LabWindows/CVI

Instrument Driver
Developers Guide

Worldwide Technical Support and Product Information

`www.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 (Ontario) 905 785 0085, Canada (Québec) 514 694 8521,
China 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, Mexico (D.F.) 5 280 7625,
Mexico (Monterrey) 8 357 7695, Netherlands 0348 433466, Norway 32 27 73 00, Poland 48 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 2377 1200, 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`

Important Information

Warranty

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™, National Instruments™, the National Instruments logo, 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.

Contents

About This Manual

Chapter 1
Instrument Driver Overview

Chapter 2
IVI Architecture Overview

Chapter 3
Developing an Instrument Driver

Chapter 4
Attribute Editor

Chapter 5
Function Tree Editor

Chapter 6
Function Panel Editor

Chapter 7
Adding Help Information

Chapter 8
Programming Guidelines for Instrument Drivers

Chapter 9
Instrument Driver Examples

Appendix A
Technical Support Resources

Glossary

Index

Figures

Tables

About This Manual

The *LabWindows/CVI Instrument Driver Developer 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. Follow the procedures in this guide when developing instrument drivers for personal use or for general distribution to other users. The software tools you use to create instrument drivers are included in the standard LabWindows/CVI package.

The *LabWindows/CVI Instrument Driver Developer 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 LabWindows/CVI* manual, the *LabWindows/CVI User Manual*, and the *LabWindows/CVI Online Help*, and that you are comfortable with the LabWindows/CVI software. Please refer to the *LabWindows/CVI User Manual* for specific instructions on operating LabWindows/CVI.

Conventions Used in This Manual

The following conventions are used in this manual:

»	The » symbol leads you through nested menu items and dialog box options to a final action. 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 bold`	Bold text in this font denotes the messages and responses that the computer automatically prints to the screen. This font also emphasizes lines of code that are different from the other examples.
`monospace italic`	Italic text in this font denotes that you must enter the appropriate words or values in the place of these items.

1

Instrument Driver Overview

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

What Is an Instrument Driver?

In the early days of computer-controlled instrumentation systems, programmers used BASIC I/O statements in their application programs to send and receive command and data strings to and from the various instruments connected to their computer via GPIB. Each instrument responded to particular ASCII strings as documented in each vendor's instrument user manuals. Programmers were responsible for learning each command set and writing the control program.

Programming is often the most time-consuming part of developing an automated test system, especially if programmers have to learn the various command sets of the different instruments they use. This effect is compounded when programmers find themselves repeating their work when they create new applications with the same instruments. It became clear that programmers 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 concept of the instrument driver 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 and modifiable, 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 has 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.

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 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 IVI (Interchangeable Virtual Instrument) Foundation was established to create open standards for instrument drivers that leverage the VXI*plug&play* model. Even though IVI stands for *Interchangeable Virtual Instrument*, the foundation addresses other system level issues such as instrument simulation and higher performance. IVI instrument drivers from National Instruments comply with the VXI*plug&play* and IVI Foundation standards and have many additional features. Three of the most important features are:

- **Interchangeability**—The IVI Foundation defines standard APIs for common types of instruments such as DMM, oscilloscope, function/arbitrary waveform generator, DC power supply, switch, and so on. National Instruments 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, they 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. IVI drivers from National Instruments return simulated data to output parameters. With simulated data, developers can develop code for instruments even when the instruments are not available. All parameters are range checked so developers can determine if their program is configuring the instrument with valid settings even while simulating.

- **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. IVI drivers from National Instruments 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*.

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 multimeter, a class 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 an ASCII string of X-Y coordinates into two integer arrays suitable for plotting.

An IVI instrument driver consists of the following six files.

- The instrument driver source or object code, which can be a `.obj`, `.dll`, or `.c` file

- The instrument include (`.h`) file, which contains function declarations, constant definitions, and external declarations of global variables

- The instrument 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 Instruments Attributes** command.

- An ASCII text file (`.txt`). which contains driver specifications, such as models supported, driver version, and required software

- A Help file (`.hlp`) that contains documentation for the instrument driver

The six filenames 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`.

For more information, refer to *Using Instrument Drivers*, in Chapter 3, *Project Window*, in the *LabWindows/CVI User Manual*.

How Users Operate the Instrument Driver

To the user, an instrument driver represents 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 the instrument driver function.

A function panel displays symbolic controls that represent parameters to the function. By manipulating the controls, the user constructs a specific function call that can then be executed or saved 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 knowing 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 instrument driver interfaces with the rest of the system.
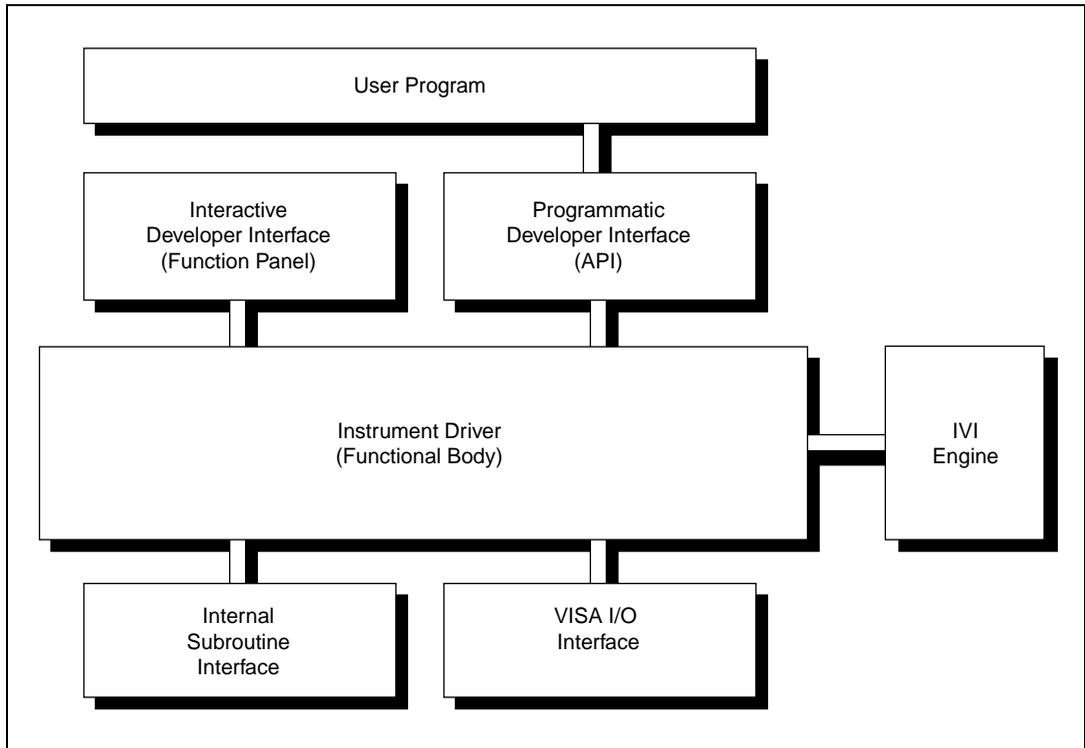
Figure 1-1. Instrument Driver External Interface Model

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 easily can 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 8, *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 graphical function panels. Function panels, referred to as the *interactive developer interface*, help you learn how to use the instrument driver. The function panel interface allows you to operate a particular instrument driver function interactively 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 it work, and a library that is an API to that 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 the National Instruments 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 might 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. This type of approach can be used to build virtual instruments that combine hardware and software capabilities. Complete high-level tests can be developed and packaged as instrument drivers that can be used by other test developers.

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 only control 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. These functions can be defined within the LabWindows/CVI instrument driver source file or supplied 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.
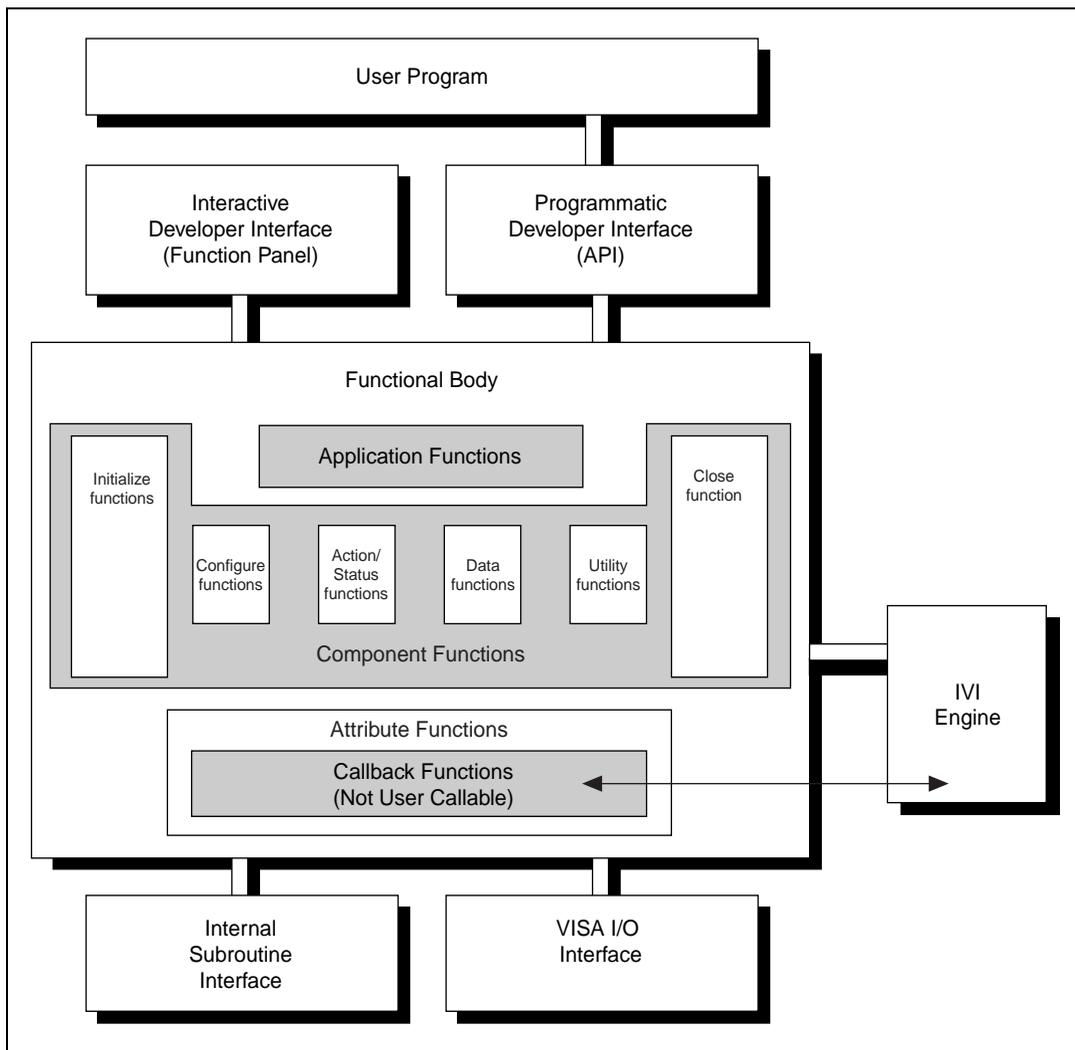
Figure 1-2. Instrument Driver Internal Design Mode

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. As another example, a user can initialize and configure an instrument once, and then trigger and read from the instrument several times.

Component Functions

LabWindows/CVI instrument drivers have component functions, which are divided into seven categories: initialize, configuration, action/status, data, utility, attribute, and close. 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 up-front 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 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

The instrument driver developer specifies the remainder of the functions in the instrument driver. All instruments have configuration functions, for example, but different instruments can have different numbers of configuration functions depending on the differences in the

ways you can configure the instruments. General guidelines in Chapter 8, *Programming Guidelines for Instrument Drivers*, define, organize, and structure the functions within each category. By following these guidelines, similar instruments will 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. They define some functionality as required and some as optional. They also allow 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, *Developing an Instrument Driver*, 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.

Initialize Functions

The initialize functions initialize the software connection to the instrument. The initialize functions can optionally perform an instrument identification query and reset operations. It performs 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. There can be numerous configuration functions, 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 functions cause the instrument to initiate or terminate test and measurement operations. 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.

Data Functions

The data functions include functions to transfer data to or from the instrument. Examples include functions for reading a measured value or waveform from an instrument, functions for downloading waveforms or digital patterns to a source instrument, and so on. The specific routines in this category and the actual operations performed by those routines are left up to the instrument driver developer.

Utility Functions

The utility functions can perform a variety of operations. Some utility functions are required, such as reset, self-test, error query, error message, and revision query, and some are defined by the developer.

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	This 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.

Invalidate All Attributes	This function invalidates the state of all attributes.
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	This instrument data functions allow the end user to perform instrument I/O directly.

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.

Close Function

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

Application Functions

The application functions are high-level test and measurement oriented routines. 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.

Note   Instrument driver application-level functions do not call the initialization or close 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.

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

Non-IVI drivers do not have callback functions.

2

IVI Architecture Overview

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

What is IVI?

IVI (Interchangeable Virtual Instruments) instrument drivers comply with the IVI Foundation specifications. Throughout the remainder of this manual, the term IVI is used to refer to National Instrument implementation of IVI instrument drivers, an architecture for these drivers, a component engine that makes it work, and a library that is an API to that engine. Following are the major features of the IVI instrument driver architecture from National Instruments:

- A standard, well-defined structure for the external interface to an instrument driver that is 100 percent VXI*plug&play* compatible.

- 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.

- Multiplatform capability. The IVI engine is available under Windows 2000/NT/98/95 and Sun Solaris 2. Under Windows 2000/NT/98/95, 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, there are detailed descriptions of inherent IVI attributes. Refer to Chapter 3, *Developing an Instrument Driver*, and Chapter 4, *Attribute Editor*, for information on the LabWindows/CVI wizards that help you create and modify IVI instrument drivers. Refer to Chapter 8, *Programming Guidelines for Instrument Drivers*, for detailed information on instrument driver source code. Refer to the *LabWindows/CVI Online 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:

- **State-caching**—You can avoid sending redundant commands to the instrument. If you try to set an attribute to a value that it already has, the IVI engine skips the command.

- **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.

