 Function Panel Status Control Help

Documentation Files

LabWindows/CVI can generate function reference help in two formats, a `.doc` file and a `.hlp` file. The `.doc` file is an ASCII text file. The `.hlp` file is a Windows help file. Both formats contain the following information:

- A brief description of the instrument
- A function tree layout
- Assumptions made by the driver developer
- A list of the LabWindows/CVI libraries that are referenced in the driver
- A description of each function, including the following:
  - Syntax
  - Purpose
  - Parameter types

  - Function type
  - Error codes
- A description of each attribute

Give the `.doc` and `.hlp` files the same base filename as the `.fp` file for the instrument driver.

You can generate a `.doc` file by selecting **Options»Generate Documentation** in the Function Tree Editor window. You can generate a `.hlp` by selecting **Options»Generate Windows Help** in the Function Tree Editor window.

Programming Guidelines for VXI Instruments

When developing drivers for VXI register-based devices, consider the following:

- When you create a driver from a template, the driver contains the *Prefix*`_ReadInstrData` and *Prefix*`_WriteInstrData` functions. These functions enable the instrument driver user to transfer character data to the instrument. You must delete the functions from the function panel file when you develop a driver for a register-based device.
- Range table entries include a command value field. Instrument drivers for register-based devices use this field to store the register value that corresponds to the entry in the range table.

Instrument Driver Checklist

All instrument drivers you add to the LabWindows/CVI Instrument Driver Library must conform to the recommendations for programming style, error handling, function tree organization, function panels, and online help. The following form is an abbreviated version of the form used to check all instrument drivers submitted for inclusion in the LabWindows/CVI Instrument Library. Use this form to verify that your instrument driver is complete and correct.

I. Function Tree

____ A. Has a logical structure and follows the instrument driver internal design model

____ B. Has all required instrument driver functions

____ C. Contains a function panel window for every user-callable function, except *Prefix*`_IviInit` and *Prefix*`_IviClose`

____ D. Has help text for all function tree nodes

II. Each Function Panel

____ A. Has an Instrument Handle control in the lower left corner

____ B. Has a Status return value control in the lower right corner

____ C. Presents all controls in a neatly organized manner

____ D. Defines a reasonable default value or no default value, as is appropriate, for each control

____ E. Uses the proper display format for each control, such as hexadecimal format for controls that represent the contents of status registers

____ F. Help text must include the following items:

____ 1. Exists for the function and for all controls

____ 2. Is in the correct format, which includes:

____ a. Description

____ b. Valid range and default value

____ c. The status control help provides users with enough information to discover the source of status codes that the function might return

____ 3. Excludes all modification instructions

III. Source File

____ A. Includes standard instrument driver comments, as listed below:

____ 1. Instrument manufacturer and name

____ 2. Author identification

____ 3. Modifications history

____ B. Includes the `visa.h` and instrument driver header files, either directly or indirectly by inclusion of another include file

____ C. Declares all functions that are not user-callable as `static`

____ D. Contains declarations for only `static` functions

____ E. Defines ID constants and values only for hidden attributes

____ F. Defines the prototype for each user-callable function:

____ 1. Includes the `_VI_FUNC` macro before the function name

____ 2. Uses only VISA data types for parameters

____ 3. Defines the return value type as `ViStatus`

____ 4. Declares arrays with square brackets (`[]`), such as `ViReal64 readingArray[]`

____ G. Uses the following structure for each user-callable function:

____ 1. Locks and unlocks the instrument driver session. Exceptions include the `Prefix_init`, `Prefix_InitWithOptions`, `Prefix_IviInit`, and `Prefix_IviClose` functions

____ 2. Checks parameters when necessary

____ 3. Calls the `Ivi_Get/SetAttribute` functions outside simulating/non-simulating blocks

____ 4. Performs direct instrument I/O in a non-simulating block

____ 5. Creates and returns simulated data for output parameters when necessary

____ 6. Calls the internal *`Prefix`*`_CheckStatus` function. Exceptions include the functions listed in the *Status Checking* section of Chapter 2, *IVI Architecture Overview*

____ H. Uses VISA for all instrument I/O, if possible

____ 1. Correctly uses the `viScanf` and `viRead` operations

____ I. Scans or formats binary instrument data for multiplatform use

____ J. Checks all `Scan` function calls for errors

____ K. Reports all errors using appropriate error codes

____ L. Uses the IVI error macros properly

____ M. Does not modify the contents of static range tables programmatically

____ N. Does not perform screen I/O, such as writing to the Standard Output, reading from the Standard Input, or calling the LabWindows/CVI User Interface library

____ O. Never calls the `exit` function

____ P. Includes complete and descriptive comments

____ Q. Excludes all modification instructions

IV. Include file

____ A. Includes the `ivi.h` header files

____ B. Declares only user-callable instrument driver functions

____ C. Defines ID constants and values only for public attributes

____ D. Defines all necessary constants, including attribute values and error codes

____ E. Correctly formats function prototypes:

____ 1. Include the macro `_VI_FUNC` before the function name

____ 2. Use only VISA data types for function parameters

____ 3. Define the return value type as `ViStatus`

____ 4. Declare arrays with square brackets (`[]`), such as `ViReal64 readingArray[]`

____ F. You have deleted all modification instructions

4

Attribute Editor

This chapter describes the operation of the Attribute Editor. It describes the dialog boxes and controls in the Attribute Editor and explains how to use the Attribute Editor to modify and navigate through instrument driver source files that the Instrument Driver Development Wizard generates.

Invoking the Attribute Editor

Select **Tools»Edit Instrument Attributes** in a Source, Function Tree Editor, or Function Panel Editor window to launch the Attribute Editor. The Attribute Editor analyzes the instrument driver source files to build a list of all attributes that the driver defines. In particular, the Attribute Editor analyzes the contents of the `Prefix_InitAttributes` function and range tables in the `.c` file for the driver. It also analyzes the contents of the `.sub` and `.h` files.

Requirements for Using the Attribute Editor

The Attribute Editor makes certain assumptions about the driver files, which enables the Attribute Editor to parse your files and present your attributes in a simple, easy-to-use manner. If your driver files violate these assumptions, an error message appears when you attempt to invoke the Attribute Editor. When you use the Instrument Driver Development Wizard to create your driver, the resulting files satisfy the requirements for using the Attribute Editor. As you modify the files by hand, keep these requirements in mind. The requirements are as follows:

- All four driver files must be in the same directory on your computer or correctly located in the VXI*plug&play* directory. The files must have the same base filename and the `.c`, `.h`, `.fp`, and `.sub` extensions.
- The `.c` file must define a `Prefix_InitAttributes` function that contains the calls to the `Ivi_AddAttribute` functions that create the attributes for the driver. It also can contain calls to `Ivi_AddAttributeInvalidation` and calls to the IVI Library functions that install check, coerce, compare, and range table callbacks.
- All attribute ID parameters to these calls must be the actual defined constant names for the attributes. The parameters must not be variables.
- In each `Ivi_AddAttribute` call, the **attributeName** parameter must be a literal string that contains the defined constant name. It must not be a variable.

- In each `Ivi_AddAttribute` call, the **flags** parameter must be zero or one or more defined constant names for attribute flags. If you pass multiple defined constant names, separate them with a vertical bar (|).
- The code in *`Prefix`*`_InitAttributes` must be syntactically correct.
- The code in each range table in your source file must be syntactically correct.
- For each range table in the `.c` file, the name of the range table entries array must be the name of the range table followed by `Entries`.
- The parameter names in callback function definitions must be the same as the parameter names that the Instrument Driver Development Wizard generates.

Limitations in Updates to Driver Files

When the Attribute Editor applies changes to the driver source and header files, it does not retain comments in the following items:

- Range tables.
- The prototype portion of callback function definitions. The prototype consists of the return type, function name, and parameter list, up through the opening curly brace of the function body.
- `#define` statements.
- Calls to IVI Library functions in the *`Prefix`*`_InitAttributes`.

When the Attribute Editor generates calls to IVI Library functions in the *`Prefix`*`_InitAttributes` function, it always uses `vi` as the name for the IVI session handle parameter, and it always wraps the call with the `checkErr` macro. Refer to the *LabWindows/CVI Help* for more information on `checkErr`.

Edit Driver Attributes Dialog Box

When you launch the Attribute Editor, the Edit Driver Attributes dialog box appears, as shown in Figure 4-1.

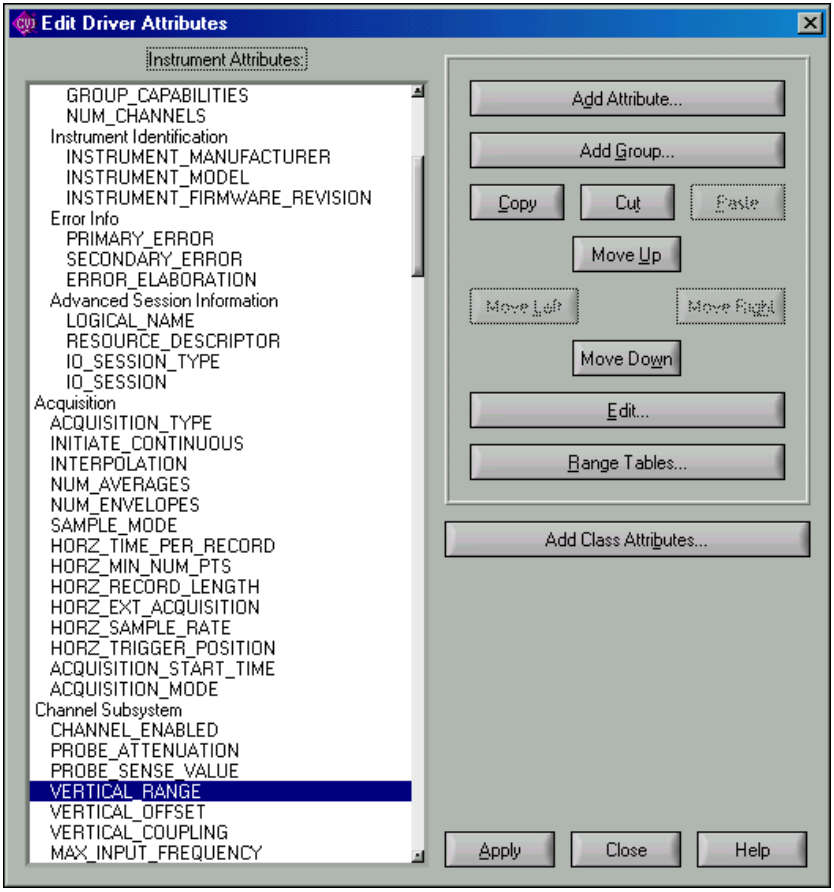

Figure 4-1. Edit Driver Attribute Dialog Box

Instrument Attributes List Box

The Instrument Attributes list box on the left side of the dialog box contains all attributes for which the Attribute Editor found an `Ivi_AddAttribute` call in the *`Prefix`*`_InitAttributes` function. The list box also displays the inherent IVI attributes that the `.sub` file describes. The Instrument Driver Development Wizard generates the descriptions of the inherent attributes in the `.sub` file automatically.

The attributes are grouped hierarchically in the list box. You can have group labels on the first and second level. You can have attributes at the first, second, or third level. The Attribute Editor reads the hierarchy information from the `.sub` file. After you select an entry, double-click, press <Enter>, or right-click in the listbox and select **Edit** to display the Edit Group or Edit Attribute dialog box, depending on the selected entry.

When you right-click on an item in the list box or press <Alt+Enter> when an item is selected, a context menu is displayed. The context menu always contains an entry to edit the selected item. If you select an attribute in the list box, then the context menu contains additional entries to navigate to an available callback or range table.

Restrictions on Modifications to Inherent and Class Attributes

In the list box, special rules apply to inherent and class attributes. Refer to the *Types of Attributes* section in Chapter 2, *IVI Architecture Overview*, for information on the distinction between inherent, class, and instrument-specific attributes.

The following items are not possible:

- Edit, expand, cut, or copy an inherent IVI attribute.
- Move the items within the Inherent IVI Attributes group.
- Edit the label or help text for the group. You can, however, move the entire group up or down in the list box.

When you use the Instrument Driver Development Wizard with a predefined instrument template, the wizard generates all the instrument class attributes into your driver files. In general, you can edit, expand, copy, and move class attributes freely. Also, you can cut class attributes. The Attribute Editor prompts you for confirmation when you cut a class attribute that the class driver include file indicates is fundamental to the class. When you cut an extended class attribute, no confirmation is necessary.

Attributes List Box Command Buttons

This section describes the command buttons in the Edit Driver Attributes dialog box.

- **Add Attribute**—Adds a new attribute to the list box. The command invokes an empty Edit Attribute dialog box in which you can enter the attribute information. Refer to the *Adding and Editing Instrument Attributes* section in this chapter for more information on the Edit Attribute dialog box.
- **Add Group**—Adds a new group label to the list box. The command launches an empty Edit Group dialog box in which you can enter the label and help text for the group. Instrument driver users view the help text in the Select Attribute Constant dialog box that they access in the `Get/Set/CheckAttribute` function panels.
- **Move Up/Down**—Moves attributes up or down in the list box.

- **Move Right/Left**—Indents attributes left or right for logical grouping and readability.
- **Edit**—Modifies a group or attribute. The command launches the Edit Group or Edit Attribute dialog box for the currently selected item.
- **Range Tables**—Displays a list of range tables in the driver. Refer to the *Adding and Editing Range Tables* section in this chapter for more information.
- **Add Class Attributes**—Adds class attributes that your driver currently does not implement. This action can be useful if you previously deleted one or more class attributes or if a new version of the class definition contains additional attributes. The Attribute Editor identifies the class definition that your specific driver uses by searching your specific driver header file for an `#include` statement that refers to a class header file. If the Attribute Editor cannot find an `#include` statement for a class driver header file, the **Add Class Attributes** button appears dim. When you click the **Add Class Attributes** button, it uses the predefined instrument template for the class that the Instrument Driver Development Wizard uses. It builds a list of the class attributes that are not currently in the list box. You can select one or more attributes to add.
- **Apply**—Updates the `.c`, `.h`, and `.sub` files for the driver with the changes you have made in the Attribute Editor. Changes include generating empty function definitions for callbacks you added, removing callbacks you deleted, adding or modifying range table entries, adding `#define` statements for attributes in the header file, and modifying help text in the `.sub` file. You can apply your changes without exiting the Attribute Editor.

  When you invoke the Attribute Editor, your source file might reference range tables in the *`Prefix`*`_InitAttributes` function even though it contains no definitions for them. When you click the **Apply** button, the Attribute Editor creates empty definitions for them. The Attribute Editor will not create empty definitions if the range table name is a variable. Refer to the *Adding and Editing Range Tables* section in this chapter for more information.

  The first time you click **Apply** after invoking the Attribute Editor, the Attribute Editor backs up your instrument driver files before it applies your modifications. If your driver files have unsaved changes, the Attribute Editor prompts you to save your files before it backs them up. The Attribute Editor creates the backup files in the directory that contains the driver files. On most platforms, it copies the driver files and appends `.bak` to each filename. The Attribute Editor overwrites any backup files that already exist in the directory.

  After **Apply** updates your driver files with your modifications, the Attribute Editor saves your driver files to disk. The Attribute Editor also purges all undo information for the `.c` and `.h` files.
- **Close**—Exits the Attribute Editor. **Close** prompts you to apply any unsaved changes.

Adding and Editing Instrument Attributes

Add and edit instrument driver attributes in the Edit Attribute dialog box, which appears in Figure 4-2.

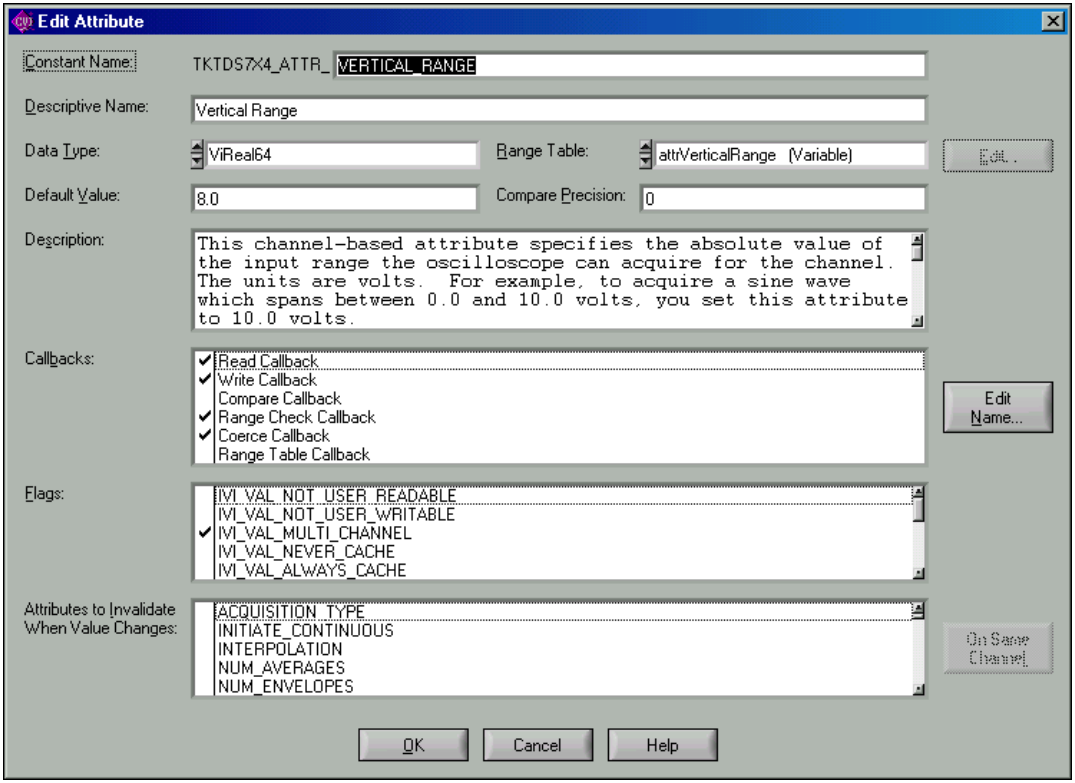

Figure 4-2. Edit Attribute Dialog Box

When you apply your changes, the Attribute Editor saves this information into the `.c`, `.h`, and `.sub` files for your instrument driver.

- **Constant Name**—Specifies the defined constant name of the attribute. All attribute constant names must begin with *`Prefix`*`_ATTR_`, where *`Prefix`* is the instrument prefix. You enter the rest of the name in this control.
- **Descriptive Name**—Specifies the name that appears for the attribute in the Select Attribute Constant dialog box that users can invoke in the `Get/Set/CheckAttribute` function panels.

- **Data Type**—Lets you select the data type of the attribute. The data type can be `ViInt32`, `ViReal64`, `ViString`, `ViBoolean`, `ViSession`, and `ViAddr`. You can use `ViAddr` only for attributes that you mark as hidden from the instrument driver user. You can hide an attribute from the instrument driver user by adding checkmarks to the `IVI_VAL_USER_NOT_READABLE` and `IVI_VAL_USER_NOT_WRITABLE` flags. If these two flags are set when you apply your changes, the Attribute Editor converts them to the `IVI_VAL_HIDDEN` macro in your source code.
- **Range Table**—Lets you select a static range table for the attribute. You can select an existing range table in the ring control. If you do not want a range table, you can select the **None** entry in the ring control. The **Range Table** ring control is enabled only for `ViInt32` and `ViReal64` attributes. The data type of the attribute must match the data type that appears in the Edit Range Table dialog box for the range table that you select.
- **Edit button**—Invokes the Edit Range Table dialog box for the range table that currently appears in the **Range Table** ring control. When the **None** entry appears in the range table ring control, the label of the button changes to **New**. The **New** button brings up the Edit Range Table dialog box for a new range table. Refer to the *Adding and Editing Range Tables* section in this chapter for more information on the Edit Range Table dialog box.
- **Default Value**—Specifies the default value for the attribute. The IVI engine uses the default value only when simulation is enabled and you obtain the value of the attribute before you set it. In effect, the default value represents a simulated initial value for the attribute. In the **Default Value** control, enter a valid C expression that is appropriate to the data type of the attribute.
- **Compare Precision**—Lets you specify the number of digits of precision to use when comparing a cache value that the IVI engine obtained from the instrument against a value you want to set this attribute to. The number of digits can be from 1 to 14. If you specify zero, the IVI engine uses the default, which is 14. Because you might want to enter a defined constant for this value, the **Compare Precision** control is a string control. This control applies only to `ViReal64` attributes. Refer to the *Comparison Precision* section in Chapter 2, *IVI Architecture Overview*, for more information.
- **Description**—Specifies the help text that the `.sub` file stores for the attribute. Instrument driver users view the help text in the Select Attribute Constant dialog box that they invoke in the `Get/Set/CheckAttribute` function panels. It is not necessary to enter help text for attributes that you hide from the user.
- **Callbacks**—A list of the different types of attribute callback functions. A checkmark next to a callback type indicates that a callback of that type has been associated with this attribute. You can dissociate a callback function from an attribute by removing the checkmark. If you add a checkmark on a callback type that did not initially have one, the Attribute Editor associates a default callback function name with the attribute. When you apply your changes, the Attribute Editor inserts the skeleton code for the new callback function in your source file. You can toggle the checkmark by pressing <spacebar> or by clicking the checkmark.