You 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 they 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. Also, 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 more acceptable to 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.`

The following diagram in Figure 2-1 illustrates this process.
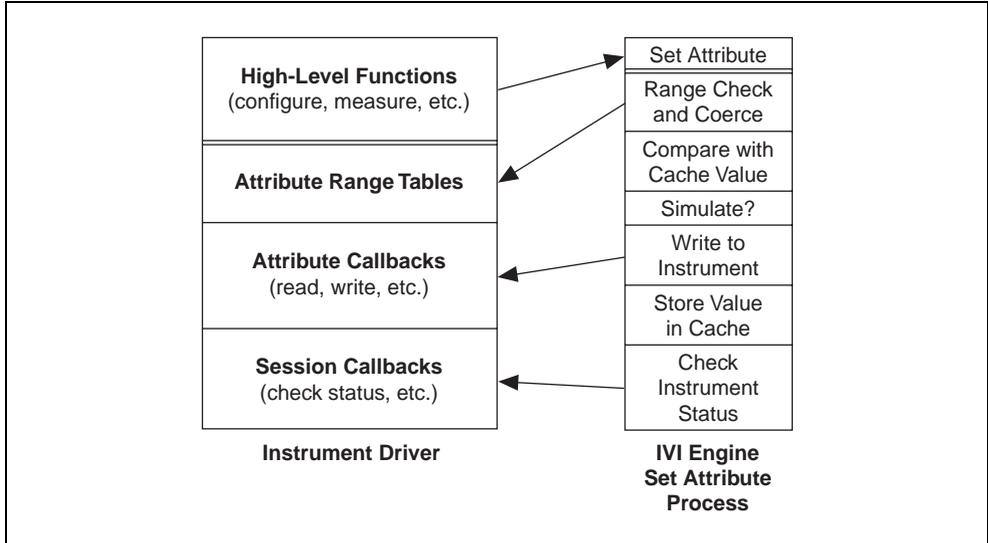
Figure 2-1. IVI Driver Operation Diagram

How You Program 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. They 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. It is best to set these attributes through the

*Prefix*_InitWithOptions function so that your settings are in effect during the
initialization procedure. This is particularly important 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 disable 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. 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 upsetting 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 VXI*plug&play* standard requires that each instrument driver contain the following
seven functions, where *Prefix* is the instrument prefix for the particular driver:

*Prefix*_init
*Prefix*_close
*Prefix*_reset
*Prefix*_self_test
*Prefix*_revision_query
*Prefix*_error_queury
*Prefix*_error_message

IVI drivers must contain the following additional functions:

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

```
Prefix_InvalidateAllAttributes
Prefix_GetNextCoercionRecord
Prefix_ReadInstrData
Prefix_WriteInstrData
```

Refer to the *LabWindows/CVI Online 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 in 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* 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 a subset of the inherent IVI attributes and most or all its instrument-specific attributes. A specific instrument driver that complies with a standard class definition also exports the class 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`, and `ViAddr`. Only hidden attributes can be of type `ViAddr`.

Refer to the *Inherent IVI Attributes* section later 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. This 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 Online Help*.

The *LabWindows/CVI Online 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 exactly what each of these functions does. The description of the `Ivi_SetAttribute` functions is particularly extensive.

Callbacks

The IVI engine contains a sophisticated attribute state-caching mechanism. Each specific instrument driver, on the other hand, contains the knowledge of how to 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 encapsulates its knowledge of how to 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 it reflects the current state of the instrument. Depending on these and other 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 later in this chapter for a description of the six attribute callbacks.

The IVI engine also allows an instrument driver to install three callback functions that are global to an entire IVI session: an operation complete callback, a check status callback, and a buffered I/O callback. Refer to the *Session Callback Functions* section later 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 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. This includes 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
7	0x0080	IVI_VAL_NO_DEFERRED_UPDATE
8	0x0100	IVI_VAL_DONT_RETURN_DEFERRED_VALUE
9	0x0200	IVI_VAL_FLUSH_ON_WRITE
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` macro 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_NO_DEFERRED_UPDATE`—Directs the IVI engine never to defer the update of the attribute when you set it, regardless of the state of the `IVI_ATTR_DEFER_UPDATE` attribute.

`IVI_VAL_DONT_RETURN_DEFERRED_VALUE`—When you query the value of an attribute and a deferred update is pending for the attribute, the flag directs the IVI engine to return a value that reflects the current state of the instrument rather than the value in the pending update. When this flag is set, it overrides the state of the `IVI_ATTR_RETURN_DEFERRED_VALUES` attribute.

`IVI_VAL_FLUSH_ON_WRITE`—By default, `Ivi_Update` does not send the `IVI_MSG_FLUSH` message to the buffered I/O callback until it completes processing all the deferred updates. This flag indicates that after processing a deferred update for the attribute, `Ivi_Update` must send the `IVI_MSG_FLUSH` to the buffered I/O callback to flush the I/O buffer.

`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.

There are three types of range tables. 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. This is useful when an instrument supports a set of ranges, each of which you must specify to the instrument using one discrete value. The discreteOrMinValue holds the minimum value of the range. maxValue holds the maximum value. coercedValue holds the discrete value that represents the range.

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

When the IVI Library searches through a ranged or coerced range table for an entry that matches a value, it returns the first entry that satisfies the following two conditions:

- The value is greater than or equal to the discreteOrMinValue field of the entry.
- The value is less than or equal to the maxValue field of the entry.

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 help it 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 that the write callback uses 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 in the IviRangeTable structure to store the format string for the instrument command.

The hasMin and hasMax fields indicate whether, as a whole, the table contains a meaningful minimum value and a meaningful 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 meaningful 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, it 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`, you 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, it 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 to such an extent that you must create a very 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 must also 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. It is convenient to 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. If it is a coerced range table, 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. If you want 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 appropriate 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.

You specify the degree of precision for an attribute when you call `Ivi_AddAttributeViReal64`. 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. You can obtain or modify the degree of precision for an attribute by calling `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
```
$$\text{if } |b| < 10^{-(d-1)}$$
```
        then a == b.
else /* a != 0 */
```
$$\text{if } \frac{|a-b|}{|a|} < 10^{-(d-1)}$$
```
        then a == b
```

IVI State-Caching Mechanism

When the state-caching mechanism is enabled, the IVI engine maintains a software copy of what it believes to be the current instrument setting for each attribute. If the IVI engine believes 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 does not believe the cache value reflects 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. It 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, on the other hand, 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 asks the driver to send the new setting to the instrument. It does this by invoking the write callback for the attribute. 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 responsibility of the user to set the instrument to a known state. Typically, the user does this by calling the instrument driver high-level configuration functions.

Special Cases

Unfortunately, instrument command sets do not always map perfectly onto the state-caching model. In such cases, the instrument driver developer 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. This is unwise. 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 earlier in this chapter explains, it is the responsibility of the user 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 in such a way 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. It can do this 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. Whenever 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 it 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. This 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 do this by using a coerce callback. The easiest way to do this 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* section earlier in this chapter and the *Coerce Callback* section later 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 earlier in 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, that 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 it sends it 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 must translate 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 must translate 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. It 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. If you want 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 it invokes the check callback. The job of the coerce callback is to convert 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 define 10.0, 100.0, and 1,000.0 as the maximum reading voltage. If an 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 and 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. For this to occur, the instrument driver must coerce the value before it sends it to the instrument. Thus, 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, that 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, and 1,000.0. However, another DMM might accept 10.0, 50.0, 100.0, 500.0, and 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 might seem to require value coercion but, in fact, 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 it sends a new value to the instrument and after it receives the current setting from the instrument. Thus, it is best to do the translations in the read and write callbacks for the attribute.

The easiest way to implement coercion is through a coerced range table. You can 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. If you want 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 nonzero 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 earlier in this chapter for more information on the compare callback function.

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

Range Table Callback

The range table callback allows you to determine dynamically which static range table you want to use for an attribute. It also allows you to modify the contents of a dynamic range table that you create with `Ivi_RangeTableNew`. Refer to the *Static and Dynamic Range Tables* section earlier 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 three callback functions that are global to an entire IVI session: operation complete, check status, and buffered I/O.

Operation Complete Callback

The purpose of the operation complete callback is to wait until the instrument has finished 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 can cause 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 OPCCallback(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 has encountered 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 CheckStatusCallback(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 it is, `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.

Buffered I/O Callback

The IVI engine allows an instrument driver to defer the transmission of new attribute values during a sequence of calls to `Ivi_SetAttribute` functions. The driver does this by enabling the `IVI_ATTR_DEFER_UPDATE` attribute. The IVI engine does not call the write callbacks. Instead, it keeps track of the deferred values. When the driver calls `Ivi_Update`, the IVI engine invokes the write callbacks for all the attributes that have deferred updates pending.

During the batch update, `Ivi_Update` invokes the buffered I/O callback with several different messages. This allows the instrument driver to buffer all the instrument commands and to send the buffered commands all at once. For each session you create, the IVI engine installs a default buffered I/O callback that works for VISA I/O.

The buffered I/O callback is optional. You can remove the default buffered I/O callback or substitute another callback by calling `Ivi_SetAttributeViAddr` on the `IVI_ATTR_BUFFERED_IO_CALLBACK` attribute.

The buffered I/O callback must have the following prototype:

```
ViStatus _VI_FUNC BufferedIOCallback(ViSession vi, IviMessage msg);
```

The **msg** parameter can be any of the values in Table 2-2.

Table 2-2. Possible Values for the msg Parameter

Defined Constant	When Ivi_Update Sends the Message
IVI_MSG_START_UPDATE	At the beginning of the batch update.
IVI_MSG_END_UPDATE	At the end of the batch update.
IVI_MSG_SUSPEND	Before invoking the operation complete and check status callbacks and before invoking a read callback when a write callback calls Ivi_GetAttribute.
IVI_MSG_RESUME	After invoking the operation complete and read callbacks.
IVI_MSG_FLUSH	After invoking the write callback for an attribute for which the IVI_VAL_FLUSH_ON_WRITE flag is set and at the end of the batch update if any write callbacks executed after the most recent flush message.

The buffered I/O callback can receive multiple, nested IVI_MSG_SUSPEND and IVI_MSG_RESUME messages.

For more information, refer to the *Deferred Updates* section later in this chapter and to the function description for Ivi_Update in the *LabWindows/CVI Online Help*.

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 must also 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. This is true 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 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 National Instruments Measurement & Automation Explorer. The application program can then reference the virtual channel names rather than specific driver channel strings.

Refer to the *Configuration Entries* section later 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 function 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_Measure`, `FL45_Configure`, and `FL45_ConfigureTrigger`.

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 8, *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 these guidelines.

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` or by setting the `RangeCheck` tag in the **optionsString** parameter of *Prefix*`_InitWithOptions` to `VI_FALSE`.

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` or by setting the `QueryInstrStatus` tag in the **optionsString** parameter of `Prefix_InitWithOptions` to `VI_FALSE`.

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 earlier 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`. It 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 it performs direct instrument I/O, it must call `Ivi_SetNeedToCheckStatus` with `VI_TRUE`. If you use the **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. This is useful when the instrument is not available, for example, 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` or by setting the `Simulate` tag in the **optionsString** parameter of the `Prefix_InitWithOptions` to `VI_TRUE`.

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, it merely records the values you set, and it returns these 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 have to take any action to support simulation. If however, the high-level function performs direct instrument I/O, it 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.

⚠️ **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. This includes check callbacks and coerce callbacks.

Multithread Safety

Users can use IVI instrument drivers in multithreaded applications. Multiple execution threads can use the same IVI instrument 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 all 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.

⚠ **Caution**   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.

Deferred Updates

When a high-level driver function makes multiple calls to the `Ivi_SetAttribute` functions, it can postpone the actual transmission of the new attribute values to the instrument. Later, the driver can cause all the updates to occur at once. This capability is called *deferring updates*.

Typically, it is not necessary to defer updates. Deferring updates can be useful in cases where the overhead of initiating instrument I/O is very high. By deferring the updates, you can buffer multiple instrument commands into one I/O action.

You defer attribute updates by setting the `IVI_ATTR_DEFER_UPDATE` attribute to `VI_TRUE` and then calling any of the `Ivi_SetAttribute` functions one or more times. You call the `Ivi_Update` function to process the deferred updates. `Ivi_Update` performs the updates in the same order in which you called the `Ivi_SetAttribute` functions. Deferral of updates is a capability for instrument driver developers, not users. Instrument driver functions that the user can call must never leave `IVI_ATTR_DEFER_UPDATE` in the enabled state.

When you call one of the `Ivi_SetAttribute` functions on a particular attribute, it checks the state of the `IVI_ATTR_DEFER_UPDATE` attribute and the state of the `IVI_VAL_NO_DEFERRED_UPDATE` flag for the attribute you are trying to set. If `IVI_ATTR_DEFER_UPDATE` is enabled and the `IVI_VAL_NO_DEFERRED_UPDATE` flag is 0, the `Ivi_SetAttribute` function performs only the following actions:

- Checks that the attribute is writable
- Checks the validity of the value you specify
- Coerces the value, if appropriate
- Posts a deferred update

You can call `Ivi_AttributeUpdateIsPending` to determine whether an attribute has a deferred update pending on a particular channel.

When you call `Ivi_Update`, it performs all the deferred updates for the session in the order in which the calls to the `Ivi_SetAttribute` functions occurred. Depending on the I/O method the specific driver uses to communicate with the instrument, the driver might have to configure some I/O buffering capabilities at certain points during the processing of the updates. `Ivi_Update` handles this by calling the buffered I/O callback for the session with various messages during the update process. By default, the IVI Library

installs `Ivi_DefaultBufferedIOCallback` as the buffered I/O callback.
`Ivi_DefaultBufferedIOCallback` works for instrument drivers that use VISA I/O.
You can change or remove the buffered I/O callback by setting the
`IVI_ATTR_BUFFERED_IO_CALLBACK` attribute.

For detailed information on what `Ivi_Update` and `Ivi_DeferredBufferedIOCallback`
do, refer to the `Ivi_Update` function description in the *LabWindows/CVI Online Help*.

Configuration Entries

When a user opens an IVI session through a class driver, the class driver must be able to
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 Measurement & Automation Explorer.

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

The inherent IVI attributes are the 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 Some of the inherent attribute names are subject to change by the IVI Foundation. Refer to the release notes for the latest information regarding these 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
Buffered I/O Callback	IVI_ATTR_BUFFERED_IO_CALLBACK
Deferring Instrument Updates	
Defer Update	IVI_ATTR_DEFER_UPDATE
Return Deferred Values	IVI_ATTR_RETURN_DEFERRED_VALUES
Updating Values	IVI_ATTR_UPDATING_VALUES
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

Category or Attribute	Defined Constant
Instrument Capabilities	
Supports VISA Buffered Writes	`IVI_ATTR_SUPPORTS_WR_BUF_OPER_MODE`
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`
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`

- The Session I/O category contains attributes you use to perform instrument I/O in a specific instrument driver.

- The User Options category contains attributes that the user can set to affect the behavior of instrument drivers and the IVI engine.

- The Session Info category contains attributes that provide information about the instrument driver that created the session and the physical resource it is using.

- The Instrument Capabilities category contains attributes that describe various capabilities of the instrument and the driver. The IVI engine requires some of this information. Other attributes are useful for application programs.

- The Version Info category contains attributes that provide version information about the instrument driver and the IVI engine.

- The 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_BUFFERED_IO_CALLBACK

Data Type: ViAddr
Restrictions: Hidden from user

Specifies the buffered I/O callback for the session. The buffered I/O callback responds to the messages that `Ivi_Update` sends when it processes deferred updates for the session. The default value is `Ivi_DefaultBufferedI/OCallback`, which works for instruments that use VISA I/O.

Set the value to `VI_NULL` if you do not want a buffered I/O callback.

IVI_ATTR_CACHE

Data Type: ViBoolean
Restrictions: None

Specifies whether to cache the value of attributes. When caching is enabled, the IVI engine keeps track of the current instrument settings so that it can avoid sending redundant commands to the instrument. This 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`. Instrument drivers provide a `Prefix_InitWithOptions` function to allow the user to override this value.

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 whenever 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 whenever 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 prefix can be up to a maximum of eight characters.

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 whenever 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 whenever you attempt to set or get this attribute.

IVI_ATTR_DEFER_UPDATE

Data Type: ViBoolean
Restrictions: Hidden from user

This attribute specifies whether to defer the actual updating of the physical instrument when you call an `Ivi_SetAttribute` function. If so, the IVI engine performs all the deferred updates for the session when you call `Ivi_Update`.

Typically, it is not necessary to defer updates. Deferring updates can be useful when the overhead of initiating instrument I/O is very high. By deferring updates, you can buffer multiple instrument commands into one I/O action.

Only instrument driver developers use this attribute, and they do so on a temporary basis around a sequence of calls to `Ivi_SetAttribute` functions. Instrument driver functions that the user calls must never return with `IVI_ATTR_DEFER_UPDATE` in the enabled state.

The default value is `VI_FALSE`.

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 of this is specifying a particular instrument model from among a family of instruments that the driver supports. This 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 that concerns 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. When the user opens an instrument session through a class driver, the user can override the default value by specifying a value in the ivi.ini configuration file.

If the user opens an instrument session through a specific driver, the IVI engine generates an error whenever 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, it 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 whenever 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. It 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 describes an error condition. A positive value describes a warning condition and indicates 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`. Specific drivers provide a `Prefix_InitWithOptions` function to allow the user to override this value. When the user opens an instrument session through a class driver, the user can override both of these values by specifying a value 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`. Specific drivers provide a *Prefix*_InitWithOptions function to allow the user to override this value. When the user opens an instrument session through a class driver, the user can override both of these values by specifying a value 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`. Specific drivers provide a *Prefix*_InitWithOptions function to allow the user to override this value. When the user opens an instrument session through a class driver, the user can override both of these values by specifying a value 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 whenever you attempt to set or get this attribute.

IVI_ATTR_RETURN_DEFERRED_VALUES

Data Type: ViBoolean
Restrictions: Hidden from user

When you call `Ivi_GetAttribute` on an attribute that has a deferred update pending, this attribute specifies whether the IVI engine returns the deferred value or the value that represents the actual state of the instrument.

The driver can prevent the IVI engine from returning deferred values for a particular attribute by setting the `IVI_VAL_DONT_RETURN_DEFERRED_VALUE` flag for the attribute.

The default value is `VI_TRUE`.

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 hard-coded 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. Specific drivers provide a *Prefix*_InitWithOptions function to allow the user to override this value. When the user opens an instrument session through a class driver, the user can override both of these values by specifying a value 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 prefix can be up to a maximum of 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. When the user opens an instrument session through a class driver, the user can override the default value by specifying a value 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_SUPPORTS_WR_BUF_OPER_MODE

Data Type: ViBoolean
Restrictions: Hidden from user

A specific driver sets this attribute to indicate whether the instrument can accept commands and data that the driver sends to it using VISA buffered I/O. The Ivi_DefaultBufferedIOCallback function uses this information to determine whether

it must temporarily modify certain VISA I/O options while the `Ivi_Update` function processes deferred updates.

The default value is `VI_FALSE`.

IVI_ATTR_UPDATING_VALUES

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

Returns `VI_TRUE` if and only if the `Ivi_Update` function is currently processing deferred updates for the session.

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

Developing an Instrument Driver

This chapter explains the proper procedure for developing an instrument driver.

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. The second half of this chapter describes many of the details for developing instrument drivers, such as defining driver functions, data types, and parameters. Refer to this information if you do not use the wizard to automate your driver development, if you want to add more functions to a driver after you use the wizard, or if you would like a more detailed understanding of instrument driver concepts.

General Guidelines

The following general guidelines help you 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 developer wizard to create your driver. You initiate the wizard by selecting **Tools»Create IVI Instrument Driver**. The wizard uses standard instrument templates for oscilloscopes, 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 carefully. 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. You initiate the attribute editor by selecting **Tools»Edit Instrument Attributes**.

- Follow the specific steps in this chapter to write your instrument driver. Each step directs you to subsequent chapters for more detailed information and further guidelines. Read all chapters referenced within each step before you perform the tasks outlined in the step.

Writing an Instrument Driver

You can develop the pieces of an instrument driver in several different sequences. More detailed information about how to perform the individual steps in the procedure appears later in this chapter and in subsequent chapters. You can use the Instrument Driver Development Wizard and the attribute editor to automate much of this process.

When you use the wizard to build a driver for an oscilloscope, multimeter, function generator, switch, or power supply, you can select a predefined template that defines common functions and attributes for these types of instruments. The wizard generates a function panel (`.fp`) file, a source (`.c`) file, an include (`.h`) file, and a `.sub` file for you, automating much of the tedious effort required when writing a driver.

To write the driver for your specific instrument, we recommend the following procedure:

1. Name the instrument driver.

2. Define the attributes (automated by the wizard when you use a template).

3. Define the instrument functions and function classes (automated by the wizard when you use a template).

4. Create a function tree for the instrument driver, adding help information to the top level of the tree (automated by the wizard when you use a template).

5. For each function in the driver:

   a. Define the parameters to the function, including variable types and limits, and error codes (automated by the wizard when you use a template).

   b. Create the function panel for the function, including help information for the panel and for each control (automated by the wizard 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 (automated by the wizard 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. You accomplish this with an instrument prefix that you assign when you create the instrument driver. Insert this 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.

For example, suppose you write an instrument driver for the Fluke 8840A digital multimeter. If you choose the instrument prefix `fl8840a`, the files that comprise the instrument driver would be `fl8840a.c`, `fl8840a.h`, `fl8840a.fp`, `fl8840.sub`, and `fl8840a.doc`. Furthermore, the driver function names each have the prefix `fl8840a` added to them, for example, `fl8840a_trigger`.

Note   The instrument prefix must have eight characters or less. LabWindows/CVI adds an underscore (`_`) separator to the eight-character prefix before appending the function name to it.

How to Use 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. The wizard asks you for this information.

Driver Information

Instrument Driver Name: _____

Prefix: _____ (8 characters or less)

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

GPIB Address: _____

To start the wizard, select **Tools»Create IVI Instrument Driver** of the Project window. Follow the instructions on each panel of the wizard and enter the information from the worksheet when prompted. The number and types of panels that appear vary according to the selections you make.

Selecting an Instrument Driver Template

After you click on the **Next** button on the initial wizard panel, you see the following panel.
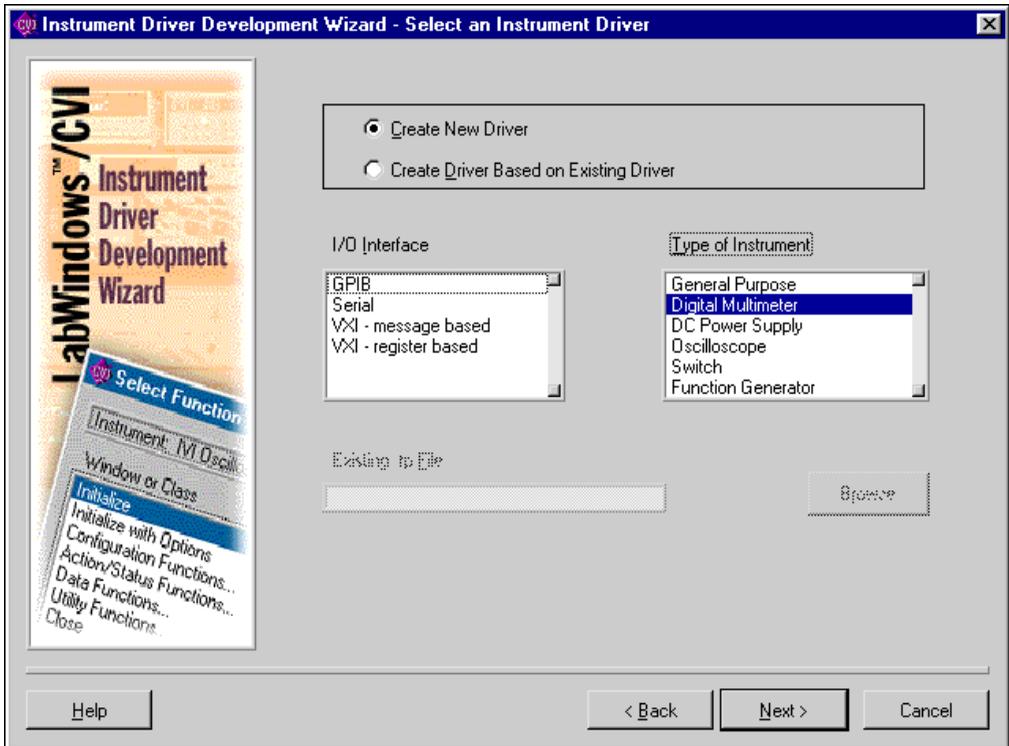
Figure 3-1. Instrument Driver Wizard Selection Panel

You can choose to copy an existing driver or create a new driver. 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. It 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.

Most predefined instrument templates define a complete driver architecture with functions and attributes for a particular type of instrument or device. An exception to this 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 VXI*plug&play* and IVI require. Use the General Purpose template only when no predefined instrument templates apply to your instrument.

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 on the **Run Tests** button. 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, a report appears that describes the success or failure of each operation. If any failures occur, you can click on the **Back** button to return to the appropriate wizard panel. Update the information and try the tests again.

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

After the wizard displays the newly created files, it gives you the option of launching the attribute editor, which National Instruments 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 Chapter 5, *Function Tree Editor*, Chapter 6, *Function Panel Editor*, and Chapter 7, *Adding Help Information*, 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. Now 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 an intuitive, hierarchical 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 on the Attribute control to display the Select Attribute Constant dialog box, as shown in Figure 3-2. You can select each attribute and view the possible values for that attribute in the lower half of the dialog box.
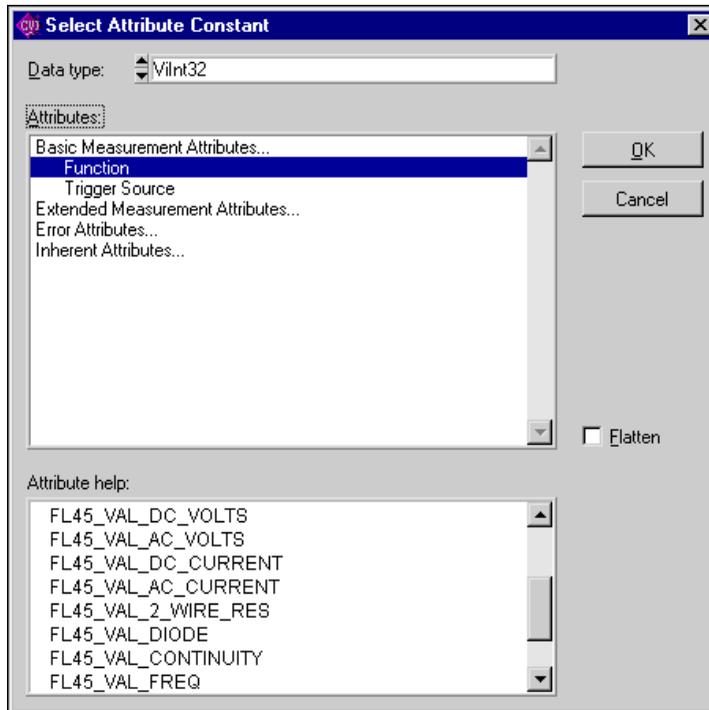
Figure 3-2. Select Attribute Constant Dialog Box

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

You can add information to the `.sub` file when you select **Tools»Edit Instrument Attributes**. The command brings up 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, intuitive constant names for the attributes and attribute values you driver uses. It is necessary to 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 multiple-point measurements from 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.

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.

Instrument Driver Fundamentals

If you want to add functions to a driver, or if you develop a driver without using the Instrument Driver Development Wizard, you must know the instrument driver fundamentals in this section. Although the wizard is the preferred method for developing a driver for an instrument that you control through a GPIB, VXI, or serial interface, many users develop instrument drivers for other purposes, such as common utility functions, analysis algorithms, or for controlling custom hardware that does not fit the IVI instrument driver model for GPIB or VXI devices.

Defining the Instrument Functions

An instrument driver exports a set of functions that programmers can use to control an instrument. To make the set of functions easy to use, you must organize them into a logical structure. Group related functions into categories or classes. For complex instruments, group related classes into higher-level classes. For very complex instruments that incorporate multiple personalities, you might consider creating multiple instrument drivers.

Structuring Functions in an Instrument Driver

If you use the wizard with a predefined instrument template, the wizard creates a function hierarchy for you. Otherwise, you must define and structure the driver functions on your own.

The three implementations of a single instrument driver hierarchy in this section show you some options for structuring functions. In this example, the driver includes seven functions with which to program the instrument.

The first implementation gives the user a simple linear list of all available functions.