- **Edit Name**—Invokes the Edit Callback Name dialog box for the callback that is currently selected in the **Callbacks** list window. Use this dialog box to specify custom names for the attribute callback functions. The Instrument Driver Development Wizard constructs default names for attribute functions. The Attribute Editor constructs default names in the same manner when you enable a callback function for an attribute in the Edit Driver Attributes dialog box. You do not have to use this dialog box unless you want to specify a callback function name other than the default name. If you want to reset the callback name to the default name, remove the checkmark next to the name and add it again.
- **Flags**—A list of the flags that you can set for the attribute. You set a flag by adding a checkmark next to its name. You can hide an attribute from the instrument driver user by adding checkmarks to the `IVI_VAL_USER_NOT_READABLE` and `IVI_VAL_USER_NOT_WRITABLE` flags. If these two flags are set when you apply your changes, the Attribute Editor converts them to the `IVI_VAL_HIDDEN` macro in your source code. Refer to the *Attribute Flags* section in Chapter 2, *IVI Architecture Overview*, for detailed information about each flag.
- **Attributes to Invalidate When Value Changes**—Lists all the other attributes in the instrument driver. Add a checkmark next to the attributes whose cache values you want the IVI engine to invalidate when you change the value of the attribute that you are currently editing. The Attribute Editor generates a call to `Ivi_AddAttributeInvalidation` for each attribute that has a checkmark. Refer to the *IVI State-Caching Mechanism* section in Chapter 2, *IVI Architecture Overview*, for information about invalidation of attribute cache values.
- **On All/Same Channels**—Allows you to specify whether an attribute invalidation occurs on all channels or only on the channel on which the value of the attribute you are currently editing changes. Click the button to toggle between the On All Channels and On Same Channel state. If the item you select in the **Attributes to Invalidate When Value Changes** list box has a checkmark, the (`All`) or (`Same`) tag appears at the end of the item label. If the item you select does not have a checkmark, the toggle button is dim. Notice that this option has no effect unless the attribute you are editing and the attribute it invalidates are both channel based.

Adding and Editing Range Tables

Range tables define valid values for attributes. Generally, only `ViInt32` and `ViReal64` attributes have range tables. Refer to the *Range Tables* section in Chapter 2, *IVI Architecture Overview*, for detailed information on range tables.

When you invoke the Attribute Editor, it finds all the range tables you define in your source file. It also associates a range table to an attribute when you pass the address of the range table to `Ivi_AddAttributeViInt32` and `Ivi_AddAttributeViReal64`.

The Attribute Editor does not associate range tables for attributes with data types other than `ViInt32` and `ViReal64`. If you pass a variable name for the range table pointer parameter to `Ivi_AddAttributeViInt32` or `Ivi_AddAttributeViReal64`, the Attribute Editor maintains the association of the attribute with the range table pointer variable name. The Attribute Editor assumes that the parameter is a variable name if you do not precede it with an ampersand (`&`) and it does not find a range table of that name in the source file.

When you click the **Range Tables** button in the Edit Driver Attributes dialog box, the Range Tables dialog box appears as shown in Figure 4-3.

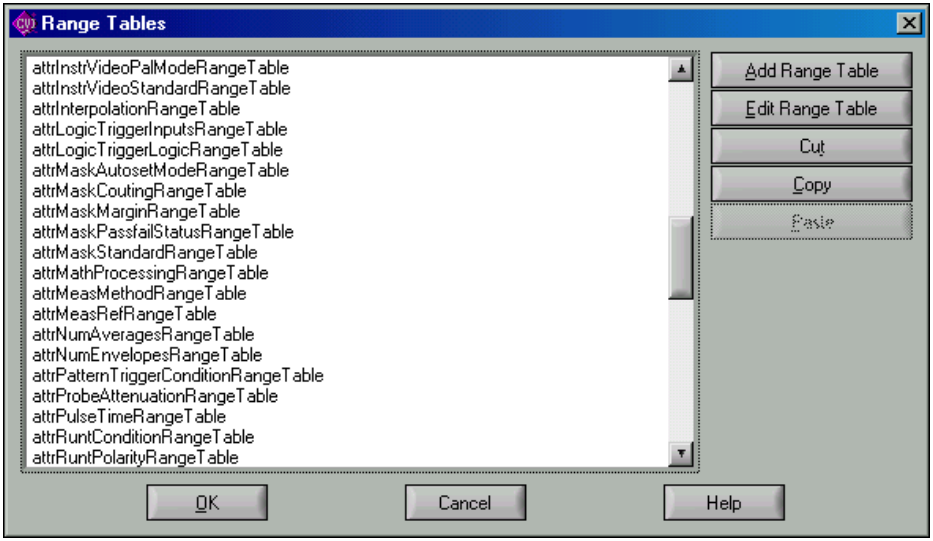

Figure 4-3. Range Tables Dialog Box

- **Add Range Table**—Adds a new range table. The command launches the Edit Range Table dialog box with default information.
- **Edit Range Table**—Edits an existing range table. The command launches the Edit Range Table dialog box for the currently selected range table.

The Edit Range Table dialog box appears as shown in Figure 4-4.

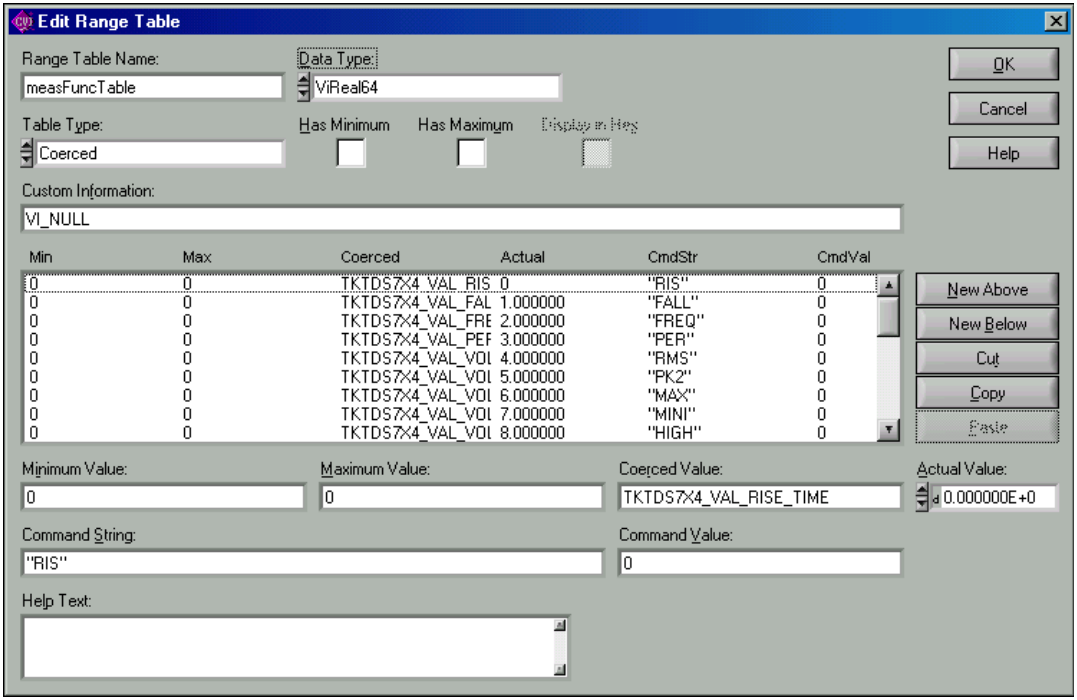

Figure 4-4. Edit Range Table Dialog Box

In the Edit Range Table dialog box, enter all the information that is necessary for the Attribute Editor to generate a definition for the range table in the source file. For a discrete or coerced range table, you also enter help text and an actual numeric value for each table entry. Instrument driver users view the help text and actual values in the Select Attribute Constant dialog box that they launch in the `Get/Set/CheckAttribute` function panels. The Attribute Editor also uses the actual values when it generates `#define` statements in the driver header file for previously undefined constant names that you specify in the Discrete Value or Coerced Value fields of table entries.

- **Range Table Name**—Specifies the name for the range table in your source code.
- **Data Type**—Lets you select the data type to use for the Actual Value controls for each entry of a discrete or coerced range table. Remember that range tables always store entries with `ViReal64` values. The Attribute Editor uses the data type to give you the correct type of numeric control for the actual values and to write the actual values to the `.h` and `.sub` files in the correct format. The data type you select must match the data type of the attributes you associate with the range table.

- **Table Type**—Lets you select from **Discrete**, **Ranged**, or **Coerced**. Refer to the *Range Tables* section in Chapter 2, *IVI Architecture Overview*, for detailed information on the three types of range tables. Notice that when you switch table types, some of the other controls on the dialog box change. In general, coerced and discrete tables are the most common. When you use a ranged range table, you typically create only one entry in the table.
- **Has Minimum**—Specifies whether the range table, as a whole, contains a meaningful minimum value. This control appears only when the **Table Type** is **Coerced** or **Ranged**. For a coerced range table, the minimum value represents the minimum coerced value.
- **Has Maximum**—Specifies whether the range table, as a whole, contains a meaningful maximum value. This control appears only when the **Table Type** is **Coerced** or **Ranged**. For a coerced range table, the minimum value represents the maximum coerced value.
- **Display In Hex**—Specifies whether to display the actual values in hexadecimal in the Select Attribute Constant dialog box that users can invoke in the `Get/Set/CheckAttribute` function panels. This control is dim when the data type is `ViReal64`.
- **Custom Information**—Specifies the contents of the `customInfo` field of the range table structure. If you do not want to use the `customInfo` field, enter `VI_NULL`. Otherwise, enter a string surrounded by double quotes.
- **Entries**—Contains the contents of each range table entry in a list box. The columns that appear depend on the type of range table. When you select an entry in the list box, its contents appear in the controls that are below the list box. You can add new entries to the range table by clicking the **New Above** and **New Below** buttons that are to the right of the list box, or by pressing <Enter> in the **Command Value** or **Help Text** controls.
- **Minimum Value**—Lets you specify the minimum value for the currently selected range table entry. This control appears only when the **Table Type** is **Coerced** or **Ranged**. The control lets you specify a text entry so that you can enter a defined constant, literal value, or expression. The Attribute Editor stores the contents of this control in the `discreteOrMinValue` field of the `IviRangeTableEntry` structure. Press <F4> to insert *`Prefix`*`_VAL_` at the beginning of the text in the control. Press <Enter> to display a list of all defined constants in the driver header file that begin with *`Prefix`*`_VAL_`.
- **Maximum Value**—Lets you specify the maximum value for the currently selected range table entry. This control appears only when the **Table Type** is **Coerced** or **Ranged**. The control lets you specify a text entry so that you can enter a defined constant, literal value, or expression. The Attribute Editor stores the contents of this control in the `maxValue` field of the `IviRangeTableEntry` structure. Press <F4> to insert *`Prefix`*`_VAL_` at the beginning of the text in the control. Press <Enter> to display a list of all defined constants in the driver header file that begin with *`Prefix`*`_VAL_`.

- **Coerced Value**—Lets you specify the coerced value for the currently selected range table entry. This control appears only when the **Table Type** is **Coerced**. The control lets you specify a text entry so that you can enter a defined constant, literal value, or expression. The Attribute Editor stores the contents of this control in the `coercedValue` field of the `IviRangeTableEntry` structure. Press <F4> to insert *`Prefix`*`_VAL_` at the beginning of the text in the control. Press <Enter> to display a list of all defined constants in the driver header file that begin with *`Prefix`*`_VAL_`.
- **Discrete Value**—Lets you specify the discrete value for the currently selected range table entry. This control appears only when the **Table Type** is **Discrete**. The control lets you specify a text entry so that you can enter a defined constant, literal value, or expression. The Attribute Editor stores the contents of this control in the `discreteOrMinValue` field of the `IviRangeTableEntry` structure. Press <F4> to insert *`Prefix_VAL_`* at the beginning of the text in the control. Press <Enter> to display a list of all defined constants in the driver header file that begin with *`Prefix_VAL_`*.
- **Actual Value**—Lets you specify the actual numeric value of the expression you enter in the **Coerced Value** or **Discrete Value** control. This control appears only when the **Table Type** is **Coerced** or **Discrete**. Instrument driver users view the actual value in the Select Attribute Constant dialog box that they launch in the `Get/Set/CheckAttribute` function panels. The Attribute Editor stores this value in the `.sub` file and, in some cases, the header file for the driver.
- **Command String**—Lets you specify the command string you use to set the instrument to the value that the currently selected range table entry defines. Enter a string surrounded by double quotes, a defined constant name for a string, an empty string, or `VI_NULL`. The Attribute Editor stores the contents of this control in the `cmdString` field of the `IviRangeTableEntry` structure.
- **Command Value**—Lets you specify the value to write to a register-based device to set the instrument to the value that the currently selected range table entry defines. Enter a literal integer value or a defined constant. The Attribute Editor stores the contents of this control in the `cmdValue` field of the `IviRangeTableEntry` structure. Press <Enter> to add a new row below the current entry in the range table.
- **Help Text**—Contains the help text for the currently selected range table entry. Instrument driver users view the help text in the Select Attribute Constant dialog box that they invoke in the `Get/Set/CheckAttribute` function panels. The Attribute Editor stores the contents of this control in the driver `.sub` file. Press <Enter> to add a new row below the current entry in the list.

Note If you define your range tables directly in the driver source code, you still must use the Edit Range Table dialog box to specify the actual values and help text for each table entry. The Attribute Editor saves this information in the `.sub` file.

5

Instrument Driver Examples

This chapter shows you how to create an IVI driver. The following examples are designed to serve as models for your instrument driver development.

- Example 1—Creating driver files with the Instrument Driver Development Wizard
- Example 2—Editing instrument driver attributes
  - Modifying attributes that the wizard creates
  - Modifying attribute callback functions
  - Deleting attributes that the instrument does not use
- Example 3—Editing high-level instrument driver functions
  - Editing instrument driver functions the wizard creates
  - Deleting instrument driver functions that the instrument does not use
- Example 4—Adding new attributes and functions
- Example 5—Creating the instrument driver documentation
  - Creating the instrument driver `.doc` file
  - Creating Windows Help
- Example 6—Modifying an existing IVI driver to work with a new instrument

Examples 1 through 5 build on each other. Together, they illustrate all the steps to create a complete IVI driver. These examples use the Fluke 45 Digital Multimeter—a GPIB message-based device. For simplicity, these examples implement only a subset of the capabilities of the Fluke 45. The examples show how to create the following functions and attributes.

- All the functions that IVI and VXI*plug&play* require
- The `FL45_Fetch` function
- The `FL45_ConfigureHold` function
- The attributes for the measurement function, hold enable, and hold threshold

You do not always have to start with the procedure that Example 1 illustrates. In many cases, you can modify an existing driver for a similar instrument. Example 6 shows how to use the wizard to generate new instrument driver files from an existing driver.

Example 1—Creating IVI Driver Files with the Instrument Driver Development Wizard

The Instrument Driver Development Wizard creates your driver files. Through a series of panels, the wizard prompts you for the type of driver you want to create, general instrument information, the common operations your driver supports, and the commands and expected responses your instrument uses for common driver operations.

Complete the following steps to use the Instrument Driver Development Wizard to create the driver files:

1. In LabWindows/CVI, select **Tools»Create IVI Instrument Driver** to launch the wizard.
2. Click **Next** on the welcome panel to begin. The Select an Instrument Driver panel appears as shown in Figure 5-1.

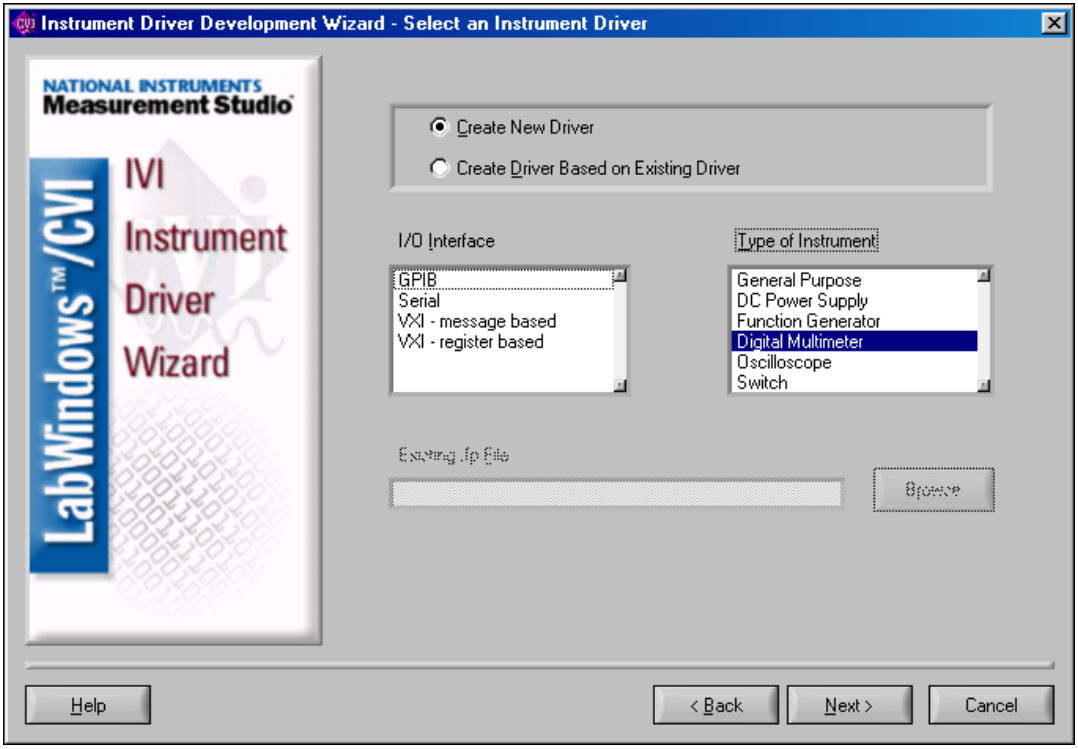

Figure 5-1. Select an Instrument Driver Panel

3. Enter the following information in the wizard panel.
   a. Enable the **Create New Driver** option.
   b. Select GPIB in the **I/O Interface** list box.
   c. Select Digital Multimeter in the **Type of Instrument** list box.
4. Click **Next** to continue. At any time, you can click **Back** to return to a previous panel and change the information.

   The General Information panel appears as shown in Figure 5-2.