```
instrumentA(1)
    function1
    function2
    function3
    function4
    function5
    function6
    function7
```

The second implementation breaks the functions into two function classes.

```
instrumentA(2)
    function_class1
        function1.1
        function1.2
        function1.3
        function1.4
    function_class2
        function2.5
        function2.6
        function2.7
```

The third implementation treats the two function classes as two distinct instruments.

```
instrumentA(3.1)
    function1
    function2
    function3
    function4

instrumentA(3.2)
    function5
    function6
    function7
```

To successfully structure the functions for your instrument, you must determine who will use the instrument driver and how they will use the instrument. Define functions that stand alone to perform a useful action. For example, it might at first seem logical to use the functions `SetDMMRange` and `SetDMMFunction` for setting the range and function of a multimeter. However, a more useful function might be `ConfigureMeasurement`, for setting up multiple parameters.

Defining the Hierarchy of Functions

It is important to design the function hierarchy for the instrument driver carefully. When you do, the user can identify the functions required by the desired action without the burden of choosing from a long list of unrelated functions.

The concept of function classes is only apparent to the user from within the LabWindows/CVI development environment. The application program calls all functions within an instrument driver the same way, regardless of which function class they are in.

Defining the Function Parameters

To design the code for an instrument driver function, you must first establish its parameters.

Function parameters provide input information to the function and output variables where the function can store its results. Output parameters often contain values that the function reads from the instrument and formats for the user.

Data Types

You must define a data type for each parameter in each instrument driver function. All data types the instrument driver uses must be intrinsic C data types or data types that you define in the `.h` file and list in the `.fp` file. You specify the data type of a parameter when you create its corresponding control on a function panel. This data type must also be consistent with the function prototypes in the instrument driver header file.

LabWindows/CVI uses the data type information to implement the variable declaration and run-time checking capabilities when users operate function panels. When you declare a variable from a function panel, LabWindows/CVI presents options based on the data type you specify for the function panel control. When you run a function from a function panel, LabWindows/CVI verifies that the data type of the control matches the prototype of the function.

Data types are divided into three classes: predefined data types, user-defined data types, and VISA data types.

Predefined Data Types

Predefined data types are available by default in the LabWindows/CVI environment. The predefined data types consist of intrinsic C data types and meta data types that LabWindows/CVI defines.

Intrinsic C Data Types

The intrinsic C data types that LabWindows/CVI defines are as follows:

```
int
long
short
char
unsigned int
unsigned long
unsigned short
unsigned char
int []
long []
short []
char []
unsigned int []
unsigned long []
unsigned short []
unsigned char []
double
float
double []
float []
char *
char *[]
void *
```

When you create a control to represent an array of data, make the data type an intrinsic C data type that ends with the square brackets, [ ]. Do *not* select a data type that ends with an asterisk (*). The brackets tell LabWindows/CVI that the control represents an array of data, not a pointer. LabWindows/CVI can then perform the appropriate variable declaration and run-time checking capabilities when the user operates the function panel.

When you define a function panel control with an intrinsic C data type, variables the user declares in the control through the **Declare Variable** command appear with that data type in the dialog box. You must define the parameter with the same data type when you prototype the function in the instrument driver include file.

Meta Data Types

The meta data types are useful for parameters through which users can pass arguments of more than one data type. The meta data types are `Numeric Array`, `Any Array`, `Any Type`, and `Var Args`. Each of these data types defines a set of multiple allowable data types. When the user executes the **Declare Variable** command on a control defined with a meta data type, the user can select from a list of the allowable data types.

Numeric Array

Numeric Array specifies a parameter that can be any of the intrinsic C numeric array data types. You must define the parameter as `void *` in the function prototype. An example of a Numeric Array data type is in the `PlotX` function of the User Interface library. The `PlotX` function plots the values of any intrinsic C numeric array data type to a graph control on a user interface panel. On the function panel, the X Array control is of type `Numeric Array`. X Array is defined as `void *` in the following function prototype.

```
int PlotX(int panel, int control, void *xArray, int numPoints,
                        int xDType, int plotStyle,
                        int pointStyle, int lineStyle,
                        int pointFreq, int color);
```

Any Array

Any Array specifies a parameter that can be any of the intrinsic C or user-defined array data types. You must define the parameter as `void *` in the function prototype. An example of an `Any Array` data type is in the `memcpy` function of the ANSI C library. This function copies a specified number of bytes from a target buffer of any type to a source buffer. In the function panel, the first parameter is the Target Buffer, which is of type `Any Type`. The Target Buffer is defined as `void *` in the function prototype.

```
void *memcpy(void *, const void *, size_t);
```

Any Type

Any Type specifies a parameter that can be any of the intrinsic C or user-defined data types. If the parameter is an output parameter, you must define it as `void *` in the function prototype. If the parameter is an input parameter, you must define it as `"..."` in the function prototype, and it must be the last parameter in the function. The Value output parameter of the `GetCtrlAttribute` function in the User Interface Library is an example of the `Any Type` data type. The function obtains the value of a particular attribute for a particular user interface control. Different attributes have different data types. Users pass the attribute value parameter by reference, and it is therefore a pointer. Consequently, the attribute value parameter is of type `Any Type` and is defined as `void *` in the following function prototype shown below.

```
int GetCtrlAttribute(int panel, int control, int attribute,
                        void *value);
```

The Value parameter of the `SetCtrlAttribute` function also applies to attributes of different data types, but it is an input rather than an output parameter. Users pass this parameter by value rather than by reference, and thus it can have different sizes. For example, it might be an `int` or a `double`. Consequently, the attribute value parameter is of type `Any Type` and is defined as `"..."` in the following function prototype.

```
int SetCtrlAttribute (int panel, int control, int attribute, ...);
```

Var Args

Var Args specifies a variable number of parameters that can be any of the intrinsic C or user-defined data types. You must define the parameters as `"..."` in the function prototype. The `printf` and `scanf` functions in the ANSI C library have examples of the `Var Args` data type. Following the format string parameter in each function, you can specify one or more parameters of different data types to match the type specifiers in the format string. In `printf`, the parameters are passed by value. In `scanf`, they are passed by reference and thus are really pointers. For both functions, one `Var Arg` function panel control is used, and `"..."` appears in the following function prototypes.

```
int printf (const char *, ...);
int scanf (const char *, ...);
```

User-Defined Data Types

LabWindows/CVI also lets you define data types and use them in function panels. You must declare user-defined data types in the function panel file of an instrument driver, and you must define the data type in the include file for the driver. Declare user-defined data types with the Data Types command box in the Function Panel Editor.

For example, you can define a `waveform_var` data type for an instrument driver to represent waveform data. This `waveform_var` data type could be a structure that contains an array of `doubles` to represent the individual points in the waveform, a `float` for the time of the first point, and a `float` for the time between points.

Creating a User-Defined Data Type

Complete the following steps to create a user-defined data type for use in a function panel.

1. Define the data type with a `typedef` statement in the instrument driver header file.

2. Add the data type to the instrument driver function panel file using the **Options»Data Types** command in the Function Panel Editor.

3. Using the previous example of the `waveform_var` data type, include the following code in the include file for the instrument driver.

    ```
    typedef struct {

    double waveform_arr [500];
    ```

```
float t_zero;

float t_delta;

} waveform_var;
```

4. Make the `waveform_var` data type available in the function panel file. Select
    Options»Data Types in the Function Panel Editor and enter

    `waveform_var`

    in the Type box of the Edit Data Type List dialog box. Then click on **Add**.

Now you can select the `waveform_var` data type when you create function panel controls for
this instrument driver. Also, users can interactively declare a variable of `waveform_var` data
type from any function panel control that you define as `waveform_var`.

Refer to Chapter 6, *Function Panel Editor*, for a discussion of the Edit Data Type List dialog
box.

User-Defined Array Data Types

Use care when you declare user-defined data types that are arrays. If you want to define a
user-defined array data type, square brackets `[ ]` must appear at the end of the type name in
the Edit Data Type List dialog box. The brackets enable the interactive variable declaration
and other capabilities of LabWindows/CVI function panels. For example, to declare an array
of `waveform_var` type from the example presented above, add

`waveform_var []`   (This example is correct because it includes brackets.)

in the **Type** box of the Edit Data Type List dialog box and include the `typedef` declaration
for `waveform_var` in the driver include file.

Examples of incorrect ways to define user-defined array data types are shown below.

Assume the following data type definitions are in an instrument driver header file.

```
typedef waveform_var * waveform_arr1;
typedef waveform_var waveform_arr2[100];
```

Then the following data type declarations in the Edit Data Type List dialog box are incorrect:

```
waveform_var *    (This example is incorrect because it lacks brackets.)
waveform_arr1     (This example is incorrect because it lacks brackets.)
waveform_arr2     (This example is incorrect because it lacks brackets.)
```

VISA Data Types

The VISA I/O library defines a special set of data types. The IVI Library also uses some of these data types. The data types strictly define the type and size of the parameters and therefore promote the portability of the functions to new operating systems and programming languages.

A subset of the VISA data types has been defined for use in the development of LabWindows/CVI instrument drivers and are accessible as user-defined data types. Table 3-1 shows these special data types for instrument drivers.

Table 3-1. VISA Data Types

VISA Type Name	Definition
`ViInt16`	Signed 16-bit integer
`ViInt32`	Signed 32-bit integer
`ViUInt16`	Unsigned 16-bit integer
`ViUInt32`	Unsigned 32-bit integer
`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`	An instrument driver resource descriptor (`string`)
`ViSession`	An instrument driver session handle
`ViStatus`	An instrument driver return status type
`ViBoolean`	Boolean value
`ViBoolean[]`	An array of `ViBoolean` values

To use these special user-defined data types in an instrument driver, do the following:

1. Add the VISA data types to the function panel file by using the **Options»Data Type** command in the Function Panel Editor. Then click the **Add VISA Types** button in the Edit Data Type List dialog box.

2. Include the file `vpptype.h` in the instrument driver header file.

Refer to Chapter 6, *Function Panel Editor*, for a discussion of the Edit Data Type List dialog box.

Input and Output Parameters

Because most instrument drivers are designed to control a physical instrument, the input and output function parameters often correspond to one or more of the controls on the face of the instrument.

Define output parameters as follows:

1. Review the purpose of the function to determine the inputs and outputs.

2. Choose the data type of each parameter. The data type should be one that the application program can use easily.

   a. If a parameter is an array data type, select a data type with square brackets `[ ]` at the end of the data type name.

   b. For output parameters, select the data type of the value that users pass by reference, not the pointer to that data type. When users operate function panels interactively, LabWindows/CVI knows to pass a variable by reference because the control is defined as an output.

      For example, if a function by the name of `examp_func` has an `examp_out` integer output parameter, you prototype the function in the instrument driver include file as

      ```
      examp_func (int *examp_out);
      ```

      When you create a function panel for this function, create an output control for `examp_out` and specify its data type as `int`, not as `int *`. When a user declares variables interactively from the function panel, LabWindows/CVI creates an `int` variable and automatically puts an `"&"` in front of the variable name to pass it by reference.

3. Assign a meaningful name to each parameter.

Return Values

Instrument driver functions can also have a return value. Instrument drivers supplied by National Instruments use function return values to implement an error-handling mechanism. All instrument driver functions have a return value of type `ViStatus` (32-bit unsigned integer) that returns error and status information about the function call.

Required Instrument Driver Functions

All instrument drivers must contain functions that perform the following operations:

- *Prefix*_init
- *Prefix*_close
- *Prefix*_error_message

The VXI*plug&play* standard requires the following additional functions for instrument drivers that control GPIB, VXI, or serial instruments:

- *Prefix*_reset
- *Prefix*_self_test
- *Prefix*_revision_query
- *Prefix*_error_query

IVI instrument drivers must have the following additional functions:

- *Prefix*_InitWithOptions
- *Prefix*_GetErrorInfo
- *Prefix*_ClearErrorInfo
- *Prefix*_LockSession
- *Prefix*_UnlockSession
- *Prefix*_ReadInstrData
- *Prefix*_WriteInstrData

IVI instrument drivers from National Instruments also must export functions by the name of *Prefix*_IviInit and *Prefix*_IviClose, but the driver must *not* have function panels for these functions. The *Prefix*_init function calls the *Prefix*_InitWithOptions function, which in turn calls the *Prefix*_IviInit function. The *Prefix*_close function calls the *Prefix*_IviClose function.

For a description of the implementation guidelines for the required instrument driver operations, refer to the *LabWindows/CVI Online Help*.

Building the Function Tree

When users access an instrument driver from the **Instrument** menu, they can select instrument functions from one or more dialog boxes. The function tree shows the organization of the functions in dialog boxes. You use the Function Tree Editor to create the function tree. Chapter 5, *Function Tree Editor*, describes the use of the Function Tree Editor.

In addition to specifying the appearance, the function tree also contains help information that the user can access from the dialog boxes. Add this help information as you create the

tree. Chapter 7, *Adding Help Information*, explains how to add help information to the function tree.

Building the Function Panels

Users operate the function panels to execute instrument driver functions and to generate code for an application program. Each primary function requires a function panel. A secondary function can appear on one or more function panels. A function panel also can consist entirely of secondary functions. The Function Panel Editor lets you build function panels by placing controls on a blank panel in the position and order that you want them to appear. Chapter 5, *Function Tree Editor*, describes the use of the Function Panel Editor.

The Function Panel Editor also lets you add online help information for each control on a panel. Add this help information as you create each panel. Chapter 7, *Adding Help Information*, explains how to add help information to a function panel.

Writing the Function Code

After you name the function and define its parameter list, you write the code to implement the function. The *LabWindows/CVI User Manual* describes the development tools available in LabWindows/CVI for testing and debugging your code. The instrument driver you create uses full C language source code.

To develop the instrument driver source code, follow the guidelines in Chapter 8, *Programming Guidelines for Instrument Drivers*.

Operating the Driver

After you create the `.c` (`.obj` or `.dll`), `.h`, and `.fp` files, you can operate the instrument driver. Load the driver using the **Instruments»Load** command and operate every function panel that you have created. Then, use the panels to generate a sample program to verify operation of the driver. Chapter 3, *Project Window*, of the *LabWindows/CVI User Manual*, gives more details about operating instrument drivers.

Testing the Instrument Driver

Before you distribute an instrument driver, you should fully test it. Test it from within the LabWindows/CVI interactive program and as a standalone application. A suggested testing sequence for instrument drivers is outlined here.

⚠ **Caution**   Be sure to save copies of the original instrument source files in a separate directory.

1. Load the instrument driver and execute all functions from the function panels.

2. Verify correct operation of all functions.

3. Create and run a sample application program that exercises all the functions in the driver within LabWindows/CVI.

4. Verify correct operation of the application program.

5. Create and run a sample application that exercises all the functions in the driver within a standalone application.

6. Verify correct operation of the application program.

Documenting the Driver

The final step in creating an instrument driver is to document the driver. The .doc file describes the purpose of the driver, the function tree, and function panels. It also contains a function reference list that explains the syntax of each function in the driver. Chapter 8, *Programming Guidelines for Instrument Drivers*, contains guidelines and suggestions for documenting your instrument driver.

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

You invoke the Attribute Editor by selecting **Tools»Edit Instrument Attributes** in a Source, Function Tree Editor, or Function Panel Editor window. 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. Making these assumptions 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 developer 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. 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 0 or one or more defined constant names for attribute flags. If you pass multiple defined constant names, you must separate them with a vertical bar (|).

- All callback function pointer parameters to these calls must be the names of actual callback functions rather than variables.

- 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 developer 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 Online Help* for more information on `checkErr`.

Edit Driver Attributes Dialog Box

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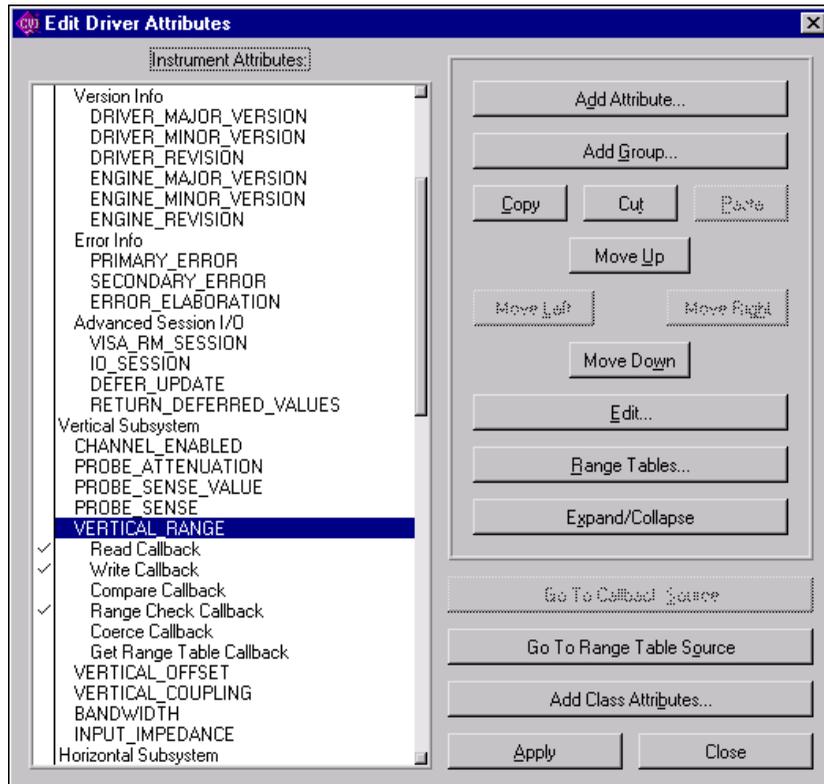
Figure 4-1. Edit Driver Attribute Dialog Box

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. It also displays the inherent IVI attributes that the `.sub` file describes. The instrument driver developer 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 or press <Enter> to display the Edit Group or Edit Attribute dialog box, depending on the selected entry.

When you select an attribute in the list box, you can expand or collapse it by pressing <Space bar> or clicking on the **Expand/Collapse** button. When you expand an attribute, a list of the different types of attribute callback functions appears under the attribute name. A checkmark next to a callback type indicates that the Attribute Editor found the name of a callback function in the `Ivi_AddAttribute` for the attribute or in a call to a function such as `Ivi_SetAttrCoerceCallbackViInt32` or `Ivi_SetAttrRangeTableCallback` that references the attribute. You can disassociate 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 <Space bar> or by clicking on the checkmark.

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.

You cannot edit, expand, cut, or copy an inherent IVI attribute. You cannot move the items within the Inherent IVI Attributes group. You cannot 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 developer 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. You also 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.

Use the **Add Attribute** button to add 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 later in this chapter for more information on the Edit Attribute dialog box.

Use the **Add Group** button to add a new group label to the list box. The command invokes 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 invoke in the `Get/Set/CheckAttribute` function panels.

Use the **Move Up/Down** buttons to move attributes up or down in the list box.

Use the **Move Right/Left** button to indent attributes left or right for logical grouping and readability.

Use the **Edit** button to modify a group or attribute. The command invokes the Edit Group or Edit Attribute dialog box for the currently selected item.

Use the **Range Tables** button to display a list of range tables in the driver. Refer to the *Adding and Editing Range Tables* section later in this chapter for more information.

Use the **Expand/Collapse** button to expand or collapse the currently selected attribute or group.

Use the **Go To Callback Source** button to jump to a callback function in the driver source file. The command finds the definition of the callback function that is currently selected in the list box. Because this closes the Attribute Editor, the Attribute Editor prompts you to apply any unsaved changes.

Use the **Go To Range Table Source** button to jump to a range table in the driver source file. The command finds the definition of the range table for the attribute that is currently selected in the list box. Because this closes the Attribute Editor, the Attribute Editor prompts you to apply any unsaved changes.

Use the **Add Class Attributes** button to add class attributes that your driver currently does not implement. This 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 execute the **Add Class Attributes** command, it uses the predefined instrument template for the class that the instrument driver developer 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.

Use the **Apply** button to update the `.c`, `.h`, and `.sub` files for the driver with the changes you have made in the Attribute Editor. Examples of changes the Attribute Editor makes are 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 callback functions or range tables in the *Prefix*_InitAttributes function even though it contains no definitions for them. When you execute the **Apply** command, the Attribute Editor creates empty definitions for them.

The first time you execute the **Apply** command 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 asks you if you want 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 might already be in the directory.

After the **Apply** command 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.

Use the **Close** button to exit the Attribute Editor. The **Close** command prompts you to apply any unsaved changes.

Adding and Editing Instrument Attributes

You add and edit instrument driver attributes by entering information in the Edit Attribute dialog box, which appears in Figure 4-2.
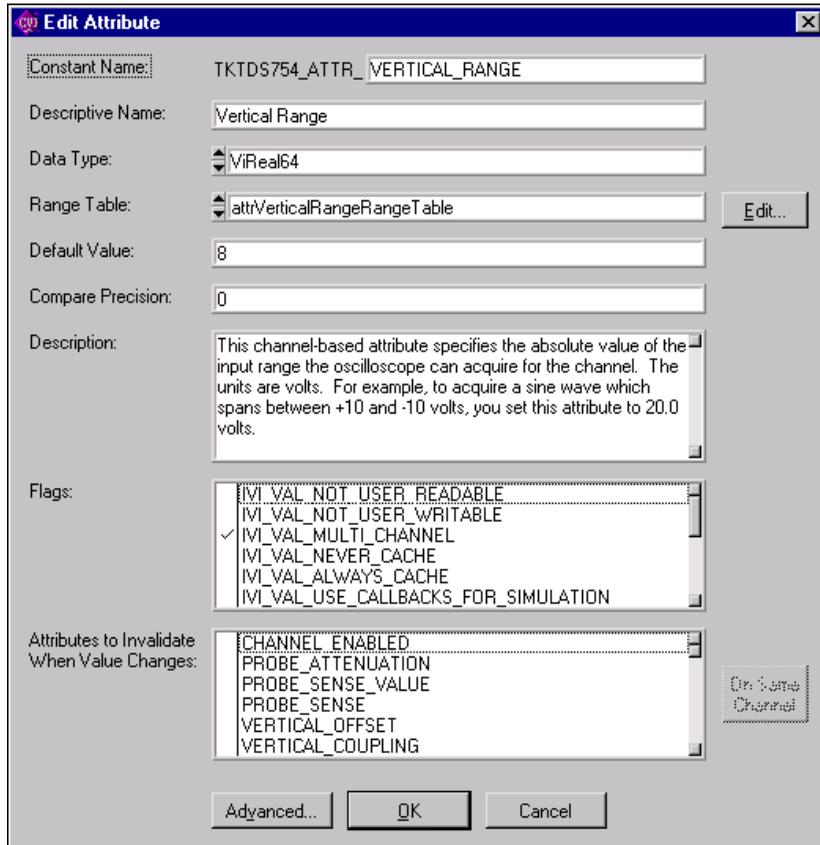
Figure 4-2. Edit Attribute Dialog Box

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 hide from the instrument driver user.

- **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.

- **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.

- **Edit button**—Lets you invoke 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 later in this chapter for more information on the Edit Range Table dialog box.

- **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 0, 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.

- **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**—Lets you 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 on 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.

- **Advanced**—Invokes the Edit Attribute Advanced dialog box, which appears in Figure 4-3.
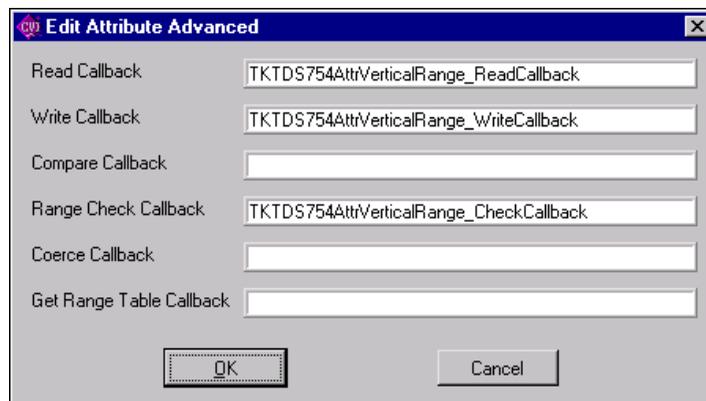
Figure 4-3. Edit Attribute Advanced Dialog Box

Use this dialog box to specify custom names for the attribute callback functions. The instrument driver developer 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.

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 press the **Range Tables** button in the Edit Driver Attributes dialog box, the Range Tables dialog box appears as shown in Figure 4-4.