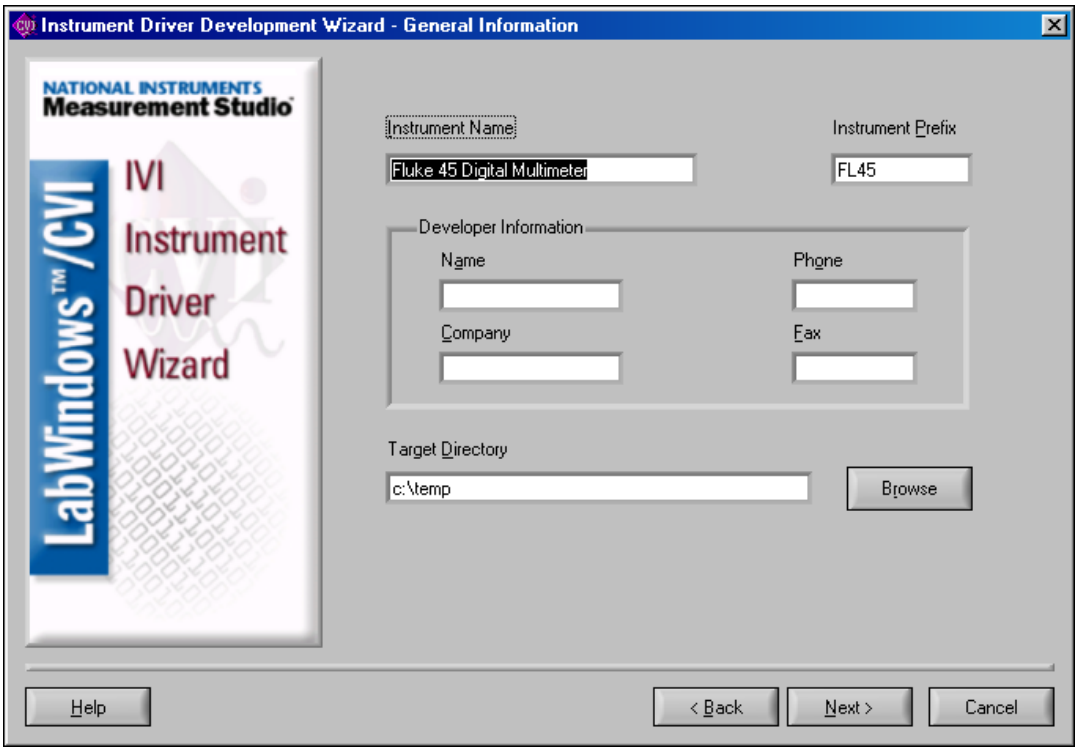

Figure 5-2. General Information Panel

5. Enter the following general instrument driver information.
   a. Enter `Fluke 45 Digital Multimeter` in the **Instrument Name** control.
   b. Enter `FL45` in the **Instrument Prefix** control.
   c. Enter your name, company, phone number, and fax number in the Developer Information section.
   d. Click **Browse** to select a target directory for the new instrument driver.

6. Click **Next** to continue.

   The General Command Strings panel appears as shown in Figure 5-3.

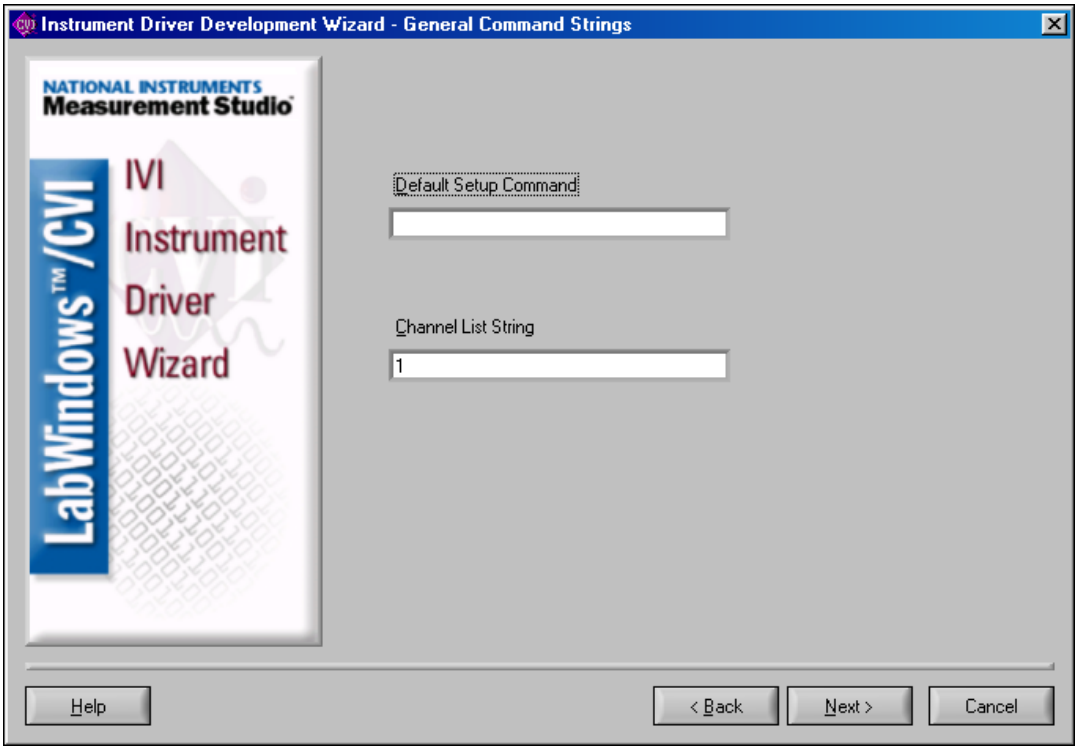

Figure 5-3. General Command Strings Panel

As part of the initialization operation, instrument drivers typically set the instrument to a default state. The default state configures the instrument so that instrument driver functions operate correctly. Specify the default setup command string in the **Default Setup Command** control. The Fluke 45 does not require the instrument driver to send a default setup command string.

7. Delete the contents of the **Default Setup Command** control.

   IVI drivers use channel strings to identify the channels of an instrument. Enter the channel strings you want to use for your instrument in the **Channel List String** control. Use commas to separate multiple channel strings. For a multichannel instrument, you typically use channel strings such as `1`, `2`, `3`, `4`; `A1` through `A4`; or `D0` through `D15`. If your instrument has a front panel, you might want to use the channel names from the front panel.

8. For the Fluke 45 and all other single-channel devices, enter `1` in the **Channel List String** control.

9. Click **Next** to continue.

   The Standard Operations panel appears as shown in Figure 5-4. This panel allows you to select which standard operations your instrument supports.

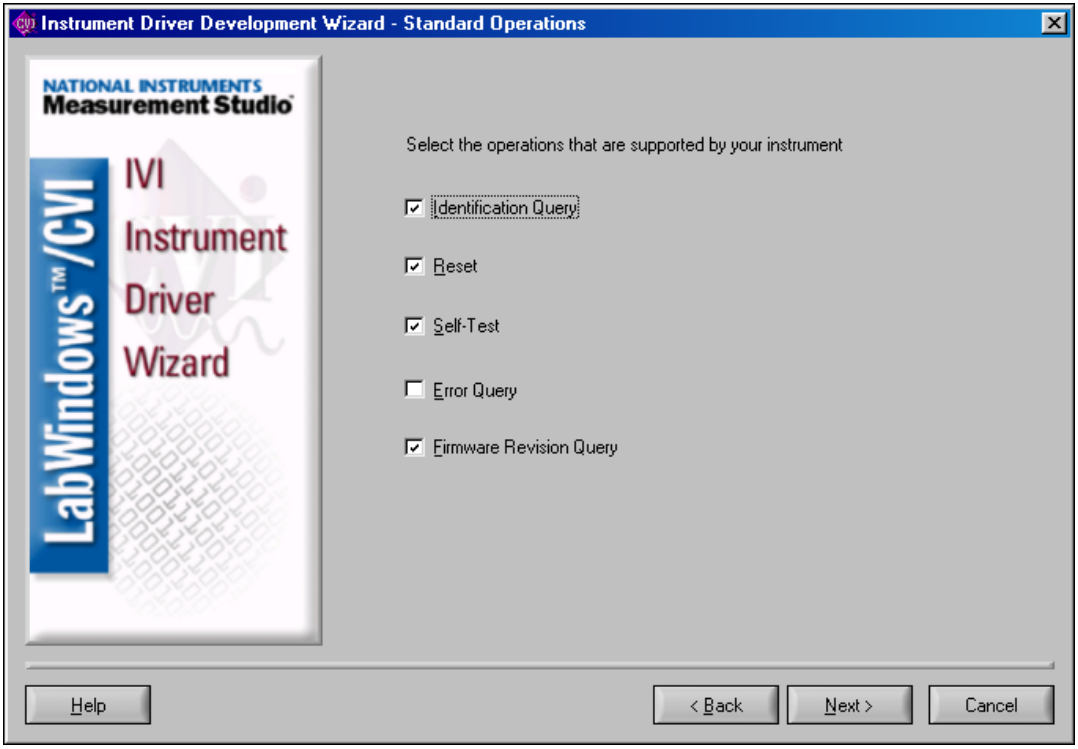

Figure 5-4. Standard Operations Panel

   The Fluke 45 supports the identification query, reset, self-test, and firmware revision query operations, but it does not support an error query operation.

10. Deselect **Error Query**.
11. Click **Next** to continue.

    The following panels prompt you for the commands and response formats that the instrument uses to implement the operations you select.

The ID Query panel appears as shown in Figure 5-5.

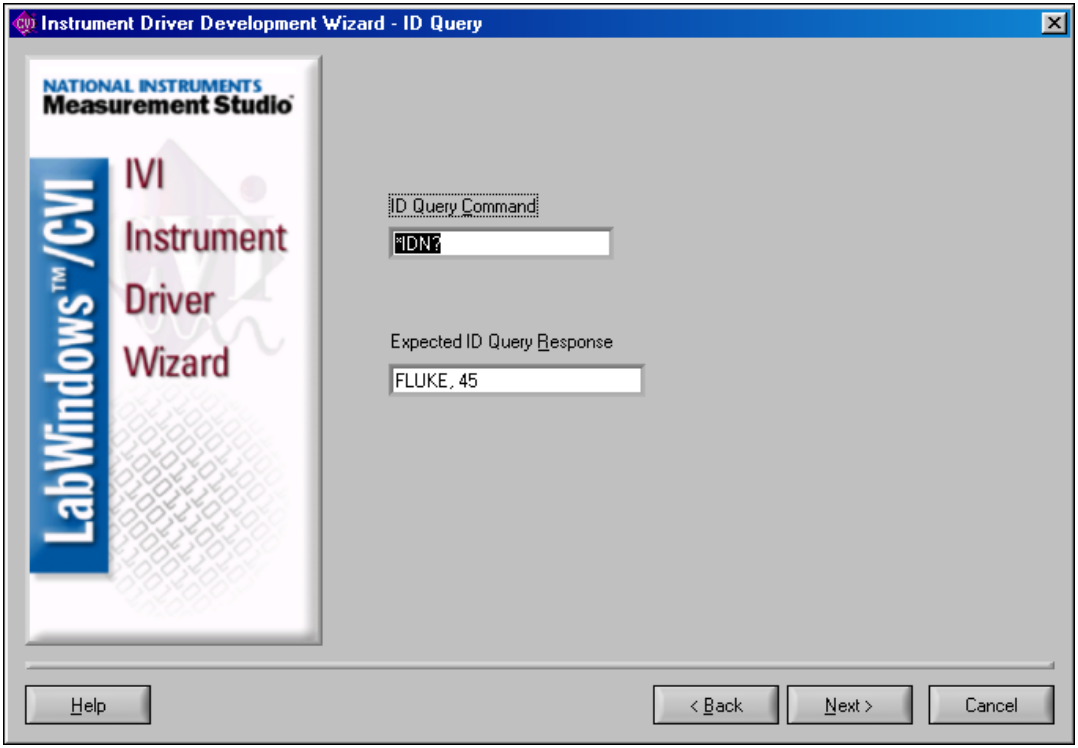

Figure 5-5. ID Query Panel

The `*IDN?` command instructs the Fluke 45 to return its identification string. The driver uses the first portion of this string to determine if it is talking to the correct type of instrument.

12. Leave `*IDN?` in the **ID Query Command** control. This setting is the default.
13. Enter `FLUKE, 45` in the **Expected ID Query Response** control.
14. Click **Next** to continue.

The Reset panel appears as shown in Figure 5-6.

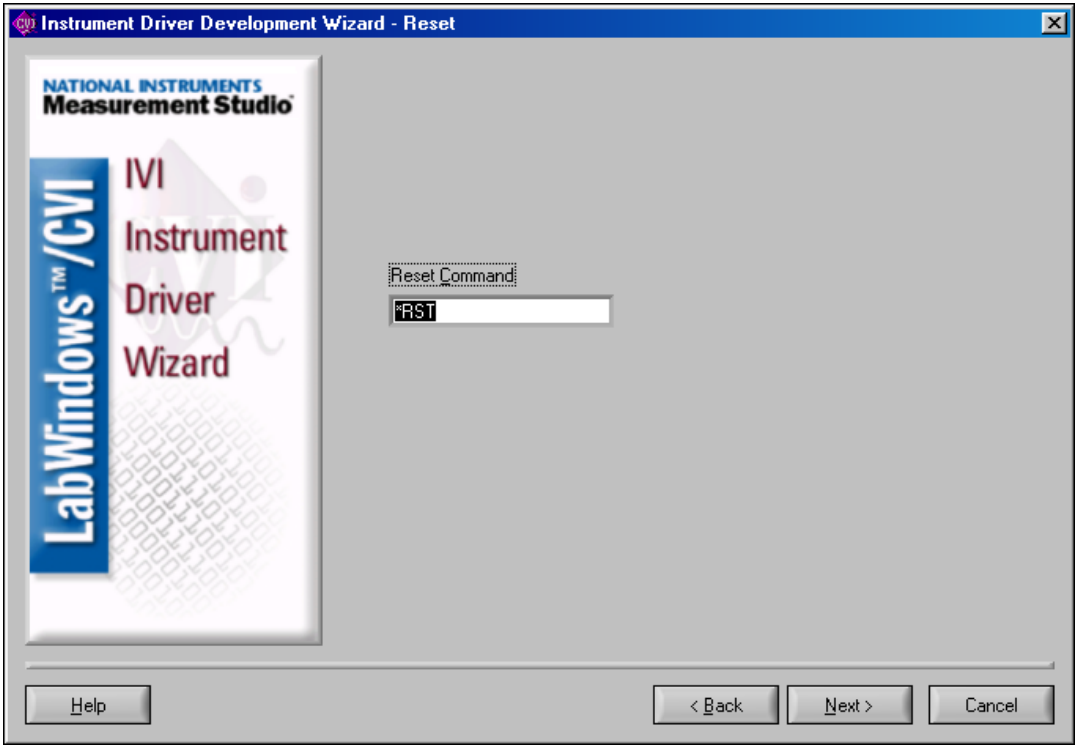

Figure 5-6. Reset Panel

The `*RST` command resets the Fluke 45.

15. Leave `*RST` in the **Reset Command** control. This setting is the default.
16. Click **Next** to continue.

The Self Test panel appears as shown in Figure 5-7.

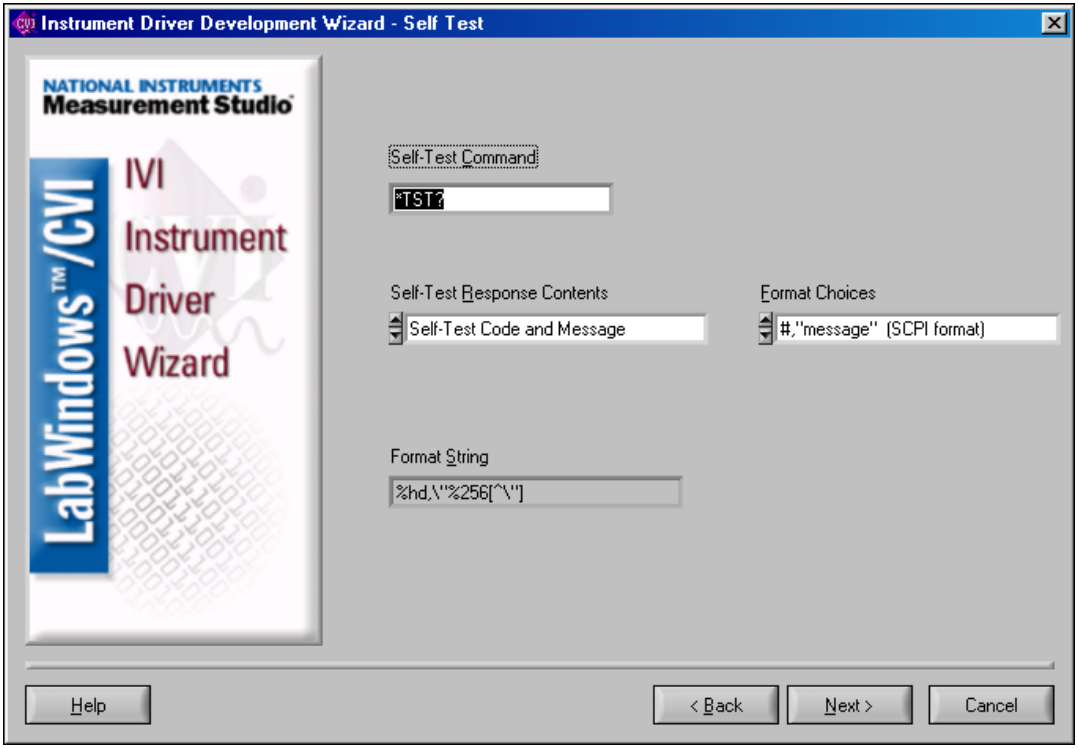

Figure 5-7. Self-Test Panel

The `*TST?` command instructs the Fluke 45 to perform its internal self-test and return the result. The Fluke 45 returns the self-test result as a code.

17. Leave `*TST?` in the **Self-Test Command** control. This setting is the default.
18. Select **Self-Test Code** from the **Self-Test Response Contents** ring control.
19. Select **%hd** from the **Format Choices** ring control.

    The **Format String** indicator displays the format string that VISA uses to interpret the instrument's response. The **%hd** format string is the VISA format specifier for a 16-bit integer.
20. Click **Next** to continue.

The Revision panel appears as shown in Figure 5-8.

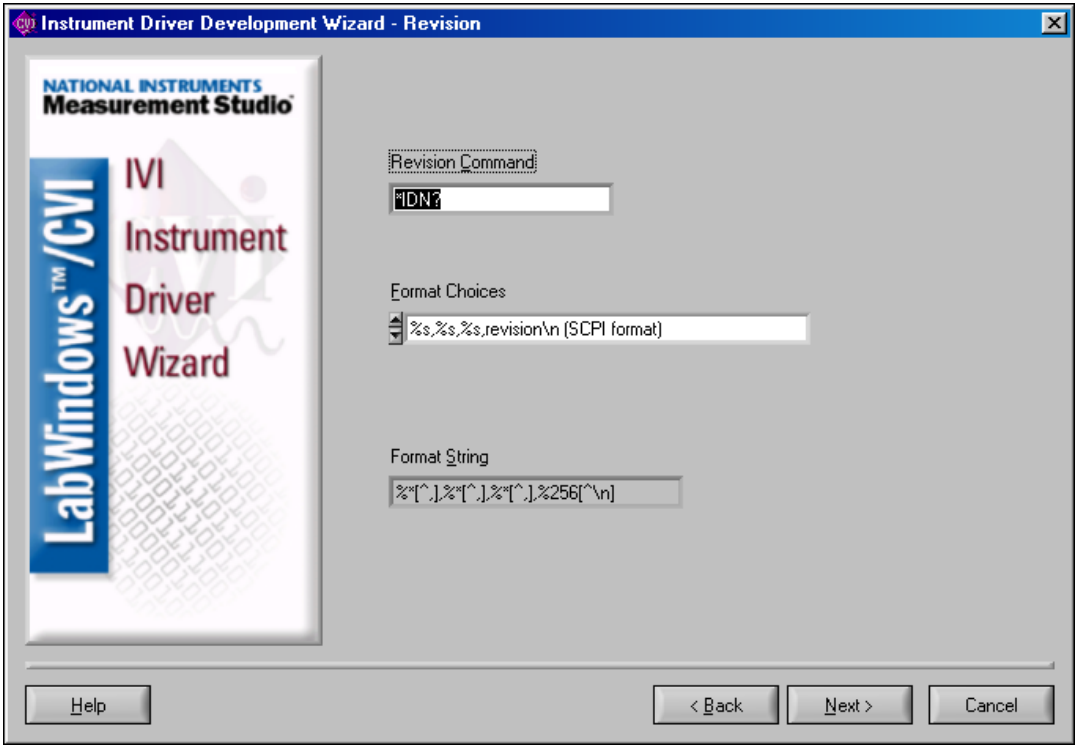

Figure 5-8. Revision Panel

The response to the `*IDN?` command also includes the revision of the Fluke 45 instrument.

21. Leave `*IDN?` in the **Revision Command** control. This setting is the default.
22. Leave **%s,%s,%s,revision\n(SCPI format)** from the **Format Choices** ring control. This setting is the default.

    The **Format String** indicator displays the format string that VISA uses to interpret the instrument's response. This format string instructs VISA to ignore everything in the response up to the third comma and then to read the remainder of the response until it encounters a linefeed.
23. Click **Next** to continue.

The Test panel appears as shown in Figure 5-9.

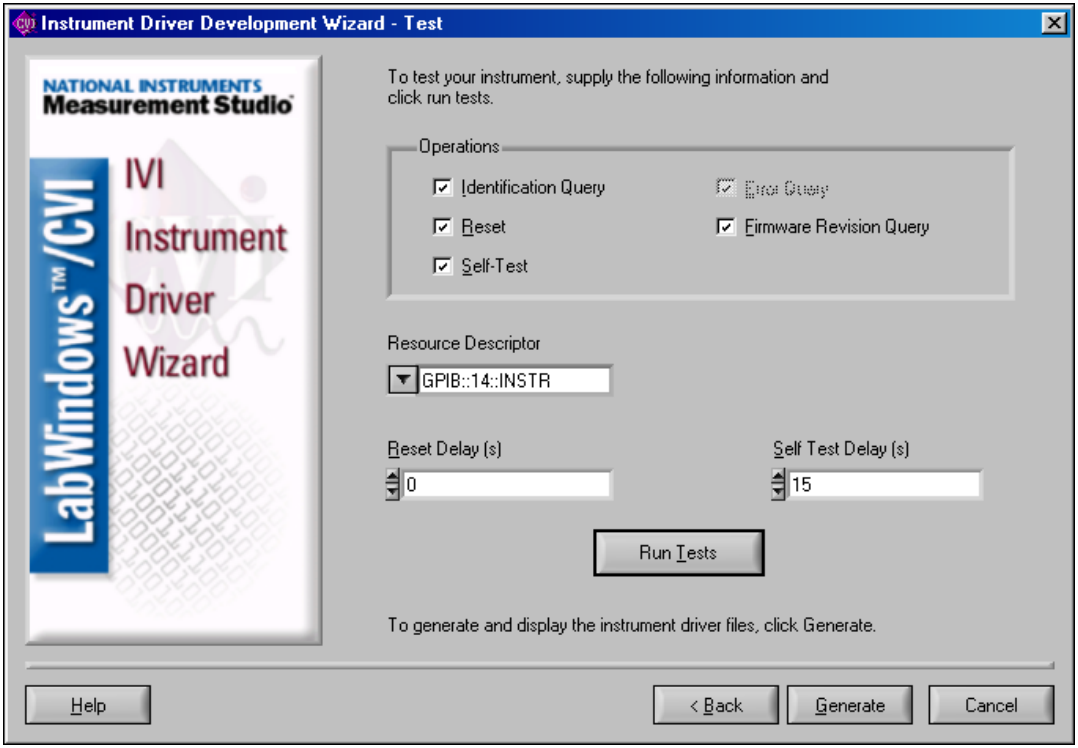

Figure 5-9. Test Panel

If you have a Fluke 45 instrument available and can connect it to the computer, the wizard tests the commands you entered in the previous panels by sending the commands to the instrument and displaying the responses.

24. Run the test as follows:
    a. Select all supported operations to test.
    b. Select the resource descriptor of your instrument in the **Resource Descriptor** control.
    c. Enter `0` in the **Reset Delay (s)** control.
    d. The Fluke 45 takes 15 seconds to perform the self-test operation. Enter `15` in the **Self-Test Delay (s)** control.
    e. Click **Run Tests**.

Figure 5-10 shows the Test Results panel. This panel displays the operations the test performs and the results of these operations.

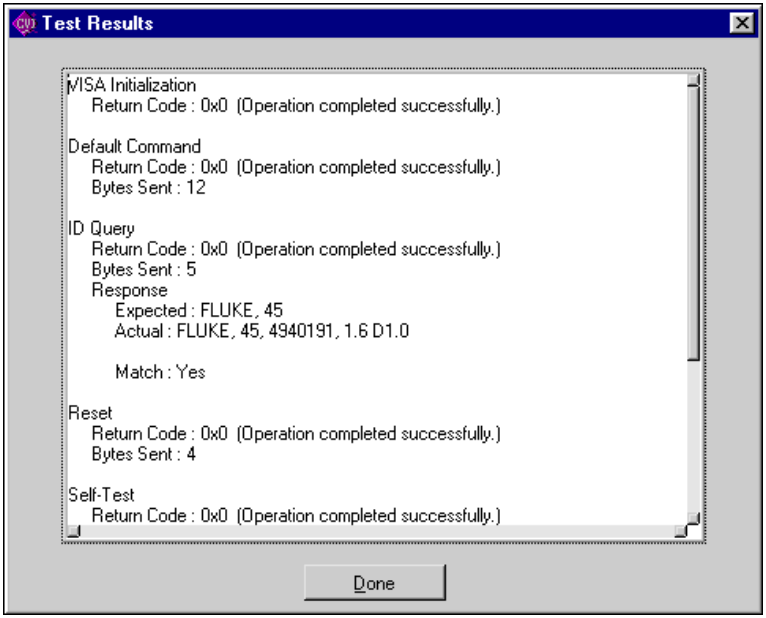

Figure 5-10. IVI Test Results Panel

25. Click **Done** to return to the previous panel.

    If any of the operations generate errors or the responses are not as you expect, you can click **Back** to return to the corresponding panel and change the information. You can then click **Next** to return to the Test panel to verify the new information.

26. When you are satisfied with the test results, click **Generate** on the Test panel. The wizard generates the instrument driver files using the information you provide.

    The driver files are the `fl45.c`, `fl45.h`, `fl45.fp`, and `fl45.sub` files. The resulting driver implements all the functions that IVI and VXI*plug&play* require. These functions are completely operational. Refer to the *LabWindows/CVI Help* for a complete list of the functions that IVI and VXI*plug&play* require. In addition, the driver has all the functions and attributes that are common to DMMs. These functions and attributes have example code with instructions on how to modify the code for a specific DMM.

27. Enable **Launch Attribute Editor** and click **Close** to launch the attribute editor.

Example 2—Editing the Instrument Driver Attributes

You can launch the Attribute Editor at any time by selecting **Tools»Edit Instrument Attributes**. Figure 5-11 shows the attributes that the Instrument Driver Development Wizard created for the Fluke 45 instrument driver.

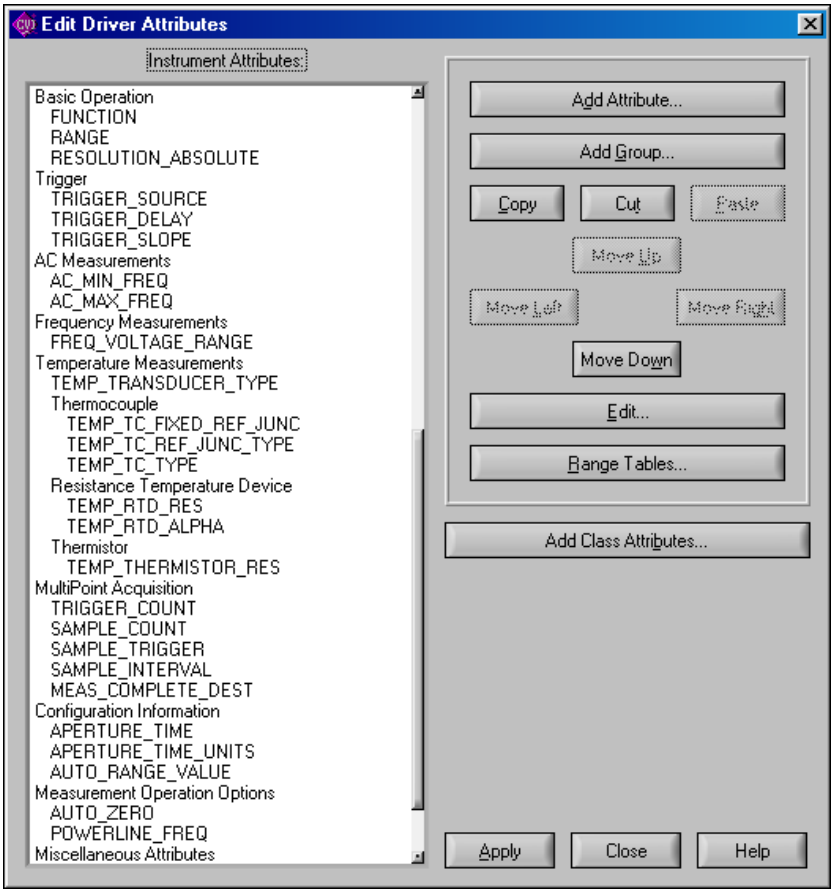

Figure 5-11. Edit Driver Attributes Dialog Box

The Fluke 45 driver you created using the Instrument Driver Development Wizard has attributes that are common to most DMMs. These attributes include basic instrument operations such as setting the measurement function, range, and resolution. The wizard also generates attributes for advanced DMM features such as configuring the trigger count and sample count. The attributes have example implementations for the help information, range tables, and callbacks. Much of the driver design is already done for you.

This example shows how to edit the attributes that the Instrument Driver Development Wizard creates. To complete the attributes for the Fluke 45, you must customize each attribute's example implementation for the Fluke 45, and delete the attributes that the Fluke 45 does not use.

Complete the following steps to customize the measurement function attribute and delete the attributes that the Fluke 45 does not use.

Customizing the Measurement Function Attribute

1. In the Edit Driver Attributes dialog box, double-click the `FUNCTION` attribute. The Edit Attribute dialog box appears as shown in Figure 5-12.

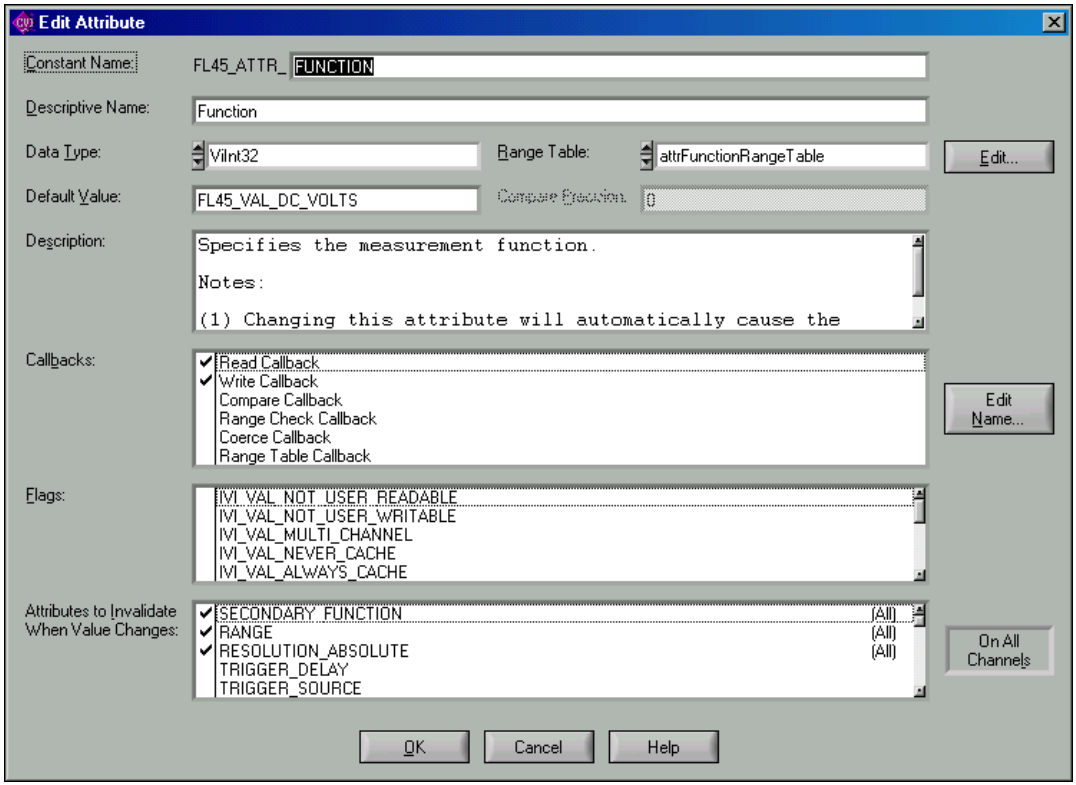

Figure 5-12. Edit Attribute Dialog Box

Notice that the Instrument Driver Development Wizard has already filled in the attribute information.

2. Click **Edit** to edit the range table for the attribute.

The Edit Range Table dialog box appears as shown in Figure 5-13.

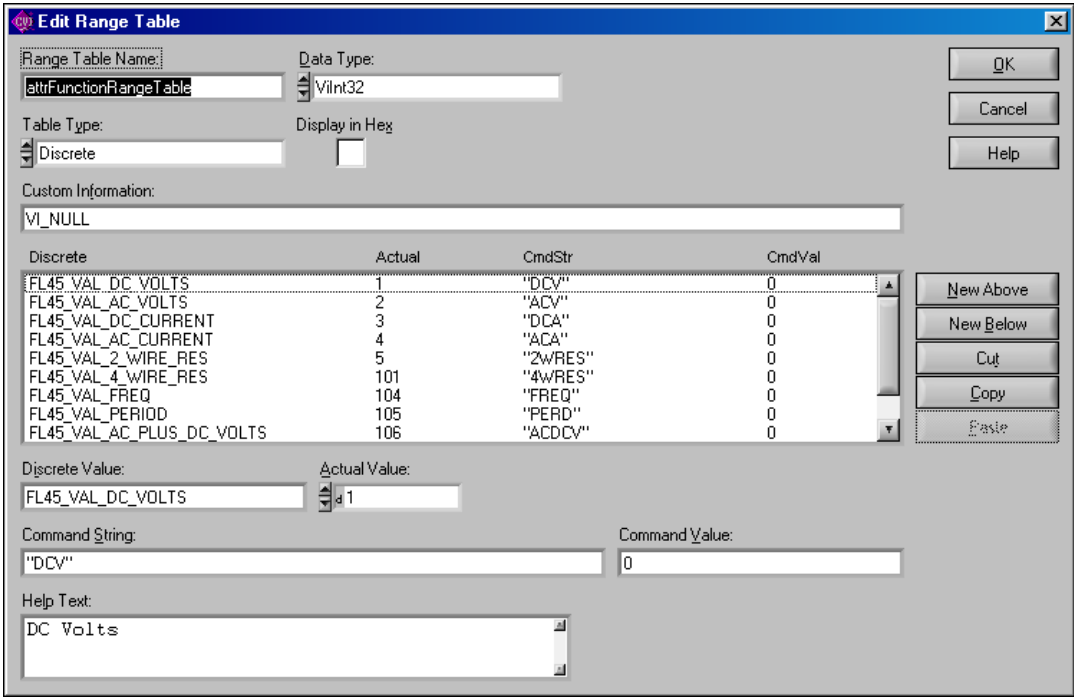

Figure 5-13. Edit Range Table Dialog Box

3. Complete the following steps to modify the range table.
   a. Delete the following range table entries:

      ```
      FL45_VAL_4_WIRE_RES
      FL45_VAL_PERIOD
      FL45_VAL_TEMPERATURE
      ```

      These entries correspond to measurement functions that the Fluke 45 does not support.

   b. Change the contents of the **Command String** control for the remaining entries as shown in Table 5-1.

Table 5-1. Command String Contents

Original Contents	New Contents
FL45_VAL_DC_VOLTS	"VDC"
FL45_VAL_AC_VOLTS	"VAC"

Table 5-1. Command String Contents (Continued)

Original Contents	New Contents
FL45_VAL_DC_CURRENT	"ADC"
FL45_VAL_AC_CURRENT	"AAC"
FL45_VAL_2_WIRE_RES	"OHMS"
FL45_VAL_DIODE	"DIODE"
FL45_VAL_CONTINUITY	"CONT"
FL45_VAL_FREQ	"FREQ"
FL45_VAL_AC_PLUS_DC_VOLTS	"VACDC"
FL45_VAL_AC_PLUS_DC_CURRENT	"AACDC"

After you enter this information, the dialog box appears as shown in Figure 5-14.

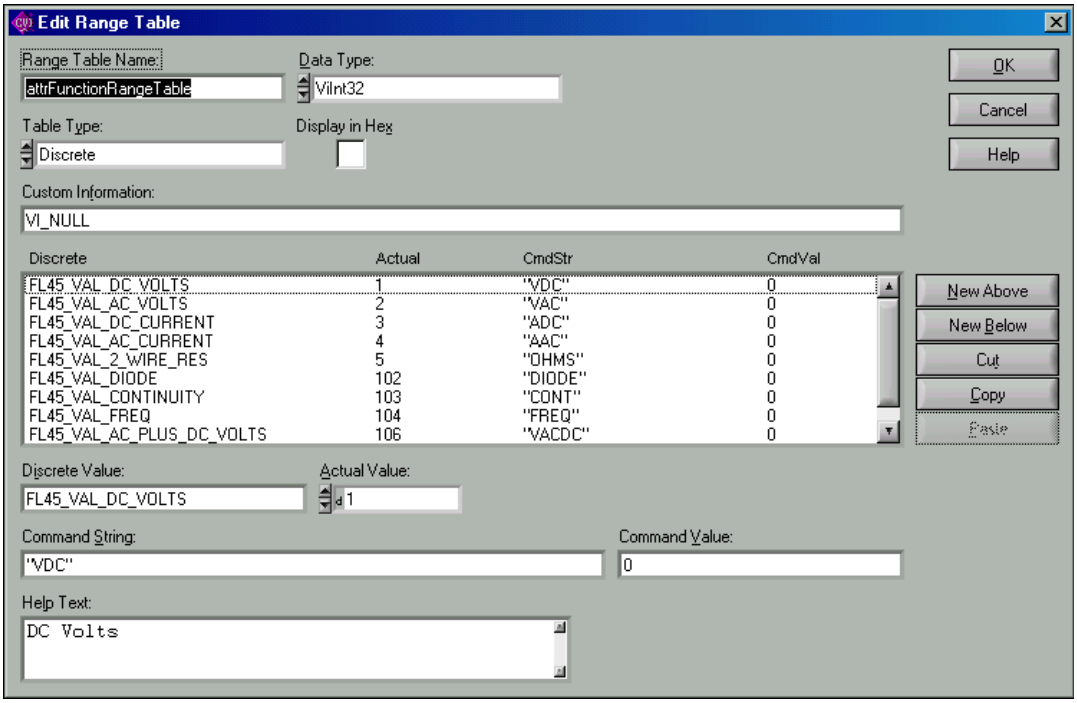

Figure 5-14. Edit Range Table Dialog Box With Modifications

4. Click **OK** to return to the Edit Attribute dialog box.
5. Click **OK** to return to the Edit Driver Attributes dialog box.
6. Click **Apply** to apply the changes to the range table.
7. Click **Close** in the Edit Driver Attributes dialog box to close the attribute editor.

   For each range table entry you delete, you also must delete the corresponding value that the `fl45.h` header file defines.
8. Complete the following steps to delete the unused measurement function values that the `fl45.h` header file defines.
   a. Open the file `fl45.h`.
   b. Delete the following lines in the header file:

```
#define FL45_VAL_PERIOD      IVIDMM_VAL_PERIOD
#define FL45_VAL_4_WIRE_RES  IVIDMM_VAL_4_WIRE_RES
#define FL45_VAL_TEMPERATURE IVIDMM_VAL_TEMPERATURE
```

Modifying the Write and Read Callbacks for the Measurement Function Attribute

1. Select **Tools»Edit Instrument Attributes** to return to the Edit Driver Attributes dialog box.
2. Right-click the `FUNCTION` attribute and select **Go to Write Callback**.

   The **Write Callback** for the `FUNCTION` attribute sends a command string to the instrument to set the measurement function to a specific value. The callback performs the following operations:
   – Uses the range table to look up the instrument-specific command that corresponds to the measurement function the callback receives in the **value** parameter.
   – Writes the command string to the DMM.
3. Enter the following code for the `FL45AttrFunction_WriteCallback` function.

```
static ViStatus _VI_FUNC FL45AttrFunction_WriteCallback
                        (ViSession vi, ViSession io,
                        ViConstString channelName,
                        ViAttr attribute, ViInt32 value)
{
    ViStatus error = VI_SUCCESS;
    ViString cmd;
    checkErr (Ivi_GetViInt32EntryFromValue (value,
                                &attrFunctionRangeTable, VI_NULL,
                                VI_NULL, VI_NULL, VI_NULL, &cmd,
                                VI_NULL));
    viCheckErr (viPrintf (io, "%s;", cmd));
```

```
Error:
        return error;
}
```

4. Select **Tools»Edit Instrument Attributes** to return to the Edit Driver Attributes dialog box.
5. Right-click the `FUNCTION` attribute and select **Go To Read Callback**.