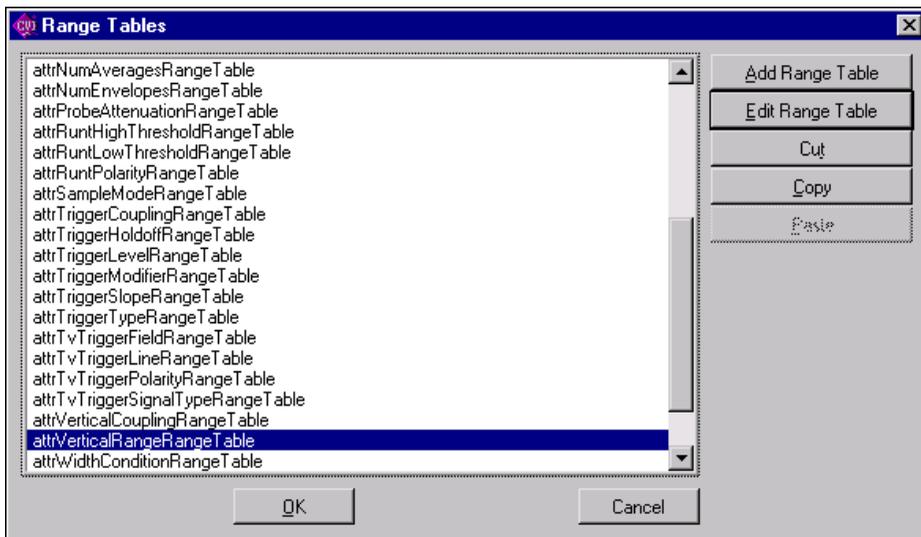
Figure 4-4. Range Tables Dialog Box

Use the **Add Range Table** button to add a new range table. The command invokes an Edit Range Table dialog box with default information.

Use the **Edit Range Table** button to edit an existing range table. The command invokes the Edit Range Table dialog box for the currently selected range table.

The Edit Range Table dialog box appears as shown in Figure 4-5.
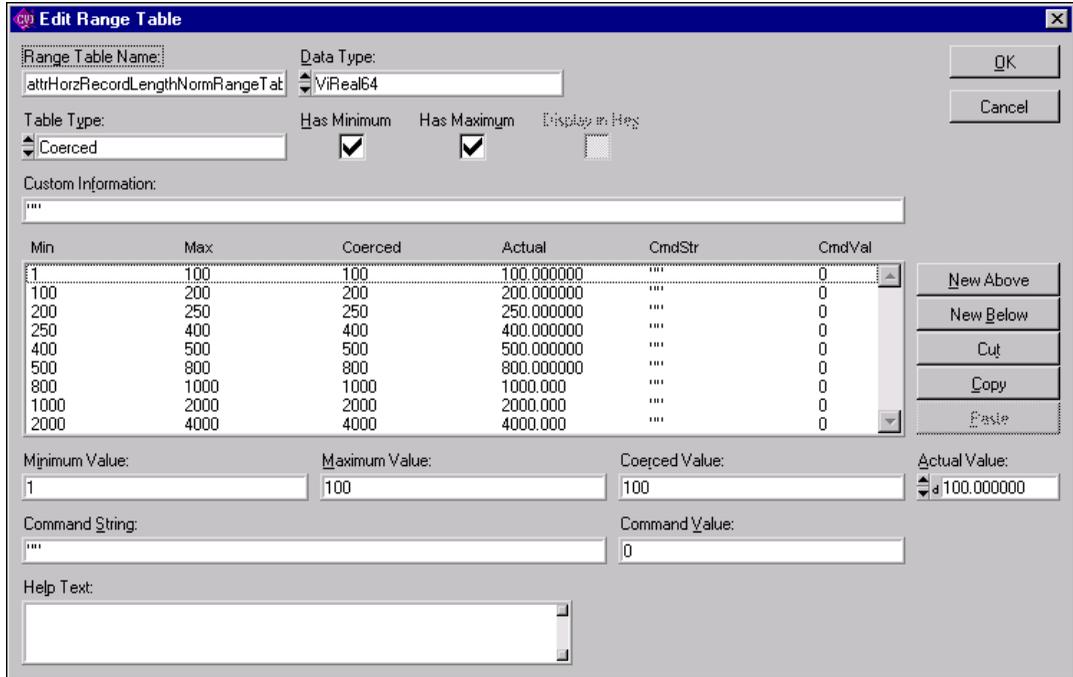
Figure 4-5. Edit Range Table Dialog Box

In the Edit Range Table dialog box, you 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 invoke 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. 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. 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 using the **New Above** and **New Below** buttons that are to the right of the list box.

- **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 invoke 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.

- **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.

Note   If you define your range tables directly in the driver source code, you still must use the Edit Range Table dialog to specify the actual values and help text for each table entry. The Attribute Editor saves this information in the `.sub` file.

5

Function Tree Editor

This chapter explains the function tree and the Function Tree Editor. It also describes the Function Tree Editor menu bar, menus, and commands.

About the Function Tree and Function Tree Editor

The function tree defines the way functions are grouped in the dialog boxes. Users access the function panels of an instrument driver through the Select Function Panel dialog box that they select from the **Instrument** menu. You use the Function Tree Editor to create and modify the function tree for an instrument driver.

To invoke the Function Tree Editor, select **File»New»Function Tree (*.fp)** or **File»Open»Function Tree (*.fp)**.

When you invoke the Function Tree Editor, a new Function Tree Editor window appears. If you selected **Open** to edit an existing function tree, the function tree for the file you selected appears in the window. To edit the function panel of an instrument driver that is loaded in the **Instrument** menu, select **Instrument»Edit**. Then highlight the name of the instrument in the selection list of the Edit Instrument dialog box and click the **Edit Function Tree** button. A function tree appears as shown in Figure 5-1.
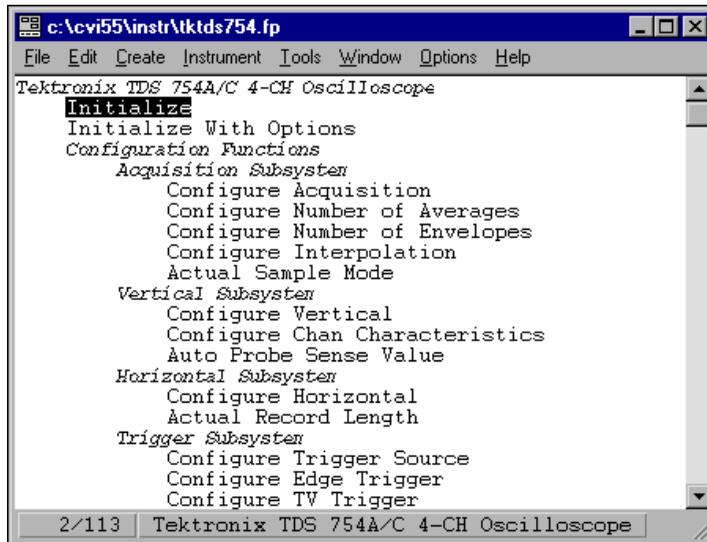
Figure 5-1. Function Tree

If you selected **New** to create a new function tree, you see a blank Function Tree Editor window.

Function Tree Editor Context Menu

The function tree context menu pops up when you right-click on a function panel window node in the Function Tree Editor. The menu appears as shown in Figure 5-2.
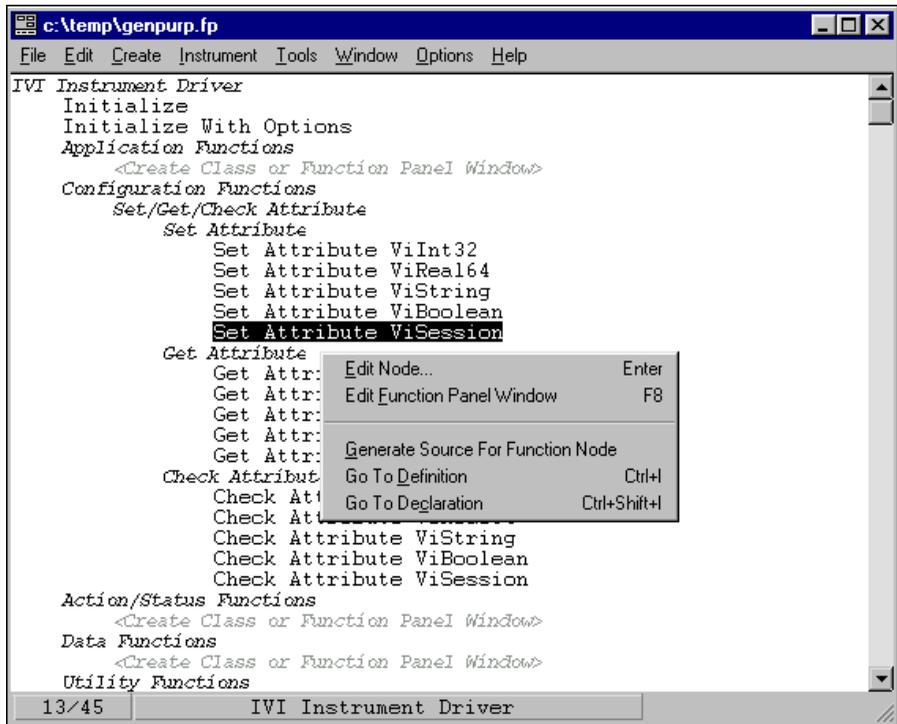
Figure 5-2. Function Tree Context Menu

The following menu options are the same or similar to options available through the menu bar selections.

- **Edit Node** is the same as the **Edit»Edit Node** command.

- **Edit Function Panel Window** is the same as the **Edit»Edit Function Panel Window** command.

- **Generate Source For Function Node** generates a function definition and declaration for the currently selected function node. If a function definition already exists in the driver source file, you are prompted for permission to update it. You can replace, insert above or below, or skip without updating. Your choice is then also applied to the declaration in the driver include file.

- **Go To Definition** opens the driver source file and jumps to the definition of the function that is currently selected in the function tree.

- **Go To Declaration** opens the driver include file and jumps to the declaration of the function that is currently selected in the function tree.

Function Tree Editor Menu Bar

You can edit an existing tree or create a new tree with the Function Tree Editor. You have the following options on the Function Tree Editor menu bar:

- **File** lets you create a new function tree, edit an existing function tree, save function panel information into a `.fp` and `.sub` file on disk, or add function panels to a project.

- **Edit** lets you modify the entries on the function tree or add help information.

- **Create** lets you create a new function tree or add new functions and classes to an existing function tree.

- **Instrument** lets you load instrument drivers, unload them, or select which function panel to edit.

- **Tools** lets you generate function definitions and declarations into your source and include files, jump to function definitions and declarations, generate `.hpp` files for the function tree, and invoke the instrument driver developer wizard and attribute editor.

- **Window** lets you select which window to make active.

- **Options** lets you select the help style, generate function prototypes, generate a `.doc` file, create a DLL project, and select whether to enable VXI*plug&play* style.

File

The **File** menu lets you create a new function tree, edit an existing function tree, save function panel information into a `.fp` and `.sub` file on disk, or add function panels to a project. The **File** menu operates like the **File** menu of the Project window. Refer to Chapter 3, *Project Window*, in the *LabWindows/CVI User Manual* for more information about the **File** menu.

For each IVI instrument driver, a `.sub` file accompanies the `.fp` file. The `.sub` file contains the information about the instrument driver attributes. You edit this information using the attribute editor. When you save the contents of a `.fp` file, LabWindows/CVI also saves the contents of the `.sub` file automatically.

Edit

The **Edit** menu lets you edit the entries in the function tree. You have the following options in the **Edit** menu.

- **Cut** deletes the selected function or class from the tree and copies it to the clipboard.

- **Copy** copies the selected function or class from the tree to the clipboard.

- **Paste Above** inserts the contents of the clipboard into the tree above the selected node.

- **Paste Below** inserts the contents of the clipboard into the tree below the selected node.

- **Edit Node** lets you edit the instrument, function, or class name on the highlighted line.

- **Edit Help** lets you add context-sensitive help information to the function tree. Refer to Chapter 7, *Adding Help Information*, to learn how to add help information.

- **Edit Function Panel Window** lets you edit the selected function panel window in the Function Panel Editor. Refer to Chapter 6, *Function Panel Editor*, for more information on using the Function Panel Editor.

- **.FP Auto-Load List** allows you to specify other instrument drivers that the current instrument driver depends on. LabWindows/CVI loads these instrument drivers automatically when you load the current instrument driver. Refer to the *.FP Auto-Load List* section later in this chapter for more information.

> **Note**   When you cut or copy a class to the Function Tree Editor clipboard, all its subclasses and functions are cut or copied as well. Similarly, when you paste the class, all its subclasses and functions are pasted.

Find

The **Find** command allows you to locate a particular text string in the function panel file. You can search for text in node names; function names; control labels; control values; item labels in ring, slide, and binary controls; message control text; and help text. When you search in help text, you cannot search in any of the other items at the same time. The search begins at the node that is currently selected.

If the **Find** command brings up a Help Editor window and you do not use the button bar, you must return to the Function Tree Editor window to continue searching throughout the panel. The **Find** command in the Help Editor window searches only within the window. You can return to the Function Tree Editor window by pressing <F7>. On the other hand, the **Find** command in the Function Panel Editor window continues searching through the entire function panel file.

Replace

The **Replace** command operates the same as the **Find** command except that you can replace the search string with another search string.

.FP Auto-Load List

The **.FP Auto-Load List** command brings up a dialog box in which you can list simple `.fp` file names. Do not include drive or directory names. When you load the current instrument driver, LabWindows/CVI tries also to load the instrument drivers identified by these `.fp` file names.

LabWindows/CVI looks for these `.fp` files in the following sequence:

1. If the `.fp` file is under the VXI*plug&play* framework directory, LabWindows/CVI looks for the `.fp` file using the following pathnames, where *vppfrmwk* is the VXI*plug&play* framework directory and *prefix* is the instrument prefix:

   vppfrmwk\support*prefix**prefix*.fp

   vppfrmwk*prefix**prefix*.fp

2. It then looks in the directory of the referencing `.fp` file.

3. It then looks for them in the instrument directories list. You edit the instrument directories list by selecting **Options»Instrument Directories** in the Project window.

4. Finally, it looks for them in the `instr` directory under the directory where LabWindows/CVI is installed.

If a `.fp` file cannot be found, the user is given a chance to look for it using a file dialog box. If the user finds the `.fp` file, the user is prompted to add the directory to the instrument directories list. The user also is given the option to add the file to the project.

If an auto-loaded `.fp` file has no classes or function panels, it does not appear in the **Instrument** menu. This is useful for support modules that contain no user-callable functions.

When the user selects **Instrument»Unload**, all auto-loaded `.fp` files are listed in the dialog. Auto-loaded instruments are not unloaded automatically when the dependent instrument is unloaded.

Create

The **Create** menu lets you create a new instrument tree or add functions and classes to an existing tree.

Instrument

The **Create»Instrument** command lets you create a new function tree. When you select **Instrument**, a dialog box appears. Enter the following information in the Create Instrument Node dialog box:

- The name of the instrument (up to 40 characters).

- The prefix that you want LabWindows/CVI to add to the beginning of each function name. The prefix cannot exceed eight characters. Do not include the underscore (_) separator in your prefix. LabWindows/CVI adds an underscore (_) separator to the prefix before appending the function name to it.

The instrument name you enter in the Create Instrument Node dialog box appears at the bottom of the Function Tree Editor window. The `Create Class or Function Panel Window` line appears beneath the instrument name. Add functions and classes to the function tree using the Function and Class commands.

Class

Use the **Class** command to add a new class to a function tree.

When you select the **Class** command, a dialog box appears. Enter the name that you want to appear in the Select Function Panel dialog box that appears when the user selects the instrument from the **Instrument** menu.

Adding a Class to an Empty Tree or Class

Complete the following steps to add a class to an empty tree.

1. Select the line that contains `Create Class or Function Panel Window`.

2. Select the **Create»Class**. The Create Class Node dialog box appears.

3. Complete the Create Class Node dialog box. The class appears in the function tree window. The new class name takes the place of the `Create Class or Function Panel Window` message on the selected line.

Inserting a Class into an Existing Tree

In the function panel hierarchy, you can insert up to eight levels of classes. Complete the following steps to insert a class into a function tree.

1. Select an existing function or class at the level you want to place the new class.

2. Select **Create»Class**. The Create Class Node dialog box appears.

3. Complete the Create Class Node dialog box. The new class is inserted on the line below the existing function or class. The class exists at the same level in the tree as the function or class that originally occupied the line.

Note   A function tree can contain a combination of up to 32,000 functions and classes.

Function Panel Window

Select **Create»Function Panel Window** to add a new function to a function tree.

When you select the **Function Panel Window** command, a dialog box appears. Enter the following information in the Create Function Panel Window Node dialog box.

1. In the Name text box, enter the name that you want to appear in the Function Panel Selection dialog box when the instrument is chosen from the **Instrument** menu.

2. In the Function Name text box, enter the actual code name used in the instrument driver for the function being added. This function name must be valid for the current language.

✎ **Note** The name of every function in an instrument driver begins with a common prefix. Do not enter the prefix of the function name. LabWindows/CVI automatically adds the prefix to each function name. You specify the prefix by selecting **Create»Instrument**.

Adding a Function to an Empty Tree or Class

Complete the following steps to add a function to an empty tree or class.

1. Select the line that contains `Create Class or Function Panel Window`.

2. Select **Create»Function Panel Window**. The Create Function Panel Window Node dialog box appears.

3. Complete the Create Function Panel Window Node dialog box. The new function name appears in place of the `Create Class or Function Panel Window` message on the selected line.

Inserting a Function into an Existing Tree

Complete the following steps to insert a function at any level in an existing function tree.

1. Select an existing function or class at the level you want to place the new function.

2. Select **Create»Function Panel Window**. The Create Function Panel Window Node dialog box appears.

3. Complete the Create Function Panel Window Node dialog box. The new function is inserted on the line below the existing function or class. The function exists at the same level in the tree as the function or class that originally occupied the line.

Instrument

Use the **Instrument** menu to load and edit an instrument driver and to edit a function in the loaded instrument driver. The **Instrument** menu operates like the **Instrument** menu on the main LabWindows/CVI menu bar except that the instrument function tree you select appears in a Function Tree Editor window. The **Instrument** menu lists the loaded instrument drivers.

Load

Select **Instrument»Load** to add a new instrument driver to the **Instrument** menu. The **Load** command operates like the **File»Open** command. When you select the **Load** command, the Load Instrument dialog box appears. Enter the appropriate information to select an existing function panel file.

Unload