   The **Read Callback** for the `FUNCTION` attribute queries the instrument for the present measurement function setting. The callback performs the following operations:

   a. Sends the `FUNC1?` query to the DMM. This command instructs the Fluke 45 to return the measurement function it is currently using.

   b. Reads the response from the instrument.

   c. Uses the range table to look up the value that corresponds to the string the instrument returns.

   Enter the following code for the `FL45AttrFunction_ReadCallback` function.

```
static ViStatus _VI_FUNC FL45AttrFunction_ReadCallback (ViSession
                        vi, ViSession io, ViConstString
                        channelName, ViAttr attribute, ViInt32
                        *value)
{
    ViStatus error = VI_SUCCESS;
    ViChar   rdBuffer[BUFFER_SIZE];
    ViInt32  rdBufferSize = sizeof(rdBuffer);

        /* Read measurement function from instrument */
    viCheckErr (viPrintf (io, "FUNC1?;"));
    viCheckErr (viScanf (io, "%#s", &rdBufferSize, rdBuffer));

   checkErr (Ivi_GetViInt32EntryFromString (rdBuffer,
                        &attrFunctionRangeTable, value, VI_NULL,
                        VI_NULL, VI_NULL, VI_NULL));

Error:
    return error;
}
```

Deleting Unused Attributes

The Fluke 45 is a simple DMM. It does not use all the attributes that the Instrument Driver Development Wizard creates for DMM instrument drivers. You must delete the attributes that the instrument does not use, the range tables that correspond to the attributes, and the values that the `fl45.h` header file defines for the range tables.

Complete the following steps to delete the attributes that the Fluke 45 does not use:

1. Select **Tools»Edit Instrument Attributes** to invoke the attribute editor.
2. For each of the following attributes, select the attribute and click **Cut**.

   ```
   SAMPLE_COUNT
   SAMPLE_TRIGGER
   SAMPLE_INTERVAL
   TRIGGER_COUNT
   TRIGGER_SLOPE
   MEAS_COMPLETE_DEST
   AUTO_ZERO
   POWERLINE_FREQ
   ```

3. Select **Advanced Triggering** and click **Cut**.
4. Click **Apply** to apply these changes in the instrument driver files.

   The attribute editor asks whether you want to delete the callbacks for each attribute. For each callback, the Generate Code dialog box appears as in Figure 5-15.

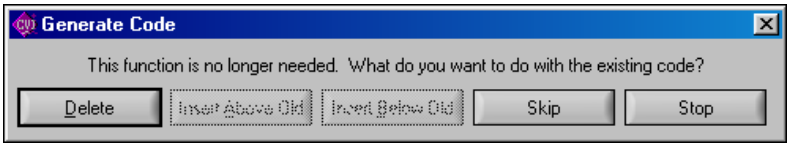

Figure 5-15. Generate Code Dialog Box

5. Click **Delete** for each callback.

Complete the following steps to delete the range tables that correspond to the attributes:

1. From the Edit Driver Attributes dialog box, click **Range Tables**.
2. In the **Range Tables** dialog box, delete each of the following range tables by selecting it in the list box and clicking **Cut**.

   ```
   attrAutoZeroRangeTable
   attrMeasCompleteDestRangeTable
   attrPowerlineFreqRangeTable
   attrSampleCountRangeTable
   attrSampleIntervalRangeTable
   attrSampleTriggerRangeTable
   attrTriggerCountRangeTable
   attrTriggerSlopeRangeTable
   ```

3. Click **OK** to return to the Edit Driver Attributes dialog box.
4. Click **Apply** to apply the range tables changes in the instrument driver files.
5. Click **Close** on the Edit Driver Attributes dialog box to close the attribute editor.

Complete the following steps to delete the defined constants for values that the Fluke 45 does not use:

1. Open the file `fl45.h`.
2. Delete the following sections of defined constants from the header file:

Section 1:

```
/*- Defined values for attribute FL45_ATTR_AUTO_ZER0 -*/

    #define FL45_VAL_AUTO_ZERO_OFF    IVIDMM_VAL_AUTO_ZERO_OFF
    #define FL45_VAL_AUTO_ZERO_ON     IVIDMM_VAL_AUTO_ZERO_ON
    #define FL45_VAL_AUTO_ZERO_ONCE   IVIDMM_VAL_AUTO_ZERO_ONCE
```

Section 2:

```
/*- Defined values for attribute FL45_ATTR_POWERLINE_FREQ -*/

    #define FL45_VAL_50_HERTZ         IVIDMM_VAL_50_HERTZ
    #define FL45_VAL_60_HERTZ         IVIDMM_VAL_60_HERTZ
    #define FL45_VAL_400_HERTZ        IVIDMM_VAL_400_HERTZ
```

Section 3:

```
/* Defined value for attribute FL45_ATTR_SAMPLE_TRIGGER -*/

    /* #define FL45_VAL_IMMEDIATE     DEFINED ABOVE */
    /* #define FL45_VAL_EXTERNAL      DEFINED ABOVE */
    /* #define FL45_VAL_GPIB_GET      DEFINED ABOVE */
    #define FL45_VAL_INTERVAL         IVIDMM_VAL_INTERVAL
    /* #define FL45_VAL_TTL0          DEFINED ABOVE */
    /* #define FL45_VAL_TTL1          DEFINED ABOVE */
    /* #define FL45_VAL_TTL2          DEFINED ABOVE */
    /* #define FL45_VAL_TTL3          DEFINED ABOVE */
    /* #define FL45_VAL_TTL4          DEFINED ABOVE */
    /* #define FL45_VAL_TTL5          DEFINED ABOVE */
    /* #define FL45_VAL_TTL6          DEFINED ABOVE */
    /* #define FL45_VAL_TTL7          DEFINED ABOVE */
    /* #define FL45_VAL_ECL0          DEFINED ABOVE */
    /* #define FL45_VAL_ECL1          DEFINED ABOVE */
    /* #define FL45_VAL_PXI_STAR      DEFINED ABOVE */
```

Section 4:

```
/*- Defined values for attribute FL45_ATTR_TRIGGER_SLOPE -*/

    #define FL45_VAL_POS              IVIDMM_VAL_POS
    #define FL45_VAL_NEG              IVIDMM_VAL_NEG
```

Section 5:

```
/*- Defined values for attribute FL45_ATTR_MEAS_COMPLETE_DEST -*/

    #define FL45_VAL_NONE             IVIDMM_VAL_NONE
    /* #define FL45_VAL_TTL0          DEFINED ABOVE */
    /* #define FL45_VAL_TTL1          DEFINED ABOVE */
    /* #define FL45_VAL_TTL2          DEFINED ABOVE */
    /* #define FL45_VAL_TTL3          DEFINED ABOVE */
    /* #define FL45_VAL_TTL4          DEFINED ABOVE */
    /* #define FL45_VAL_TTL5          DEFINED ABOVE */
    /* #define FL45_VAL_TTL6          DEFINED ABOVE */
    /* #define FL45_VAL_TTL7          DEFINED ABOVE */
    /* #define FL45_VAL_ECL0          DEFINED ABOVE */
    /* #define FL45_VAL_ECL1          DEFINED ABOVE */
    /* #define FL45_VAL_PXI_STAR      DEFINED ABOVE */
```

Example 3—Editing High-Level Instrument Driver Functions

This example shows how to edit the high-level instrument driver functions that the Instrument Driver Development Wizard creates. This example shows how to modify the `Fetch` function for the Fluke 45 and delete the functions that the Fluke 45 does not support.

Editing the Fetch Function

Complete the following steps to modify the `Fetch` function for the Fluke 45 and delete the functions that the Fluke 45 does not support.

1. Select **File»Open»Function Tree (*.fp)** to open the `fl45.fp` function panel file. The function tree appears as shown in Figure 5-16.

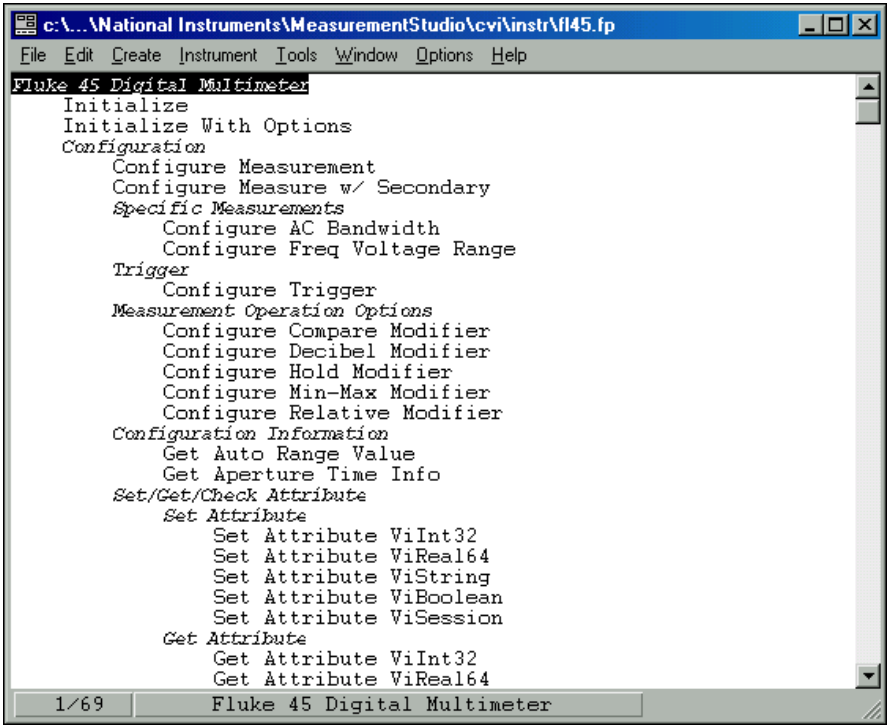

Figure 5-16. Fluke 45 Function Tree

2. Scroll down to the Fetch function. Right-click the `Fetch` function to display the context menu as shown in Figure 5-17.

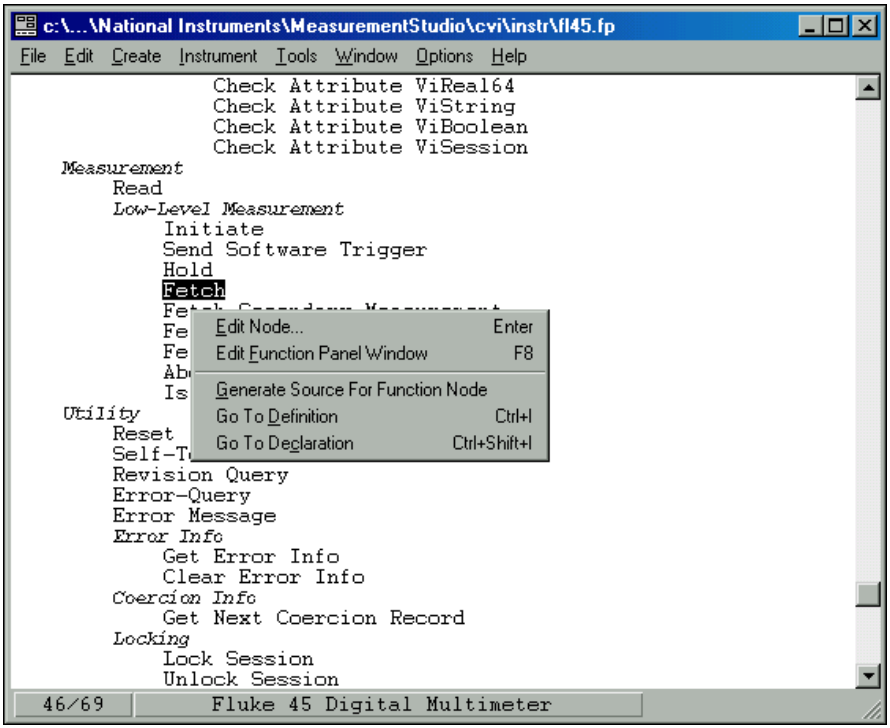

Figure 5-17. Function Tree Editor Context Menu

3. Select **Go To Definition** from the context menu to go to the `FL45_Fetch` function definition in the `fl45.c` file.

Notice that the Instrument Driver Development Wizard has already created the `FL45_Fetch` function. The function contains the code for a typical implementation. The function also contains instructions on how to modify the instrument-specific segments of code. In general, the modification instructions appear within comments that start with `CHANGE` and end with `END CHANGE`, in the `FL45_Fetch` function. The modification instructions include explanations and sample source code.

Complete the following steps to modify the code for the `FL45_Fetch` function.

1. Change the command string in the `viPrintf` statement to `VAL1?`.
2. Change the `if` statement that tests for over-range to the following:

```
if ((reading == 1000000000) || (reading == -1000000000))
```

3. Change the comments that start with a double-slash (//) to the traditional C comment style (/*...*/).
4. Delete the CHANGE and END CHANGE comment lines and the explanation text from the modification instructions.

The code appears as follows:

```
/*****************************************************************
 * Function:   FL45_Fetch
 * Purpose:    This function returns the measured value from a
 *             previously initiated measurement.  This function does
 *             not trigger the instrument.
 *
 *             After this function executes, the value in *readingRef
 *             is an actual reading or a value indicating that an
 *             over-range condition occurred.  If an over-range
 *             condition occurs, the function sets *readingRef to
 *             FL45_VAL_OVER_RANGE_READING and returns
 *             FL45_WARN_OVER_RANGE.
*****************************************************************/

ViStatus _VI_FUNC FL45_Fetch (ViSession vi, ViInt32 maxTime,
                              ViReal64 *readingRef)
    {
    ViStatus  error = VI_SUCCESS;
    ViReal64  reading;
    ViBoolean overRange = VI_FALSE;
    ViSession io = VI_NULL;
    ViUInt32  oldTimeout;
    ViBoolean needToRestoreTimeout = VI_FALSE;

    checkErr (Ivi_LockSession (vi, VI_NULL));

    if (readingRef == VI_NULL)

        viCheckParm( IVI_ERROR_INVALID_PARAMETER, 3,
                              "Null address for Reading");

    if (!Ivi_Simulating (vi))
        {
        io = Ivi_IOSession (vi);

        checkErr( Ivi_SetNeedToCheckStatus (vi, VI_TRUE));

            /* Store the old timeout so that it can be restored later */

        viCheckErr (viGetAttribute (io, VI_ATTR_TMO_VALUE,
                              &oldTimeout)26);
        viCheckErr (viSetAttribute (io, VI_ATTR_TMO_VALUE, maxTime));
        needToRestoreTimeout = VI_TRUE;
```

```
        viCheckErr (viPrintf (io, "VAL1?;"));
        error = ( viscanf (io, “%lf”, &reading));

        if (error == VI_ERROR_TMO)
           error = FL45_ERROR_MAX_TIME_EXCEEDED;
        viCheckErr (error);

           /* Test for over-range */

        if ((reading == 1000000000) || (reading == -1000000000))
           {
           *readingRef = IVIDMM_VAL_OVER_RANGE_READING;
           overRange = VI_TRUE;
           }
        else
           {
           *readingRef = reading;
           }
        }
     else
        {
        ViReal64    range;

        checkErr (Ivi_GetAttributeViReal64 (vi, VI_NULL,
                            FL45_ATTR_RANGE, 0, &range));
        if (range <= 0.0)    /* If auto-ranging, use the max value. */

           checkErr (Ivi_GetAttrMinMaxViReal64 (vi, VI_NULL,
                            FL45_ATTR_RANGE, VI_NULL, &range, VI_NULL,
                            VI_NULL));

        *readingRef = range * ((ViReal64)rand() / (ViReal64)RAND_MAX);
        }
        /*
           Do not invoke FL45_CheckStatus here. FL45_Read invokes
           FL45_CheckStatus after it calls this function.  After the
           user calls this function, the user can check for errors by
           calling FL45_error_query.
        */
  Error:
     if (needToRestoreTimeout)
        {
           /* Restore the original timeout */
        viSetAttribute (io, VI_ATTR_TMO_VALUE, oldTimeout);
        }
     Ivi_UnlockSession (vi, VI_NULL);
     if (overRange && (error >= VI_SUCCESS))
        return FL45_WARN_OVER_RANGE;
```

```
else
   return error;
}
```

Deleting Functions the Instrument Does Not Use

The Fluke 45 does not support the multipoint operation, as shown in Table 5-2.

Table 5-2. Multipoint Operations

Function	Function Panel Name
`FL45_ReadMultiPoint`	Read Multipoint
`FL45_FetchMultiPoint`	Fetch Multipoint
`FL45_ConfigureMultiPoint`	Configure Multipoint

For each of these operations, you must delete the corresponding function definition in the source file, delete the corresponding function declaration in the header file, and delete the corresponding function panel in the function panel file.

Complete the following steps to delete the functions:

1. Open the function tree for the `fl45.fp` function panel file.
2. Right-click the `Configure Multipoint` function to display the context menu. Select **Go To Declaration** to go to the function's declaration in the `fl45.h` file.
3. Delete the declaration.
4. In the Source window, select **Edit Function Tree** from the context menu to return to the Function Tree Editor.
5. Right-click the `Configure Multipoint` function and select **Go To Definition** from the context menu to go to the function's definition in the `fl45.c` file.
6. Delete the entire function body.
7. In the Source window, select **Edit Function Tree** from the context menu to return to the Function Tree Editor.
8. Select the `Configure Multipoint` function. Select **Edit»Cut** to delete the function panel.
9. For each of the remaining functions, you can either repeat the steps above or make all the edits in each file at once.

Example 4—Adding New Attributes and Functions

This example shows how to add new attributes and functions to your instrument driver as follows:

- A Hold Enable attribute that selects whether to enable the hold capability.
- A Hold Threshold attribute that specifies how stable the signal must be for the Fluke 45 to take a new measurement.
- A high-level `Configure Hold` function.

The hold feature enables the Fluke 45 to take a new measurement only when it detects a stable input signal.

Adding the Hold Enable Attribute

Complete the following steps to add the Hold Enable attribute.

1. Select **Tools»Edit Instrument Attributes** to launch the attribute editor.
2. Select `Configuration Information`.
3. Click **Add Group** to create a group for the new attributes.
4. Enter the following information in the Edit Group dialog box:
   a. Enter `Hold Modifier` in the **Name** control.
   b. Enter the following text in the **Description** control:

   ```
   This group contains attributes that control the DMM's hold
   modifier. The hold modifier configures the DMM to take a new
   measurement only when the input signal is stable and 'hold' that
   measurement on the display. This feature can be particularly
   advantageous in difficult or hazardous circumstances when you
   might want to keep your eyes fixed on the probes, and then read
   the display when it is safe or convenient to do so.
   ```
5. Click **OK** to return to the Edit Driver Attributes dialog box. Click **Add Attribute** to create a new attribute. Enter the following information in the Edit Attribute dialog box:
   a. Enter `HOLD_ENABLE` in the **Constant Name** control.
   b. Enter `Hold Enable` in the **Descriptive Name** control.
   c. Select `ViBoolean` from the **Data Type** ring control.
   d. Enter `VI_FALSE` in the **Default Value** control.

e. Enter the following text in the **Description** control:

```
Specifies whether to enable the hold function modifier. This
modifier configures the DMM to take a new measurement only when
the input signal is stable and 'hold' that measurement on the
display.
```

f. Click in the left margin of the **Callbacks** list box control next to the Read Callback and Write Callback entries to create read and write callbacks for the attribute.

6. Click **OK** to return to the Edit Driver Attributes dialog box.
7. Click **Apply** to apply the changes.
8. Right-click the `HOLD_ENABLE` attribute and select **Go To Write Callback**.
9. Enter the following code for the `FL45AttrHoldEnable_WriteCallback` function:

```
static ViStatus _VI_FUNC FL45AttrHoldEnable_WriteCallback
                        (ViSession vi, ViSession io,
                        ViConstString channelName,
                        ViAttr attributeId, ViBoolean value)
{
   ViStatus error = VI_SUCCESS;

   if (value)
      viCheckErr (viPrintf (io, "HOLD;"));
   else
      viCheckErr (viPrintf (io, "HOLDCLR;"));

Error:
   return error;
}
```

10. Select **Tools»Edit Instrument Attributes** to return to the Edit Driver Attributes dialog box.
11. Right-click the `HOLD_ENABLE` attribute and select **Go To Read Callback**.
12. Enter the following code for the `FL45AttrHoldEnable_ReadCallback` function.

```
static ViStatus _VI_FUNC FL45AttrHoldEnable_ReadCallback
                        (ViSession vi, ViSession io,
                        ViConstString channelName,
                        ViAttr attributeId, ViBoolean *value)
{
   ViStatus error = VI_SUCCESS;
   ViInt32 modebyte;

   viCheckErr (viPrintf (io, "MOD?;"));
   viCheckErr (viScanf (io, "%ld", &modebyte));

   if (modebyte & 0x04)
      *value = VI_TRUE;
```

```
    else
        *value = VI_FALSE;
Error:
    return error;
}
```

Adding the Hold Threshold Attribute

Complete the following steps to add the Hold Threshold Attribute.

1. Select **Tools»Edit Instrument Attributes** to launch the attribute editor.
2. Select the `HOLD_ENABLE` attribute.
3. Click **Add Attribute** to create a new attribute.
4. Enter the following information in the Edit Attribute dialog box:
   a. Enter `HOLD_THRESHOLD` in the **Constant Name** control.
   b. Enter `Hold Threshold` in the **Descriptive Name** control.
   c. Select `ViInt32` from the **Data Type** ring control.
   d. Click **New** to create a range table for the attribute.
5. Enter the following information for the range table:
   a. Enter attr`holdThresholdRangeTable` in the **Range Table Name** control.
   b. Select `ViInt32` from the **Data Type** ring control.
   c. Select `Discrete` from the **Table Type** ring control.
   d. Enter the information in Table 5-3 for the range table entries.

Table 5-3. Range Table Entry Information

Discrete	Actual	CmdStr	Help Text
FL45_VAL_HOLD_VERY_STABLE	0	"1"	Very Stable Input (5% of range)
FL45_VAL_HOLD_STABLE	1	"2"	Stable Input (7% of range)
FL45_VAL_HOLD_NOISY	2	"3"	Noisy Input (8% of range)

Note You can insert `FL45_VAL_` into the **Discrete Value** control automatically by placing your cursor in the control and pressing <F4>.

6. Click **OK** to return to the Edit Attribute dialog box.

7. Complete the dialog box with the following information:
   a. Enter `FL45_VAL_HOLD_STABLE` in the **Default Value** control.
   b. Enter the following text in the **Description** control:

      `This attribute specifies the hold measurement threshold. The DMM takes a new measurement only when the input signal is as stable as you specify with this attribute.`

      `This attribute affects instrument behavior only when you set the FL45_ATTR_HOLD_ENABLE attribute to VI_TRUE.`
   c. Click in the left margin of the **Callbacks** list box control next to the Read Callback and Write Callback entries to create read and write callbacks for the attribute.
8. Click **OK** to return to the Edit Driver Attributes dialog box.
9. Click **Apply** to apply the changes.
10. Right-click the `HOLD_THRESHOLD` attribute and select **Go To Write Callback**.
11. Enter the following code for the `FL45AttrHoldThreshold_WriteCallback` function:

```
static ViStatus _VI_FUNC FL45AttrHoldThreshold_WriteCallback
                        (ViSession vi, ViSession io,
                        ViConstString channelName,
                        ViAttr attributeId, ViInt32 value)
{
    ViStatus error = VI_SUCCESS;
    ViString cmd;
   checkErr (Ivi_GetViInt32EntryFromValue (value,
                        &holdThresholdRangeTable, VI_NULL,
                        VI_NULL, VI_NULL, VI_NULL, &cmd,
                        VI_NULL));
        viCheckErr (viPrintf ( io, "HOLDTHRESH %s;",cmd));
Error:
    return error;
}
```

12. Select **Tools»Edit Instrument Attributes** to return to the Edit Driver Attributes dialog box.
13. Right-click the `HOLD_THRESHOLD` attribute and select **Go To Read Callback**.
14. Enter the following code for the `FL45AttrHoldThreshold_ReadCallback` function.

```
static ViStatus _VI_FUNC FL45AttrHoldThreshold_ReadCallback
                        (ViSession vi, ViSession io,
                        ViConstString channelName,
                        ViAttr attributeId, ViInt32 *value)
```

```
{
    ViStatus error = VI_SUCCESS;
    ViChar rdbuffer[5];

    viCheckErr (viPrintf ( io, "HOLDTHRESH?;"));
    viCheckErr (viScanf ( io, "%s", rdbuffer));

   checkErr (Ivi_GetViInt32EntryFromString (rdbuffer,
                        &holdThresholdRangeTable, value, VI_NULL,
                        VI_NULL, VI_NULL, VI_NULL));

Error:
    return error;
}
```

Adding the Configure Hold Function Panel

Complete the following steps to add the Configure Hold function panel:

1. Select **File»Open** to open the `fl45.fp` function panel file.
2. Select the `Configure Freq Voltage Range` function. Select **Create»Function Panel Window** and enter the following information:
   a. Enter `Configure Hold Modifier` in the **Name** control.
   b. Enter `ConfigureHold` in the **Function Name** control.
3. Click **OK** to return to the Function Tree Editor.
4. Complete the following steps to edit the function panel.
   a. Select the `Configure Hold Modifier` function and select **Edit»Edit Function Panel Window**.
   b. Select **Function Help** from the **Edit** menu.
   c. Enter the following help text:

      ```
      This function configures the DMM's hold modifier capability.
      The hold modifier configures the DMM to take a new measurement
      only when the input signal is stable and 'hold' that measurement
      on the display. This feature can be particularly advantageous
      in difficult or hazardous circumstances when you might want to
      keep your eyes fixed on the probes, and then read the display
      when it is safe or convenient to do so.
      ```

   d. Select **File»Close** in the Help Editor.
5. Complete the following steps to add an Instrument Handle control to the function panel.
   a. Press <Ctrl-Page Up> to display the Configure Freq Voltage Range function panel.
   b. Select the **Instrument Handle** control.
   c. Select **Edit»Copy Controls**.

d. Press <Ctrl-Page Down> to display the Configure Hold Modifier function panel.

e. Select **Edit»Paste** to place a copy of the **Instrument Handle** control on the Configure Hold Modifier panel.

f. Position the **Instrument Handle** control in the lower left corner of the panel.

6. Complete the following steps to add a control to specify whether to enable the Hold Modifier:

a. Select **Create»Binary**.

b. Complete the Create Binary Control and Edit On/Off Settings dialog boxes as shown in Figures 5-18 and 5-19.

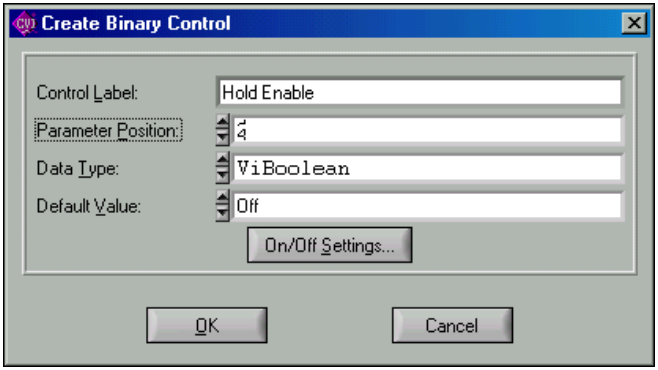

Figure 5-18. Create Binary Control Dialog Box

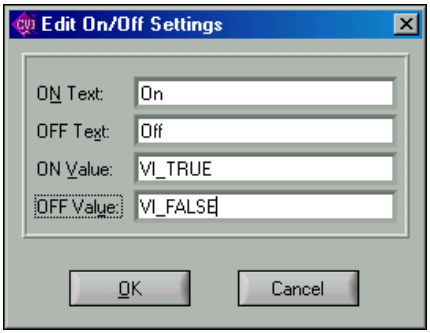

Figure 5-19. Edit On/Off Setting Dialog Box

c. Click **OK** twice to return to the function panel window. Position the **Hold Enable** control in the upper left portion of the panel.

7. Complete the following steps to add help to the **Hold Enable** control:
   a. Select the **Hold Enable** control and select **Edit»Control Help**.
   b. Enter the following text in the Edit Help dialog box:

```
Specify whether you want to enable the hold modifier. Setting
this parameter to VI_TRUE configures the DMM to take a new
measurement only when the input signal is stable and 'hold' that
measurement on the display. The value you specify in the Hold
Threshold parameter determines how stable the signal must be
for the DMM to take a measurement.

Valid Values: VI_TRUE - Enables the hold modifier
              VI_FALSE - Disables the hold modifier

Default Value:VI_FALSE
```

   c. Select **File»Close** to return to the function panel window.
8. Complete the following steps to add a control to specify the hold threshold.
   a. Select **Create»Slide**.
   b. Complete the Edit Slide Control dialog box as shown in Figure 5-20.

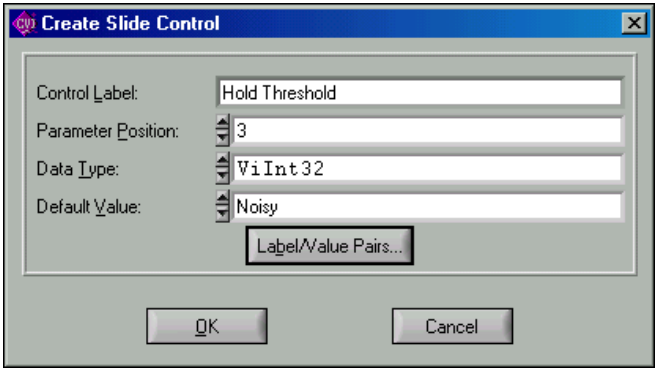

Figure 5-20. Edit Slide Control Dialog Box

c. Click **Label/Value Pairs** and complete the Edit Label/Value Pairs dialog box as shown in Figure 5-21.

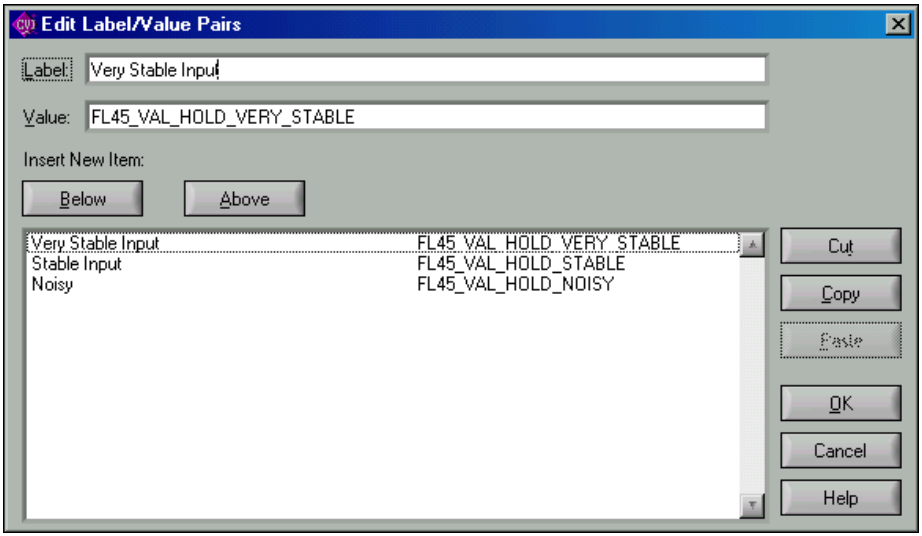

Figure 5-21. Edit Label/Value Pairs Dialog Box

d. Return to the Function Panel Editor. Position the **Hold Threshold** control in the upper right portion of the panel.

9. Complete the following steps to add help to the **Hold Threshold** control:

a. Select the **Hold Threshold** control and select **Edit»Control Help**.

b. Enter the following text in the Help Editor dialog box:

```
Pass the hold threshold you want the DMM to use. The DMM takes
a new measurement when the input signal is as stable as you
specify with this parameter.

This parameter affects instrument behavior only when you set
the Hold Enable parameter to VI_TRUE.

Valid Values:
   FL45_VAL_HOLD_VERY_STABLE (0) - 5% of range
   FL45_VAL_HOLD_STABLE      (1) - 7% of range
   FL45_VAL_HOLD_NOISY       (2) - 8% of range

Default Value:  FL45_VAL_HOLD_NOISY
```

c. Select **File»Close** in the Help editor.

10. Complete the following steps to add a return control to indicate the status of the function:
    a. Press <Ctrl-Page Up> to display the Configure Freq Voltage Range function panel.
    b. Select the **Status** control.
    c. Select **Edit»Copy Controls**.
    d. Press <Ctrl-Page Down> to display the Configure Hold Modifier function panel.
    e. Select **Edit»Paste** to place a copy of the **Status** control on the Configure Hold Modifier panel.
    f. Position the **Status** control in the lower right corner of the panel.
11. Select the **Status** return value and select **Edit»Control Help** and modify the text to add help to the **Status** control.

    The Configure Hold Modifier function panel window now appears as shown in Figure 5-22.

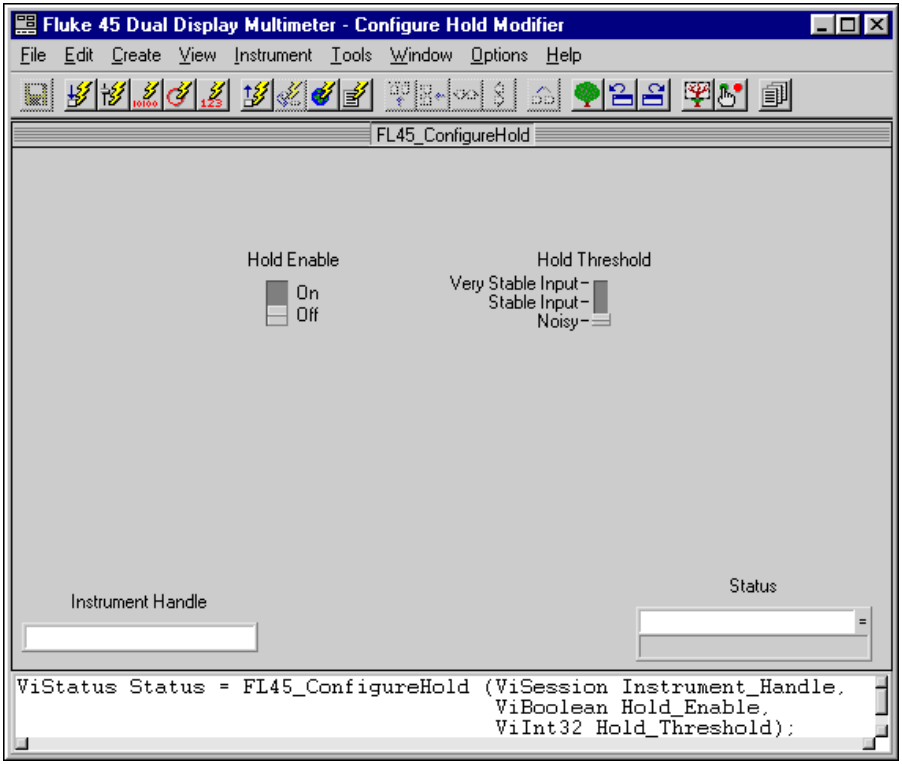

Figure 5-22. Configure Hold Function Panel Window

Creating the Configure Hold Function Body

Complete the following steps to create the Configure Hold Function Body:

1. Select **File»Open»Function Tree (*.fp)** to open the `fl45.fp` function panel file.
2. Right-click the `Configure Hold Modifier` function to display the context menu.
3. Choose the **Generate Source For Function Node** command in the context menu to create the function prototype in the `fl45.h` file and the function definition in the `fl45.c` file.
4. Right-click the `Configure Hold` function and select **Go To Definition** from the context menu to go to the function's definition in the `fl45.c` file.
5. Enter the following code for the `Configure Hold` function.

```
/*****************************************************************
 * Function:   FL45_ConfigureHold
 * Purpose:    Configures the DMM's hold capability.
*****************************************************************/

ViStatus _VI_FUNC FL45_ConfigureHold (ViSession vi, ViBoolean
                          holdEnable, ViInt32 holdThreshold)

{
   ViStatus error = VI_SUCCESS;

   checkErr (Ivi_LockSession (vi, VI_NULL));

   viCheckParm (Ivi_SetAttributeViBoolean (vi, VI_NULL,
                          FL45_ATTR_HOLD_ENABLE, 0, holdEnable), 2,
                          "Hold Enable");

   if (holdEnable)

      viCheckParm (Ivi_SetAttributeViInt32 (vi, VI_NULL,
                              FL45_ATTR_HOLD_THRESHOLD, 0,
                              holdThreshold), 3, "Hold Threshold");

   checkErr (FL45_CheckStatus (vi));

Error:
   Ivi_UnlockSession (vi, VI_NULL);
   return error;
```

Example 5—Creating Instrument Driver Documentation

LabWindows/CVI creates two types of documentation for your instrument driver—a text `.doc` file and Windows Help. This example shows how to generate both types of documentation files for your instrument driver.

Creating the Instrument Driver .doc File

Complete the following steps to create the Instrument Driver `.doc` file:

1. Open the `fl45.fp` file and edit the function tree.
2. Select **Options»Generate Documentation**.

   The Generate Documentation dialog box lets you choose the programming language for which you want to generate documentation.
3. Select the language you want and click **OK**.

   The instrument driver documentation appears as shown in Figure 5-23.

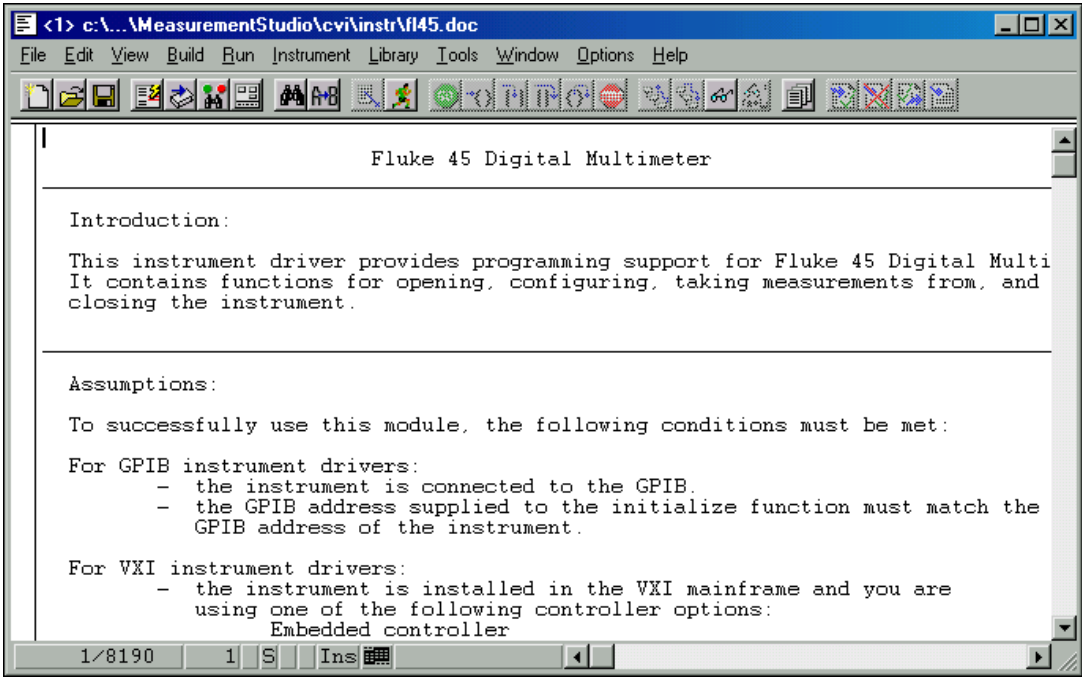

Figure 5-23. Fluke 45 Instrument Driver Documentation Window

4. Select **File»Save** and save the file as `fl45.doc`.

Creating Windows Help for the Driver

Complete the following steps to create Windows Help for your driver.