Select **Instrument»Unload** to remove one or all instrument drivers from the **Instrument** menu. When you select the **Unload** command, the Unload Instrument dialog box appears. In this dialog box, you have the following options.

•   Use the mouse or the cursor keys and space bar to individually select which instrument drivers to unload.

•   Select all instrument drivers by clicking on the **Check All** button.

•   Deselect all instrument drivers by clicking on the **Check None** button.

•   Click on the **OK** button to unload the selected instrument drivers.

•   Click on the **Cancel** button to return without unloading any instrument drivers.

Edit

The **Edit** command lets you invoke the Function Panel Editor or modify the relationship between the function panel file and its associated program file. When you select **Instrument»Edit**, the dialog box shown in Figure 5-3 appears.
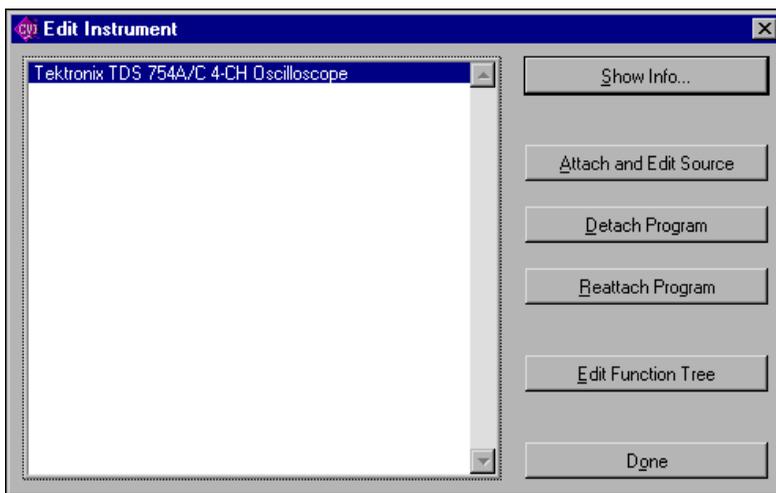
Figure 5-3. Edit Instrument Dialog Box

The Edit Instrument dialog box presents the following options:

•   **Show Info** lets you display the names of the current function panel file and the attached program file. It also shows whether these files are in the current project and if the program file is compiled. The attached program file contains the functions that are called when users operate the function panel.

- **Attach and Edit Source** searches the directory that contains the function panel file for a filename that has the same prefix as the function panel file and a `.c` extension. If the file is found, a new Source window opens with the file displayed in it, and the source file is attached to the function panel. If the file is not found, you are prompted to create a new source file and a blank Source window appears.

- **Detach Program** detaches the program file from the function panel.

- **Reattach Program** attaches a program file to a function panel. It searches the directory that contains the function panel file for a filename that has the same prefix as the function panel file and a `.obj`, `.dll`, or `.c` extension. If a file is found, the program attaches it to the function panel.

- **Edit Function Tree** invokes the Function Tree Editor.

- **Done** exits the Edit Instrument dialog box without modifying the function panel.

Tools

The Tools menu presents the following options:

- **Create ActiveX Automation Controller** generates a new instrument driver for an ActiveX Automation Server. When you select **Tools»Create ActiveX Automation Controller**, LabWindows/CVI displays the ActiveX Automation Controller Wizard.

- **Create IVI Instrument Driver** initiates the instrument driver developer wizard. Refer to Chapter 3, *Developing an Instrument Driver*, for more information on the instrument driver developer wizard.

- **Edit Instrument Attributes** initiates the attribute editor. Refer to Chapter 4, *Attribute Editor*, for more information on the attribute editor.

- **Generate IVI C++ Wrapper** generates a `.hpp` file for the function tree. This file contains the definition of a C++ class for the instrument driver. A wrapper function is generated for each function in the function tree. Required C++ member functions, such as constructors and destructors, are also generated. The generated class references classes that are defined in `ivibase.hpp` and `iviexcpt.hpp`, which are located in the `cvi\include` directory. You can edit these classes to customize the generated wrapper class. This command is available only for IVI drivers.

- **Enable Auto Replace** enables the updating of instrument driver source files to reflect changes to function names in the function tree. This option is global to LabWindows/CVI, so enabling it in one Function Tree Editor window enables it for all Function Tree Editor windows. This command is dimmed for function trees that have not yet been saved.

  When this option is enabled and not dimmed, LabWindows/CVI updates the instrument driver `.c`, `.h`, and `.sub` files to reflect changes you make to function names or the instrument prefix in the Function Tree Editor window or Function Panel Editor window. When you change a function name, you are prompted for permission to update your

instrument driver to reflect the new name. When you change the instrument prefix, you are prompted for permission to update your instrument driver to reflect the new prefix.

- **Generate New Source For Function Tree** generates function definitions and declarations into your driver source and include files. For each function panel in the function tree, the source (`.c`) file is searched for the function definition. The header (`.h`) file is searched for the function declaration. If a function definition or declaration cannot be found in the appropriate file, an empty function shell is generated in the file.

- **Go To Definition** opens the driver source file and jumps to the definition of the function that is currently selected in the function tree.

- **Go To Declaration** opens the driver include file and jumps to the declaration of the function that is currently selected in the function tree.

- **Source Code Control** contains menu items that you can use to perform operations with your source code control system. LabWindows/CVI does not provide a source code control system. If you have one that implements the standard Source Code Control Interface, you can attach a LabWindows/CVI project to your source code control system using **Options»Source Code Control Options** in the Project window.

Window

The **Window** menu lets you select which window to make active. The **Window** menu operates like the **Window** menu of the Project window. Refer to Chapter 3, *Project Window,* in the *LabWindows/CVI User Manual*, for more information about the **Window** menu.

Options

The **Options** menu lets you operate the function tree or select the help style. The **Options** menu presents the following options.

- **.FP File Format** allows you to select the format for the current function panel file as well as the default format for new function panel files. Two function panel file formats are available. Table 5-1 lists the features of the two formats.

Table 5-1. FP File Format Features

Feature	CVI 5.0.1 and earlier format	CVI 5.5 and later format
Maximum Instrument Prefix Length	8	31
Maximum Function Name Length	31	79
Maximum Class Name Length	31	79
Maximum Window Name Length	31	79
Maximum Type String Length	50	80

Table 5-1. FP File Format Features (Continued)

Feature	CVI 5.0.1 and earlier format	CVI 5.5 and later format
Default Text for Output Controls	Not supported	Supported
Function Qualifiers	Not supported	Supported
Set Precision on Numeric Controls	Not supported	Supported

- **Help Style** lets you choose the help style **New (Recommended)** or **Old (LabWindows DOS)** when you are editing context-sensitive help information of the function tree.

  The new and old help styles differ significantly. The old help style maintains compatibility with function panels created in LabWindows version 2.3 or earlier. This help style uses the DOS/IBM character set so that it can display special extended ASCII characters that many older instrument drivers use. Also, the old style gives help information for the entire function panel window, not the individual function panels within a function panel window.

  The new help style uses the standard Windows character set and gives help information for each individual function panel. In addition, the new help style automatically generates control name and data type information when displaying control help and automatically generates a function prototype when displaying function help. Also, the help text editor for the new style help uses word-wrap mode.

  Changing the help style only changes how the program interprets help information. If you use special extended ASCII characters in your help information and then change to the new style, you must change the help text to a Windows-compatible character set.

  Use the new help style whenever possible.

- **Transfer Window Help to Function Help** helps you convert your function panel from old to new style. For each function panel window, the window help text is transferred to the first function unless the function already has help text.

- **Generate Function Prototypes** creates an untitled `.h` window that contains prototypes for the functions in the function tree.

- **Generate Documentation** creates a window that contains a `.doc` file for the function panel file.

- **Generate Windows Help** creates a project file (`.hpj`) and two source files (`.rtf` and `.whh`) that you can use with Microsoft Windows Help Compiler to create a Windows help file. You are prompted to choose the output language as either C or Visual Basic.

- **Generate ODL File** creates an Object Description Language (`.odl`) file for the instrument driver. The `.odl` file can be input to the `MkTypeLib` program that comes with the Microsoft OLE 2 SDK. This is useful when you create a DLL version of the instrument driver. The `MkTypeLib` program creates a type library that describes the

function entry points in the DLL. Refer to the *OLE 2 Programmers Reference, Volume 2*, from Microsoft Press for information on using type libraries.

- **Generate DEF File** generates a `.def` file for the instrument driver. External compilers use the `.def` file to compile your instrument driver into a DLL. The file contains entries to export each function in the function tree.

- The **Create DLL Project** command creates a LabWindows/CVI project (`.prj`) file that can be used to create a dynamic link library (`.dll`) from the program file associated with the function panel (`.fp`) file. When you execute this command, you are prompted to enter a pathname for the project file. After the file is written, you are asked if you want to load the project immediately. If you do, your current project is unloaded. Refer to the *Preparing Source Code for Use in a DLL* section in Chapter 3, *Compiler/Linker Issues*, in the *LabWindows/CVI Programmer Reference Manual* or more information on creating DLLs.

- **VXIplug&play Style** affects the contents of the DLL project that you create using the **Create DLL Project** command. If the **VXIplug&play Style** command is enabled, **Create DLL Project** adds project settings that allow the DLL, import libraries, and distribution kit you create to conform to various aspects of the VXI*plug&play* specification. You can modify all these settings using commands in the **Build** menu of the Project window. The following list describes the default settings.

  – The **Instrument Driver Support Only** command is enabled.

  – In the Create Dynamic Link Library dialog box:

    - "`_32`" is appended to the base filename of the DLL but not to the base filename of the import libraries.

    - In the Import Library Choices dialog box, the **Generate import library for all compilers** option is enabled.

    - In the Type Library dialog box:

      – The **Add type library resource to DLL** option is enabled.

      – The **Include links to help file** option is enabled.

      – **Function panel file** is set to the full pathname of the `.fp` file of the current Function Tree Editor window.

    - In the Change dialog box in the **Exports** section:

      – The **Export What** option is set to `Include File Symbols`

      – The **Which Project Include Files** list contains the name of the include file associated with the `.fp` file of the current Function Tree Editor window.

– In the Create Distribution Kit dialog box:

- The **Install Run-Time Engine** option is disabled. The instrument driver support DLL is included in the file groups instead. If you need the LabWindows/CVI Run-time Engine for the soft front panel executable, you must enable this option manually.

- File groups are created that contain all the files that are required of a VXI*plug&play* instrument driver installation. For example, only the import libraries for Visual C/C++ and Borland C/C++ are included, and their directory names are msc and bc. Files that you must create independently are also named in the file groups, even if they do not currently exist. These files include the following:

  – A Visual Basic include file, which you can create by selecting **Options»Generate Visual Basic Include** in the Source window

  – A documentation file, which you can create by selecting **Options»Generate Documentation**

  – A help file, which you can create by selecting **Options»Generate Windows Help** and the Windows Help Compiler

  – A knowledge base file as defined in the VXI*plug&play* specification

  – Files for a soft front panel executable (an empty file group is created for this)

– In the Advanced dialog box:

- The **Use Custom Script** option is enabled.

- **Script Filename** is set to cvi\bin\vxipnp.inf.

- **Executable Filename** is left empty. After you create a soft front panel executable and add it to the soft front panel file group, click on the **Select** button to specify the soft front panel executable as the **Executable Filename**.

- The **Installation Title** names are set to <*instrument prefix*> Instrument Driver.

Function Tree Editor Examples

These examples teach you about creating and editing function trees, specifically creating a function tree with multiple classes, cutting and pasting functions and classes in a function tree, and cutting and pasting functions and classes between the function trees of different drivers.

Example—Multiple Classes in a Function Tree

In this example you create a function tree with several nested classes. Before beginning, invoke the Function Tree Editor by selecting **File»New» Function Tree (*.fp)**.

Complete the following steps to create a new instrument and function tree as follows:

1. Select **Create»Instrument**.

2. Enter the name Function Tree Examples as the Name and tree as the Prefix. Click on **OK**.

3. Select **Create»Function Panel Window**.

4. Enter the name Function 1 as the Name and fun1 as the Function Name. Click on **OK**.

5. Select **Create»Class**.

6. Enter the name Class 1 as the Name. Click on **OK**.

7. Select the line beneath the name Class 1.

8. Select **Create»Function Panel Window**.

9. Enter the name Function 2 as the Name and fun2 as the Function Name. Click on **OK**.

10. Select **File»Save .FP File As** and save the file as mltcls.

The new function tree is shown in Figure 5-4.
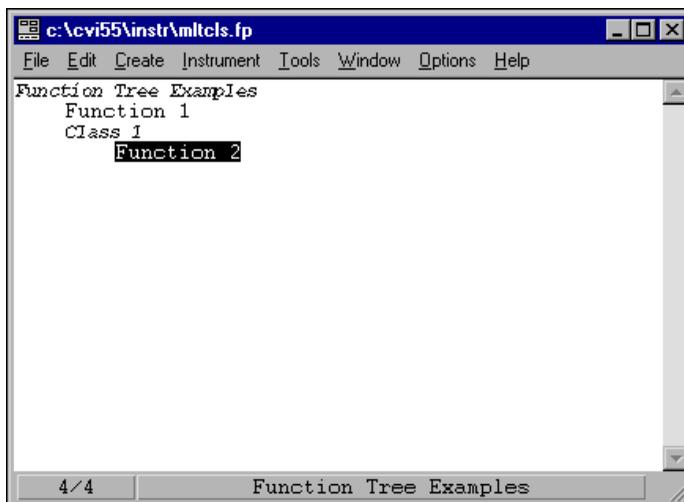
Figure 5-4. Sample Function Tree

To view the structure of the function tree as it is seen by the user of the driver, select the instrument name from the **Instrument** menu.

Example—Cutting and Pasting Functions and Panels

Frequently, you want to copy a function in a function tree and its associated function panel to a new position within the function tree.

Complete the following examples to cut and paste a function within a function tree.

1. Position the selection on the name Function 1.

2. Select **Edit»Cut**. The function disappears from the tree and is stored on the clipboard.