1. Open the file `fl45.fp` and edit the function tree.
2. Select **Options»Generate Windows Help**.
3. Select the programming language for which you want to generate the Windows help from the Generate Windows Help dialog box.
4. Click **OK**. A message appears as shown in Figure 5-24.

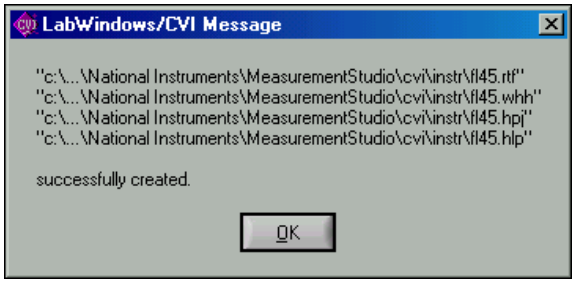

Figure 5-24. LabWindows/CVI Message Dialog Box

5. Click **OK**.

Example 6—Modifying an Existing IVI Driver to Work with a New Instrument

You do not always have to create an instrument driver from scratch. In many cases, it is easier to modify an existing driver for a similar instrument. This example shows how to use the Instrument Driver Development Wizard to generate a new instrument driver from an existing one.

To modify an existing driver, you first use the Instrument Driver Development Wizard.

1. Select **Tools»Create IVI Instrument Driver** to launch the wizard.
2. Click **Next** on the welcome panel to begin.
3. Select **Create Driver Based on Existing Driver**.
4. Click **Browse** to select an instrument driver to copy and modify.

   This example uses the Fluke 45 instrument driver that you create in Examples 1 through 4. However, you can choose any instrument driver. Make sure that the driver you select has most of the attributes and functions that are necessary for your instrument driver.

After you select the template, the panel appears as shown in Figure 5-25.

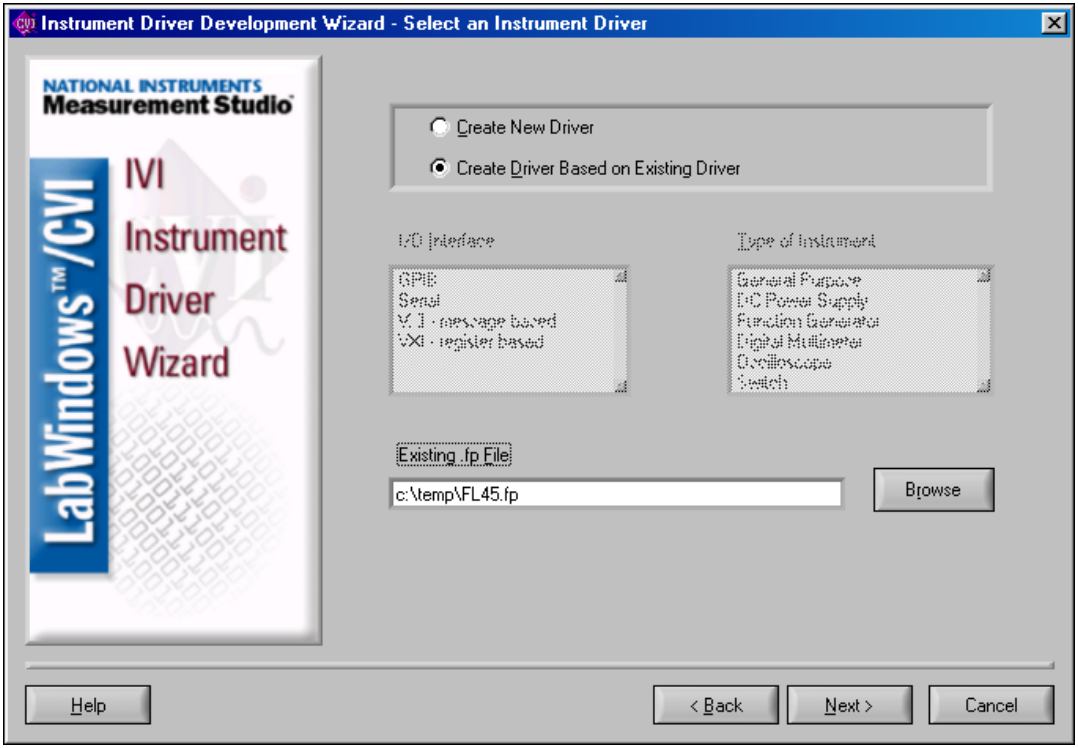

Figure 5-25. Select an Instrument Driver Panel

5. Click **Next** to continue.
6. Enter the name, prefix, and target directory for the new driver.
7. Click **Generate** to generate the new driver files.

   The wizard copies the existing driver to the new location. The wizard changes the filename prefixes, function prefixes, and macro prefixes to the instrument prefix you specify. The wizard also changes all occurrences of the old instrument prefix in the function panel help to the new instrument prefix. The resulting driver is completely operational and is ready for you to modify for use with the new instrument.

Typical modifications you might have to perform include the following:

- Modifying existing attributes and functions.
- Deleting attributes and functions that the instrument does not use. Refer to *Example 2—Editing the Instrument Driver Attributes* and *Example 3—Editing High-Level Instrument Driver Functions* in this chapter.

- Adding new attributes and functions. Refer to *Example 4—Adding New Attributes and Functions* in this chapter.
- Creating the instrument driver documentation. Refer to *Example 5—Creating Instrument Driver Documentation* in this chapter.

A

Technical Support Resources

Web Support

National Instruments Web support is your first stop for help in solving installation, configuration, and application problems and questions. Online problem-solving and diagnostic resources include frequently asked questions, knowledge bases, product-specific troubleshooting wizards, manuals, drivers, software updates, and more. Web support is available through the Technical Support section of `ni.com`.

NI Developer Zone

The NI Developer Zone at `ni.com/zone` is the essential resource for building measurement and automation systems. At the NI Developer Zone, you can easily access the latest example programs, system configurators, tutorials, technical news, as well as a community of developers ready to share their own techniques.

Customer Education

National Instruments provides a number of alternatives to satisfy your training needs, from self-paced tutorials, videos, and interactive CDs to instructor-led hands-on courses at locations around the world. Visit the Customer Education section of `ni.com` for online course schedules, syllabi, training centers, and class registration.

System Integration

If you have time constraints, limited in-house technical resources, or other dilemmas, you may prefer to employ consulting or system integration services. You can rely on the expertise available through our worldwide network of Alliance Program members. To find out more about our Alliance system integration solutions, visit the System Integration section of `ni.com`.

Worldwide Support

National Instruments has offices located around the world to help address your support needs. You can access our branch office Web sites from the Worldwide Offices section of `ni.com`. Branch office Web sites provide up-to-date contact information, support phone numbers, e-mail addresses, and current events.

If you have searched the technical support resources on our Web site and still cannot find the answers you need, contact your local office or National Instruments corporate. Phone numbers for our worldwide offices are listed at the front of this manual.

Glossary

A

ANSI	American National Standards Institute
Any Array	Special data type that represents any of the intrinsic C or user-defined array data types.
Any Type	Special data type that represents any of the intrinsic C or user-defined data types.
Attribute	Represents an instrument setting or a driver option.

B

binary control	A function panel control that operates like a mechanical on/off switch. A binary control specifies a parameter value to be one of two predefined values depending on whether the control is in the up or down position.

C

callback function	User-defined function that can be invoked by the IVI engine when a predefined event occurs.
channel-based attribute	An attribute that applies separately to each channel in a multi-channeled instrument driver.
channel string	A string used to represent the name of a channel in an instrument driver that supports multiple channels.
class attribute	An attribute defined by an instrument class specification that applies to all instruments of one type.
class instrument driver	High level instrument driver that accesses all specific instrument drivers of a particular type, thus allowing the application program to work on all instruments of the same type.

common control — A function panel control that specifies the first parameter in every function, primary and secondary, associated with a function panel. When a function panel has a common control, secondary functions have two parameters, the second of which is specified by a secondary control.

control — An input and output device that appears on a function panel for specifying function parameters and displaying function results.

D

deferring updates — The ability of the IVI engine to postpone the modification of attribute settings to the instrument until the driver calls `Ivi_Update`.

E

external module — A `.lib`, `.obj`, or `.dll` file that can be loaded and executed.

F

`.fp` file — A file that contains information that allows the LabWindows/CVI interactive program to display function panels that correspond to a specific instrument driver.

function panel — A user interface to the LabWindows/CVI libraries that allows interactive execution of library functions and is capable of generating code for inclusion in a program.

Function Panel Editor — The window used to create and modify instrument driver function panels.

function panel window — A window containing a collection of function panels representing all the functions that the user interactively can call from that window.

function tree — The hierarchical structure that defines the way functions in an instrument driver are grouped.

Function Tree Editor — The window used to create and modify the function tree for an instrument driver.

G

Generated Code window	A small window located at the bottom of the function panel that displays the code produced by the manipulation of function panel controls.
global variable control	A function panel control that displays the value of a global variable defined in LabWindows/CVI at the time the function panel is operated.

H

hex	hexadecimal

I

include file	A file that contains function declarations, constant definitions, and external declaration of global variables exported by the instrument driver.
inherent attribute	An attribute that is required by all IVI instrument drivers.
input control	A function panel control in which a value or variable name is entered from the keyboard.
instrument driver	A set of routines designed to control an instrument, and a set of data structures to represent the driver within LabWindows/CVI.
Instrument Library	A LabWindows/CVI library that contains instrument drivers.
instrument-specific attribute	An attribute, defined by a specific instrument driver, that applies only to a particular instrument model or family of models.
internal subroutine interface	The mechanism through which the driver can call other software modules it might use 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.
IVI engine	A support library for IVI instrument drivers that performs common tasks such as session creation, attribute management, and instrument status checking.

M

MB	megabytes of memory
message control	A function panel control that serves as a documentation tool that allows you to place text on a function panel.

N

numeric control	A function panel control that allows you to specify a numeric value using the mouse.

O

output control	A function panel control that displays the value of an output parameter after the function is called.

P

primary control	A function panel control that specifies parameters in the primary function.
primary function	The function that performs the main task associated with a function panel. The primary function always appears in the Generated Code window and is always executed when Go is selected from the command bar of a function panel.
primary parameter	A parameter that becomes a formal parameter to the function call.
private attribues	An attribute that an instrument driver uses internally and is not exported to the user.
public attributes	An attribute that an instrument driver exports to the user.

R

range tables	A table that specifies the valid range of values for an instrument attribute.
required functions	Instrument driver functions that are common to all instrument drivers.

return value control — A function panel control that displays a value returned from the primary function.

ring control — A control that displays a list of options one option at a time.

S

secondary control — A function panel control that specifies the parameter in a secondary function. Each secondary control is associated with a different secondary function, as opposed to primary controls, which are associated with the same function.

secondary function — A function that performs a task that is complementary to, but not required by, the primary task. Secondary functions do not appear in the Generated Code window unless you specifically activate them.

secondary parameter — A parameter that becomes a parameter to a separate function.

session — A collection of data structures maintained by the IVI engine necessary for the application program to communicate with the instrument.

session callbacks — Callback function that applies to the instrument as a whole.

slide control — A function panel control that resembles a mechanical slide switch. Inserts a parameter value depending on the position of the cross-bar on the slide control.

T

type library — A file or component within another file (such as a `.dll`) that contains type information about the exported functions.

typesafe functions — A set of functions whose prototypes strictly define the data type of all parameters such as the `Ivi_SetAttribute<type>` functions. You use the set attribute function that corresponds to the data type of the attribute you are attempting to set.

V

value parameter — An integer, long, or double-precision scalar parameter whose value is not modified by the subroutine or function. In other words, an integer, long, single-precision, or double-precision scalar parameter is a value parameter if and only if its function panel control is *not* an output control.

virtual channel name — An alias for a specific driver channel string. You specify virtual channel names and the specific driver channel strings to which they refer in the `ivi.ini` configuration file.

Index

A

action/status functions, 1-12
application functions
initialization and close functions not called by (note), 1-10
purpose and use, 1-10
architecture. *See* instrument driver architecture.
ASCII text file, as component of instrument drivers, 1-4
attribute callback functions, 2-21 to 2-25
check callback, 2-23
coerce callback, 2-23 to 2-24
compare callback, 2-24 to 2-25
overview, 2-21 to 2-22
programming, 3-15 to 3-29
range table callbacks, 3-18
range tables, 3-18
read and coerce callbacks for ViString attributes, 3-15 to 3-16
reading strings from instrument, 3-16 to 3-17
using viRead, 3-17
using viScanf, 3-16 to 3-17
write callbacks, 3-16
range table callback, 2-25
read callback, 2-22
write callback, 2-22 to 2-23
Attribute Editor, 4-1 to 4-12
adding and editing
instrument attributes, 4-6 to 4-8
range tables, 4-8 to 4-12
customizing files generated by Instrument Driver Development Wizard, 3-10
Edit Attribute dialog box, 4-6 to 4-8
entering information, 4-6 to 4-8
illustration, 4-6
Edit Driver Attributes dialog box, 4-3 to 4-5
command buttons, 4-4 to 4-5
illustration, 4-3
Instrument Attributes list box, 4-3 to 4-4
restrictions on modification to inherent and class attributes, 4-4
invoking, 4-1
limitations in updates to driver files, 4-2
Range Tables dialog box, 4-9 to 4-12
Edit Range Table dialog box, 4-10 to 4-12
illustration, 4-9
requirements for using, 4-1 to 4-2
attribute functions
callback functions. *See* attribute callback functions.
purpose and use, 1-14
attributes. *See also* state-caching, IVI.
accessing with user-callable functions, 3-34
adding new attributes (example), 5-26 to 5-35
Configure Hold function panel, 5-30 to 5-34
creating Configure Hold Function Body, 5-35
Hold Enable attribute, 5-26 to 5-28
Hold Threshold attribute, 5-28 to 5-30
class attributes, 2-9
class-defined attributes, 2-7
comparison precision, 2-17 to 2-18
creating and declaring, 2-9 to 2-18
customizing Wizard-generated attributes, 3-10 to 3-12
adding new attributes, 3-11 to 3-12
deleting attributes, 3-11

general modifications, 3-12
modifying existing attributes, 3-10 to 3-11
editing attributes (example), 5-12 to 5-20
customizing measurement function attribute, 5-13 to 5-16
deleting unused attributes, 5-17 to 5-20
modifying Write and Read callbacks, 5-16 to 5-17
flags, 2-10 to 2-12
description of individual flags, 2-11 to 2-12
list of flags (table), 2-10 to 2-11
getting or setting values in one command, 2-20
IDs
programming considerations, 3-12 to 3-13
requirements, 2-9 to 2-10
inherent. *See* inherent attributes, IVI.
instrument-specific
constant name for ID, 2-10
definition, 2-7
invalidation
by changing one attribute, 2-19
by changing two attributes, 2-20
private or hidden, 2-7
programming considerations, 3-12 to 3-29
callbacks, 3-15 to 3-29
data types, 3-15
ID values, 3-12 to 3-13
range tables, 3-18
simulation, 3-14
value definitions, 3-13 to 3-14
programming examples, 3-19 to 3-29
changing valid range, 3-24 to 3-28
check, coerce, and compare callbacks, 3-29
continuous range, 3-21 to 3-22
continuous range with discrete settings, 3-22 to 3-24
discrete settings, 3-19 to 3-21
dynamic range tables, 3-27 to 3-28
multiple static range tables, 3-24 to 3-27
public, 2-7
purpose and use, 2-7
range tables. *See* range tables.
types of attributes, 2-7
unsupported, in generated driver files, 3-9
user-callable functions for setting, 3-36 to 3-37

C

callback functions
attribute callbacks, 2-21 to 2-25
check callback, 2-23
coerce callback, 2-23 to 2-24
compare callback, 2-24 to 2-25
overview, 2-21 to 2-22
programming, 3-15 to 3-29
range table callbacks, 3-18
range tables, 3-18
read and coerce callbacks for ViString attributes, 3-15 to 3-16
reading strings from instrument, 3-16 to 3-17
write callbacks, 3-16
programming considerations, 3-15 to 3-29
range table callback, 2-25
read callback, 2-22
write callback, 2-22 to 2-23
default, 2-17
definition, 2-3
overview, 2-8
purpose and use, 1-14

session callbacks, 2-25 to 2-27
check status, 2-26 to 2-27
definition, 1-14
instruments without error queues, 2-27
operation complete callback, 2-25 to 2-26
channel strings
definition, 2-27
user-callable functions, 3-37 to 3-38
channel-based attribute, defined, 2-27
channels, 2-27 to 2-28
coercing and validating channel names, 2-28
overview, 2-27 to 2-28
passing channel names to IVI functions, 2-28
virtual channel names, 2-28
check callback functions
attribute programming example, 3-29
default, 2-17
purpose and use, 2-23
check status callback, 2-26 to 2-27
class attributes, 2-9
class-defined attributes, 2-7
classes
function tree classes, 3-43 to 3-44
IVI Foundation instrument driver classes, 1-11 to 1-12
close function
definition, 1-14
user-callable functions, 3-38
coerce callback functions
attribute programming example, 3-29
default, 2-17
programming read and coerce callbacks for ViString attributes, 3-15 to 3-16
purpose and use, 2-23 to 2-24
coerced range tables
example, 2-15 to 2-16
IVI_VAL_COERCED, 2-14
coercion
channel names, 2-28
state-caching mechanism, 2-20 to 2-21
compare callback functions
attribute programming example, 3-29
purpose and use, 2-24 to 2-25
comparison precision for attributes, 2-17 to 2-18
component functions, 1-10 to 1-12
categories, 1-10
developer-specified functions, 1-11
instrument class specifications, 1-11 to 1-12
required functions, 1-11
configuration entries, IVI instrument drivers, 2-33
configuration functions
overview, 1-12
user-callable functions that set attributes, 3-36 to 3-37
conventions used in manual, *iv*
customer education, A-1

D

data types
programming considerations, 3-15
VISA data types (table), 3-38 to 3-39
default callback functions, 2-17
developing instrument drivers. *See* instrument driver development.
direct instrument I/O, 3-34
discrete range tables
example, 2-15
IVI_VAL_DISCRETE, 2-13
documentation
creating (example), 5-36 to 5-37
.doc file, 5-36 to 5-37
Windows help, 5-37
.doc and .hlp files, 3-49 to 3-50

online help examples, 3-45 to 3-49
function class help (figure), 3-46
function panel control help (figure), 3-48
function panel help (figure), 3-47
function panel status control help (figure), 3-49
instrument help (figure), 3-45
dynamic range tables
programming examples, 3-27 to 3-28
purpose and use, 2-16 to 2-17

E

Edit Attribute dialog box, 4-6 to 4-8
entering information, 4-6 to 4-8
illustration, 4-6
Edit Driver Attributes dialog box, 4-3 to 4-5
command buttons, 4-4 to 4-5
illustration, 4-3
Instrument Attributes list box, 4-3 to 4-4
restrictions on modification to inherent and class attributes, 4-4
error codes, 3-41 to 3-42
completion and warning codes (table), 3-41
instrument driver error codes (table), 3-42
suggested error values (table), 3-41
error info attributes, 2-35
Error Message function (table), 1-13
Error Query function (table), 1-13
error queue
checking with check status callback, 2-27
instruments without error queues, 2-27
error-reporting guidelines, 3-41 to 3-42
external interface model. *See* instrument driver architecture.

F

files for instrument drivers, 1-4
flags for instrument driver attributes, 2-10 to 2-12
description of individual flags, 2-11 to 2-12
list of flags (table), 2-10 to 2-11
floating point numbers, precision comparison, 2-17 to 2-18
Fmt function, using with portable instrument drivers, 3-40
function panels. *See also* interactive developer interface.
generated by Instrument Driver Development Wizard, 3-7
instrument function panel (.fp) file, 1-4
programming considerations, 3-43
function tree classes, 3-43 to 3-44
functional body
definition, 1-6
purpose and use, 1-7
functions. *See* instrument driver functions.

G

generated driver files. *See* Instrument Driver Development Wizard.
Get Next Coercion Record function (table), 1-13
GetAttribute functions, not used in application programs, 2-5
Get/Clear Error Info functions (table), 1-13
get/set/check functions, 2-7 to 2-8

H

help file, as component of instrument drivers, 1-4
help information examples, 3-45 to 3-49
function class help (figure), 3-46
function panel control help (figure), 3-48

function panel help (figure), 3-47
function panel status control help (figure), 3-49
instrument help (figure), 3-45
hidden attributes, 2-7
high-level functions
editing (example), 5-20 to 5-25
overview, 2-28 to 2-29

I

include files
contents of, 1-4
generated by Instrument Driver Development Wizard, reviewing, 3-9
inherent attribute reference
IVI_ATTR_CACHE, 2-35 to 2-36
IVI_ATTR_CHECK_STATUS_CALLBACK, 2-36
IVI_ATTR_CLASS_MAJOR_VERSION, 2-36
IVI_ATTR_CLASS_MINOR_VERSION, 2-36
IVI_ATTR_CLASS_PREFIX, 2-37
IVI_ATTR_CLASS_REVISION, 2-37
IVI_ATTR_DRIVER_MAJOR_VERSION, 2-37
IVI_ATTR_DRIVER_MINOR_VERSION, 2-38
IVI_ATTR_DRIVER_REVISION, 2-38
IVI_ATTR_DRIVER_SETUP, 2-38
IVI_ATTR_ENGINE_MAJOR_VERSION, 2-38
IVI_ATTR_ENGINE_MINOR_VERSION, 2-39
IVI_ATTR_ENGINE_REVISION, 2-39
IVI_ATTR_ERROR_ELABORATION, 2-39
IVI_ATTR_FUNCTION_CAPABILITIES, 2-39
IVI_ATTR_GROUP_CAPABILITIES, 2-39
IVI_ATTR_INTERCHANGE_CHECK, 2-40
IVI_ATTR_I/O_SESSION, 2-40
IVI_ATTR_LOGICAL_NAME, 2-40 to 2-41
IVI_ATTR_MODULE_PATHNAME, 2-41
IVI_ATTR_NUM_CHANNELS, 2-41
IVI_ATTR_OPC_CALLBACK, 2-41
IVI_ATTR_PRIMARY_ERROR, 2-42
IVI_ATTR_QUERY_INSTR_STATUS, 2-42
IVI_ATTR_RANGE_CHECK, 2-42 to 2-43
IVI_ATTR_RECORD_COERCIONS, 2-43
IVI_ATTR_RESOURCE_DESCRIPTOR, 2-44
IVI_ATTR_SECONDARY_ERROR, 2-44
IVI_ATTR_SIMULATE, 2-44
IVI_ATTR_SPECIFIC_PREFIX, 2-45
IVI_ATTR_SPY, 2-45
IVI_ATTR_VISA_RM_SESSION, 2-45
inherent attributes, IVI
categories, 2-34 to 2-35
constant name for ID, 2-9
definition, 2-3, 2-7
error info, 2-35
instrument capabilities, 2-34
session info, 2-34
session I/O, 2-34
user options, 2-34
version info, 2-34 to 2-35
initialization functions
overview, 1-12
*Prefix*_init, 2-5
*Prefix*_InitWithOptions, 2-5
programming considerations, 2-5
user-callable functions, 3-37 to 3-38
instrument capabilities attributes, 2-34

instrument driver architecture, 1-5 to 1-15
external interface model, 1-5 to 1-8
functional body, 1-7
general model (figure), 1-6
interactive developer interface, 1-7
IVI engine, 1-7
programmatic developer interface, 1-7
subroutine interface, 1-8
VISA I/O interface, 1-7 to 1-8
internal design model, 1-8 to 1-15
achieving interchangeability, 1-14 to 1-15
action/status functions, 1-12
application functions, 1-10
attribute functions, 1-14
close function, 1-14
component functions, 1-10 to 1-12
configuration functions, 1-12
illustration, 1-9
initialization functions, 1-12
measurement functions, 1-13
utility functions, 1-13 to 1-14
instrument driver classes, IVI Foundation, 1-11 to 1-12
instrument driver development, 3-1 to 3-53. *See also* function panels; instrument driver development examples; Instrument Driver Development Wizard.
attribute examples, 3-19 to 3-29
check, coerce, and compare callbacks, 3-29
continuous range, 3-21 to 3-22
continuous range with discrete settings, 3-22 to 3-24
discrete settings, 3-19 to 3-21
dynamic range tables, 3-27 to 3-28
multiple static range tables, 3-24 to 3-27
attributes, 3-12 to 3-29
callbacks, 3-15 to 3-29
data types, 3-15
ID values, 3-12 to 3-13
range tables, 3-18
simulation, 3-14
value definitions, 3-13 to 3-14
checklist, 3-50 to 3-53
data types
programming considerations, 3-15
VISA data types (table), 3-38 to 3-39
documentation guidelines, 3-44 to 3-50
.doc and .hlp files, 3-49 to 3-50
online help, 3-45 to 3-49
error-reporting guidelines, 3-41 to 3-42
function panels, 3-43
function tree hierarchy, 3-43 to 3-44
general guidelines, 3-1 to 3-2, 3-42 to 3-43
IVI instrument drivers, 2-5
naming drivers, 3-3
portable instrument drivers, 3-38 to 3-40
steps for programming, 3-2 to 3-3
user-callable functions, 3-30 to 3-38
accessing attributes, 3-34
channel strings, 3-37 to 3-38
checking instrument status, 3-35 to 3-36
close functions, 3-38
direct instrument I/O, 3-34
functions that only set attributes, 3-36 to 3-37
initialization functions, 3-37 to 3-38
instrument driver function structure, 3-30 to 3-32
locking/unlocking sessions, 3-33
parameter checking, 3-33 to 3-34
simulating output parameters, 3-34 to 3-35
VXI instrument guidelines, 3-50

instrument driver development examples, 5-1 to 5-39
adding new attributes and functions, 5-26 to 5-35
Configure Hold function panel, 5-30 to 5-34
creating Configure Hold Function Body, 5-35
Hold Enable attribute, 5-26 to 5-28
Hold Threshold attribute, 5-28 to 5-30
creating documentation, 5-36 to 5-37
.doc file, 5-36
Windows help, 5-37
creating IVI instrument driver with Wizard, 5-2 to 5-11
General Command Strings panel, 5-4
General Information panel, 5-3
ID Query panel, 5-6
Reset panel, 5-7
Revision panel, 5-9
Select an Instrument Driver panel, 5-2
Self Test panel, 5-8
Standard Operations panel, 5-5
Test panel, 5-10
Test Results panel, 5-11
editing attributes, 5-12 to 5-20
customizing measurement function attribute, 5-13 to 5-16
deleting unused attributes, 5-17 to 5-20
modifying Write and Read callbacks, 5-16 to 5-17
editing high-level functions, 5-20 to 5-25
deleting unnecessary functions, 9-25
Fetch function, 5-20 to 5-25
modifying existing IVI driver, 5-37 to 5-39
Instrument Driver Development Wizard, 3-3 to 3-9
creating IVI instrument driver (example), 5-2 to 5-11
General Command Strings panel, 5-4
General Information panel, 5-3
ID Query panel, 5-6
Reset panel, 5-7
Revision panel, 5-9
Self Test panel, 5-8
Standard Operations panel, 5-5
Test panel, 5-10
Test Results panel, 5-11
customizing generated driver files, 3-10 to 3-12
adding new attributes and functions, 3-11 to 3-12
deleting attributes and functions, 3-11
general modifications, 3-12
modifying existing attributes and functions, 3-10 to 3-11
reviewing generated driver files, 3-7 to 3-9
extended functions and attributes, 3-9
function panels, 3-7
include file, 3-9
source file, 3-8 to 3-9
.sub file, 3-7 to 3-8
using Attribute Editor, 3-10
running preliminary I/O tests, 3-7
selecting template, 3-5 to 3-6
Selection Panel (figure), 3-5
starting, 3-5
worksheet for necessary information, 3-4
instrument driver functions
action/status functions, 1-12
adding new attributes and functions (example), 5-26 to 5-35
Configure Hold function panel, 5-30 to 5-34

creating Configure Hold Function Body, 5-35
Hold Enable attribute, 5-26 to 5-28
Hold Threshold attribute, 5-28 to 5-30
callbacks. *See* callback functions.
close function, 1-14
configuration functions, 1-12
customizing Wizard-generated functions, 3-10 to 3-12
adding new attributes and functions, 3-11 to 3-12
deleting attributes and functions, 3-11
general modifications, 3-12
modifying existing attributes and functions, 3-10 to 3-11
editing high-level functions (example), 5-20 to 5-25
deleting unnecessary functions, 9-25
Fetch function, 5-20 to 5-25
extended functions, in generated driver files, 3-9
get/set/check functions, 2-7 to 2-8
high-level functions, 2-28 to 2-29
initialization functions, 1-12, 2-5
measurement functions, 1-13
required functions, 2-6
typesafe functions, 2-7
user-callable functions, 3-30 to 3-38
accessing attributes, 3-34
channel strings, 3-37 to 3-38
checking instrument status, 3-35 to 3-36
close functions, 3-38
direct instrument I/O, 3-34
functions that only set attributes, 3-36 to 3-37
initialization functions, 3-37 to 3-38
instrument driver function structure, 3-30 to 3-32
locking/unlocking sessions, 3-33
parameter checking, 3-33 to 3-34
simulating output parameters, 3-34 to 3-35
utility functions, 1-13 to 1-14
instrument drivers. *See also* IVI instrument drivers.
definition, 1-1, 2-2
files for instrument drivers, 1-4
historical evolution, 1-2 to 1-3
operating, 1-5
overview, 1-1
purpose and use, 1-4
instrument simulation. *See* simulation of instruments.
instrument-specific attributes
constant name for ID, 2-9
definition, 2-7
Intelligent Virtual Instrument drivers. *See* IVI instrument drivers.
interactive developer interface
definition, 1-6
purpose and use, 1-7
interchangeability
IVI Foundation standards, 1-2
requirements for achieving, 1-14 to 1-15
internal design model. *See* instrument driver architecture.
internal subroutine interface, 1-6
I/O tests, running from Wizard, 3-7
IVI engine
definition, 1-6
interaction with IVI instrument drivers, 2-3 to 2-4
purpose and use, 1-7
IVI instrument drivers, 2-1 to 2-45
attribute callback functions, 2-21 to 2-25
channels, 2-27 to 2-28
comparison precision, 2-17 to 2-18
configuration entries, 2-33

creating and declaring attributes, 2-9 to 2-18
definition, 2-2
features, 1-2 to 1-3, 2-1 to 2-2
functions and attribute model, 2-6 to 2-8
high-level driver functions, 2-28 to 2-29
inherent IVI attributes, 2-34 to 2-45
interaction with IVI engine, 2-3 to 2-4
modifying existing IVI driver (example), 5-37 to 5-39
multithread safety, 2-32 to 2-33
operation, 2-3 to 2-4
overview, 2-2 to 2-3
programming, 2-5. *See also* instrument driver development.
range checking, 2-29 to 2-30
range tables, 2-12 to 2-17
session callback functions, 2-25 to 2-27
simulation, 2-21 to 2-32
state-caching mechanism, 2-18 to 2-21
status checking, 2-30 to 2-31
types of IVI drivers, 1-3
IVI_ATTR_CACHE, 2-35 to 2-36
IVI_ATTR_CHECK_STATUS_CALLBACK, 2-36
IVI_ATTR_CLASS_MAJOR_VERSION, 2-36
IVI_ATTR_CLASS_MINOR_VERSION, 2-36
IVI_ATTR_CLASS_PREFIX, 2-37
IVI_ATTR_CLASS_REVISION, 2-37
IVI_ATTR_DRIVER_MAJOR_VERSION, 2-37
IVI_ATTR_DRIVER_MINOR_VERSION, 2-38
IVI_ATTR_DRIVER_REVISION, 2-38
IVI_ATTR_DRIVER_SETUP, 2-38
IVI_ATTR_ENGINE_MAJOR_VERSION, 2-38
IVI_ATTR_ENGINE_MINOR_VERSION, 2-39
IVI_ATTR_ENGINE_REVISION, 2-39
IVI_ATTR_ERROR_ELABORATION, 2-39
IVI_ATTR_FUNCTION_CAPABILITIES, 2-39
IVI_ATTR_GROUP_CAPABILITIES, 2-39
IVI_ATTR_INTERCHANGE_CHECK, 2-40
IVI_ATTR_I/O_SESSION, 2-40
IVI_ATTR_LOGICAL_NAME, 2-40 to 2-41
IVI_ATTR_MODULE_PATHNAME, 2-41
IVI_ATTR_NUM_CHANNELS, 2-41
IVI_ATTR_OPC_CALLBACK, 2-41
IVI_ATTR_PRIMARY_ERROR, 2-42
IVI_ATTR_QUERY_INSTR_STATUS, 2-42
IVI_ATTR_RANGE_CHECK, 2-42 to 2-43
IVI_ATTR_RECORD_COERCIONS, 2-43
IVI_ATTR_RESOURCE_DESCRIPTOR, 2-44
IVI_ATTR_SECONDARY_ERROR, 2-44
IVI_ATTR_SIMULATE, 2-44
IVI_ATTR_SPECIFIC_PREFIX, 2-45
IVI_ATTR_SPY, 2-45
IVI_ATTR_VISA_RM_SESSION, 2-45
IVI-C class drivers, 1-3
IVI-C driver, 1-3
IVI-C specific drivers, 1-3
IVI-COM driver, 1-3
ivi.ini file, 2-33
IVI_VAL_ALWAYS_CACHE flag, 2-11
IVI_VAL_COERCEABLE_ONLY_BY_INSTR flag, 2-11
IVI_VAL_COERCED range tables, 2-14
IVI_VAL_DISCRETE range tables, 2-13
IVI_VAL_DONT_CHECK_STATUS flag, 2-12
IVI_VAL_HIDDEN flag, 2-11
IVI_VAL_MULTI_CHANNEL flag, 2-11
IVI_VAL_NEVER_CACHE flag, 2-11
IVI_VAL_NOT_READABLE flag, 2-11

IVI_VAL_NOT_SUPPORTED flag, 2-11
IVI_VAL_NOT_USER_READABLE flag, 2-11
IVI_VAL_NOT_USER_WRITABLE flag, 2-11
IVI_VAL_NOT_WRITABLE flag, 2-11
IVI_VAL_RANGED range tables, 2-13
IVI_VAL_USE_CALLBACKS_FOR_SIMULATION flag, 2-12
IVI_VAL_WAIT_FOR_OPC_AFTER_WRITES flag, 2-12
IVI_VAL_WAIT_FOR_OPC_BEFORE_READS flag, 2-12

L

Lock/Unlock Session functions
    overview (table), 1-14
    programming considerations, 2-5
    user-callable functions, 3-33

M

measurement functions, 1-13
models for instrument drivers. *See* instrument driver architecture.
multithread safety, IVI instrument drivers, 2-32 to 2-33

N

names
    files for instrument drivers, 1-4
    instrument drivers, 3-3
NI Developer Zone, A-1

O

operation complete callback, 2-25 to 2-26

P

parameter checking, user-callable functions, 3-33 to 3-34
portable instrument drivers, developing, 3-38 to 3-40
    declaring instrument driver functions, 3-39
    instrument driver data types (table), 3-38 to 3-39
    using Scan and Fmt functions, 3-40
precision of floating point numbers, 2-17 to 2-18
prefix for instrument driver names, 3-3
*Prefix*_CheckAttribute functions, 2-7
*Prefix*_GetAttribute functions, 2-7
*Prefix*_init function, 2-5
*Prefix*_InitWithOptions function, 2-5
*Prefix*_LockSession function, 2-5
*Prefix*_SetAttribute functions, 2-7
*Prefix*_UnLockSession function, 2-5
private attributes, 2-7
programmatic developer interface
    definition, 1-6
    purpose and use, 1-7
programming examples. *See* instrument driver development examples.
programming instrument drivers. *See* instrument driver development.
public attributes, 2-7

R

range checking
    definition, 2-3
    purpose and use, 2-29 to 2-30
range table callbacks
    programming considerations, 3-18
    purpose and use, 2-25

range tables, 2-12 to 2-17
attribute programming examples
continuous range, 3-21 to 3-22
continuous range with discrete settings, 3-22 to 3-24
discrete settings, 3-19 to 3-21
dynamic range tables, 3-27 to 3-28
multiple static range tables, 3-24 to 3-27
coerced range table example, 2-15 to 2-16
default check and coerce callbacks, 2-17
definition, 2-3
discrete range table example, 2-15
IVI_VAL_COERCED, 2-14
IVI_VAL_DISCRETE, 2-13
IVI_VAL_RANGED, 2-13
programming considerations, 3-18
ranged range table example, 2-16
static and dynamic, 2-16 to 2-17
structures, 2-13 to 2-14
Range Tables dialog box, 4-9 to 4-12
Edit Range Table dialog box, 4-10 to 4-12
illustration, 4-9
ranged range tables
example, 2-16
IVI_VAL_RANGED, 2-13
read callback functions
programming read and coerce callbacks for ViString attributes, 3-15 to 3-16
purpose and use, 2-22
reading strings
using viRead, 3-17
using viScanf, 3-16 to 3-17
required functions for instrument drivers, 2-6
Reset function (table), 1-13
Revision Query function (table), 1-13

S

Scan function, using with portable instrument drivers, 3-40
Select Attribute Constant dialog box, 3-8
session callback functions, 2-25 to 2-27
check status, 2-26 to 2-27
definition, 1-14
instruments without error queues, 2-27
operation complete callback, 2-25 to 2-26
session info attributes, 2-34
session I/O attributes, 2-34
sessions, initializing, 2-5
SetAttribute functions, not used in application programs, 2-5
simulation of instruments
definition, 2-3
overview, 1-2
preventing instrument I/O during (note), 2-32
programming considerations, 3-14
purpose and use, 2-31 to 2-32
user-callable functions, 3-34 to 3-35
source files
as component of instrument drivers, 1-4
generated by Instrument Driver Development Wizard
categories of functions in, 3-8 to 3-9
reviewing, 3-8 to 3-9
state-caching, IVI, 2-18 to 2-21
definition, 2-3
enabling and disabling, 2-21
initial instrument state, 2-19
instrument coerce values, 2-20 to 2-21
invalidation of attributes
by changing one attribute, 2-19
by changing two attributes, 2-20
overview, 1-2, 2-18 to 2-19
setting/getting values of two attributes with one command, 2-20
special cases, 2-19 to 2-21

static range tables
programming examples, 3-24 to 3-27
purpose and use, 2-16 to 2-17
status checking
check status callback, 2-26 to 2-27
purpose and use, 2-30 to 2-31
user-callable functions, 3-35 to 3-36
status query, configurable, 2-3
strings, reading
using viRead, 3-17
using viScanf, 3-16 to 3-17
structures
programming user-callable functions, 3-30 to 3-32
range tables, 2-13 to 2-14
.sub file
definition, 1-4
generated by Instrument Driver Development Wizard, 3-7 to 3-8
subroutine interface, 1-8
system integration, by National Instruments, A-1

T

technical support resources, A-1 to A-2
templates, selecting for instrument driver, 3-5 to 3-6
testing instrument drivers, running preliminary I/O tests from Wizard, 3-7
typesafe functions, 2-7

U

unlock session functions. *See* Lock/Unlock Session functions.
user options attributes, 2-34
user-callable functions, programming, 3-30 to 3-38
accessing attributes, 3-34
channel strings, 3-37 to 3-38
checking instrument status, 3-35 to 3-36
close functions, 3-38
direct instrument I/O, 3-34
functions that only set attributes, 3-36 to 3-37
initialization functions, 3-37 to 3-38
instrument driver function structure, 3-30 to 3-32
locking/unlocking sessions, 3-33
parameter checking, 3-33 to 3-34
simulating output parameters, 3-34 to 3-35
utility functions, 1-13 to 1-14

V

version info attributes, 2-34 to 2-35
_VI_FUNC macro, 3-39
viRead function, 3-17
virtual channel names, 2-28
Virtual Instrumentation Software Architecture. *See* VISA I/O interface.
VISA data types (table), 3-38 to 3-39
VISA I/O interface
definition, 1-6
purpose and use, 1-7 to 1-8
VISA I/O library macros (table), 3-39
viScanf function, 3-16 to 3-17
ViString attributes, read and coerce callbacks for, 3-15 to 3-16
VXI instrument programming guidelines, 3-50
VXI*plug&play* instrument driver, 1-2 to 1-3

W

Web support from National Instruments, A-1
Worldwide technical support, A-2
write callback functions
programming considerations, 3-16
purpose and use, 2-22 to 2-23
Write/Read Instrument Data functions (table), 1-14