3. Position the selection on the name Function 2.

4. Select **Edit»Paste Above**. The function now appears under Class 1.

Suppose that instead of moving the function, you want to replicate it. Because the function is still in the Function Tree Editor clipboard, you can move the selection to the name Class 1 and select **Edit»Paste Above**. The name Function 1 reappears at the top of the tree.

Note   Pasting functions and classes within the Function Tree Editor copies all items associated with the function or class, including controls and function panel help.

Using Existing Function Panels in a New Driver

Complete the following steps to copy some of the function panels from this driver to a new driver.

1. Select **File»New»Function Tree (*.fp)**. A new blank function tree window appears on the screen.

2. Select **Create»Instrument**.

3. Name the instrument New Instrument and type new in the prefix box. Click on **OK**.

4. Select **Window»Function Tree** and select the file called mltcls.

5. Position the selection on the item Class 1.

6. Select **Edit»Copy**.

7. Return to the New Instrument file through the **Window** menu.

8. Position the selection on the line beneath the name of the instrument.

9. Select **Edit»Paste Below**. Class 1 and its associated functions appear in the new tree.

When you paste a class into a new tree, all information associated with the class and the functions of the class are retained.

Example—Editing Items in the Function Tree

In this example you edit the names displayed in the function tree. You edit all the function tree items by selecting **Edit»Edit Node**.

Complete the following steps to change the name of the instrument driver and its prefix:

1. Select New Instrument.

2. Select **Edit»Edit Node**. The Edit Instrument Node dialog box originally used to create the instrument appears.

3. Change the name of the instrument to Tree #2 and the prefix to tree2. Click on **OK**.

The changes in the instrument driver name appears at the top of the function tree in the Function Tree Editor as well as at the bottom of the window. The changes to the prefix are reflected in the Generated Code Window in each function panel.

6

Function Panel Editor

This chapter describes how to create and modify instrument driver function panels using the Function Panel Editor.

Invoking the Function Panel Editor

You can invoke the Function Panel Editor from the Function Tree Editor or from a function panel.

Invoking from the Function Tree Editor

To invoke the Function Panel Editor from the Function Tree Editor:

1. Highlight the function that corresponds to the function panel you want to edit.

2. Select **Edit»Edit Function Panel Window** on the Function Tree Editor menu bar.

You also can invoke the Function Panel Editor with the shortcut key <F8> or by double-clicking on the function name.

Invoking from a Function Panel

To edit a function panel that you are currently operating, select **Option»Edit Function Panel Window** on the Function Panel menu bar. If the current function panel is for a library that is in the **Library** menu, you cannot use the **Edit Panel** command.

Function Panel Editor Menu Bar

When you invoke the Function Panel Editor to create a new function panel, a screen similar to the one shown in Figure 6-1 appears.
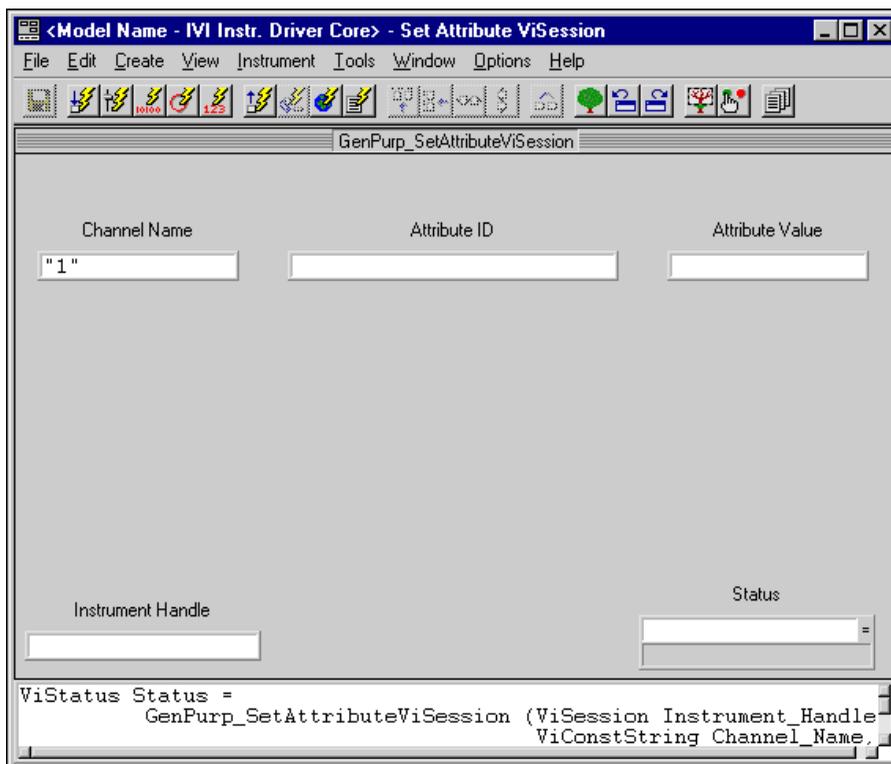
Figure 6-1. Function Panel Editor

The following items appear on the function panel:

•   The Function Panel Editor menu bar appears at the top of the screen above the function panel.

•   The Instrument Name and Function Panel Name appear in the title bar of the function panel window.

•   The Function Name appears in the title bar of the function panel.

•   The Function Name appears with an empty argument list in the Generated Code window, below the Function Panel Editor window.

You have the following options in the Function Panel Editor menu bar.

- **File** lets you create a new function tree, edit an existing function tree, save function panel information into a `.fp` and `.sub` file on disk, or add function panels to a project.

- **Edit** lets you modify controls, panels, and functions, add context-sensitive help information, or align and distribute objects.

- **Create** lets you add controls, function panels, or a common control panel to the function panel window.

- **View** lets you select another panel in the current instrument or from the panel list.

- **Instrument** lets you select a panel that you want to edit from a different instrument driver.

- **Tools** lets you generate function definitions and declarations into your source and include files, jump to function definitions and declarations, and invoke the instrument driver developer wizard and attribute editor.

- **Window** lets you select which window to make active.

- **Options** lets you invoke the Function Tree Editor, operate the function panel, or toggle the scroll bars.

File

The **File** menu lets you create a new function tree, edit an existing function tree, save function panel information into a `.fp` and `.sub` file on disk, or add function panels to a project. The **File** menu operates like the **File** menu of the Project window. Refer to Chapter 3, *Project Window*, of the *LabWindows/CVI User Manual* for more information about the **File** menu.

For each IVI instrument driver, a `.sub` file accompanies the `.fp` file. The `.sub` file contains the information about the instrument driver attributes. You edit this information using the attribute editor. When you save the contents of a `.fp` file, LabWindows/CVI also saves the contents of the `.sub` file automatically.

Edit

The **Edit** menu lets you edit the objects on a function panel window. You have the following options in the **Edit** menu.

- **Cut Controls** deletes the selected controls and copies them to the clipboard.

- **Copy Controls** copies the selected controls to the clipboard.

- **Paste** inserts the contents of the clipboard into the selected function panel.

- **Cut Panel** deletes the selected panel from the function panel window and copies it to the clipboard.

- **Copy Panel** copies the selected panel to the clipboard.

- **Edit Control** lets you edit attributes of a control.
- **Change Control Type** lets you change the type of an existing control.
- **Edit Function** lets you edit a function.
- **Alignment** lets you align controls on a function panel.
- **Align Horizontal Centers** repeats your previous alignment operation.
- **Distribution** lets you distribute controls on a function panel.
- **Distribute Vertical Centers** repeats your previous distribution operation.
- **Find** allows you to locate a particular text string in the function panel file.
- **Replace** allows you to replace a particular text string in the function panel file with another text string.
- **Control Help** lets you create or modify help information for a specific control.
- **Function Help** or **Window Help** lets you create or modify help information for the function.

Cut Controls

The **Cut Controls** command removes the selected controls from the function panel and places the controls and their associated help information on the clipboard. The contents of the clipboard stay in place when you change panels.

Copy Controls

The **Copy Controls** command copies the selected controls and their associated help information to the clipboard. The contents of the clipboard stay in place when you change panels.

Paste

The **Paste** command copies objects from the clipboard and places them on a function panel window. You can paste the same object as many times as you need to.

You cannot paste a return value control on a function panel that already contains one. A function panel can contain only one return value control.

Cut Panel

The **Cut Panel** command removes the selected panel from the function panel window and places the panel, its controls, and all the associated help information on the clipboard. The contents of the clipboard stay in place when you change function panel windows.

Copy Panel

The **Copy Panel** command copies the selected panel, its controls, and all the associated help information to the clipboard. The contents of the clipboard stay in place when you change function panel windows.

Edit Control

You can modify an existing control with **Edit Control**. When you select **Edit Control**, you see the same series of dialog boxes you use to create the control. The *Create* section later in this chapter discusses the proper use of these dialog boxes.

Change Control Type

You can change the type of a control with **Change Control Type**. When you select **Change Control Type**, a dialog box appears that lists the available control types.

Select the desired control type from the dialog box. When you select a new control type, you see the same series of dialog boxes that you use to create the control. The *Create* section later in this chapter gives more information about using these dialog boxes.

If you change a control type from slide to ring, or vice versa, the new control type retains the option list associated with the old control.

Edit Function

You can modify an existing function panel with **Edit Function**. When you select **Edit Function**, you see the same series of dialog boxes you use to create the panel. The *Create* section later in this chapter discusses the proper use of these dialog boxes.

Alignment

Alignment lets you align the selected controls. The **Alignment** command operates like the **Alignment** command in the User Interface Editor. Refer to Chapter 4, *User Interface Editor Window*, of the *LabWindows/CVI User Manual* for more information about the **Alignment** command.

Align Horizontal Centers

Align Horizontal Centers repeats your previous alignment operation. The **Align Horizontal Centers** command operates like the **Align Horizontal Centers** command in the User Interface Editor. Refer to Chapter 4, *User Interface Editor Window*, of the *LabWindows/CVI User Manual* for more information about the **Align Horizontal Centers** command.

Distribution

Distribution lets you distribute the selected controls. The **Distribution** command operates identically to the **Distribution** command in the User Interface Editor. Refer to Chapter 4, *User Interface Editor Window*, of the *LabWindows/CVI User Manual* for information about the **Distribution** command.

Distribute Vertical Centers

Distribute Vertical Centers repeats the previous distribution. The **Distribute Vertical Centers** command operates like the **Distribute Vertical Centers** command in the User Interface Editor. Refer to Chapter 4, *User Interface Editor Window*, of the *LabWindows/CVI User Manual* for more information about the **Distribute Vertical Centers** command.

Find

The **Find** command allows you to locate a particular text string in the function panel file. You can search for text in node names; function names; control labels; control values; item labels in ring, slide, and binary controls; message control text; and help text. When you search in help text, you cannot search in any of the other items at the same time. The search begins at the function tree node for the current Function Panel Editor window. The **Find** command always searches all controls on the panel regardless of whether any are currently selected.

 If the **Find** command brings up a Help Editor window and you do not use the button bar, you must return to the Function Panel Editor window or Function Tree Editor window to continue searching throughout the function panel file. The **Find** command in the Help Editor window searches only within the window. You can return to the Function Panel Editor window by pressing <F8>. You can return to the Function Tree Editor window by pressing <F7>.

Replace

The **Replace** command operates the same as the **Find** command except that you can replace the search string with another search string.

Control Help

You can add or modify context-sensitive help information for a particular control with **Control Help**. Refer to Chapter 7, *Adding Help Information*, for more information about adding help to a function panel.

Function Help or Window Help

You can add or modify context-sensitive help information for the entire function panel
with **Function Help** or **Window Help**. **Function Help** corresponds to new style help and
Window Help corresponds to old style help. Refer to Chapter 5, *Function Tree Editor*, for
more information on how to set the help style of the instrument driver. Refer to Chapter 7,
Adding Help Information, for more information about adding help to a function panel.

Create

The **Create** menu lets you add controls to a function panel. There are nine control types in
the **Create** menu: input, slide, binary, ring, numeric, output, return value, global variable, and
message.

Function Panel Window, Function Panel,
and Common Control Panel

The function panel window is a collection of panels that represent all functions that users can
interactively call from that window. Two types of panels are associated with a function panel
window: function panels and common control panels. You can create controls on either type
of panel.

A function panel graphically represents a single function. Function panels can contain any of
the nine different control types. A function panel can have only one return value control. The
function panel window can contain more than one function panel.

A common control panel contains controls that are common to all functions represented by
function panels in the function panel window. Common control panels are useful only when
you have multiple function panels in the function panel window. Controls on the common
control panel appear as the first parameter of every function associated with a function panel
window. A function panel window can contain only one common control panel. You could
use a common control with an instrument driver that allows multiple instruments of the same
model type to exist on a GPIB board. In this case, the common control panel can contain a
control that is an index to specify which instrument is addressed.

Note   In general, National Instruments recommends that you have only one function panel
per window and no common control panels.

Input

An input control accepts a variable name or value entered from the keyboard. When you select
Create»Input, the dialog box shown in Figure 6-2 appears.
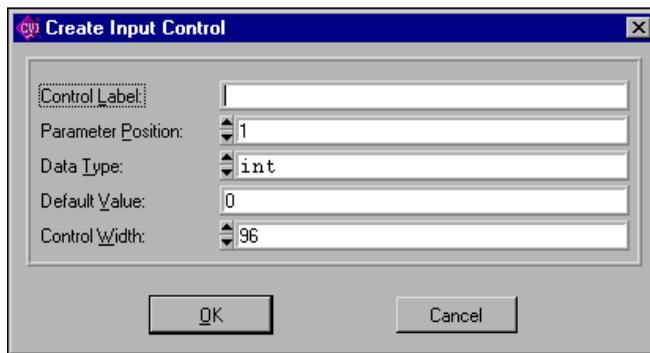
Figure 6-2. Create Input Control Dialog Box

You see the following items in the dialog box:

• **Control Label** specifies the label that appears above the control on the panel.

• **Parameter Position** lets you select the location of the control value in the function
  parameter list. For a control in a common control panel, Parameter Position specifies the
  control value in the parameter lists of all function panels in a function panel window.
  The first position is one (1).

  For a control on a function panel, Parameter Position specifies the control value in the
  parameter list after the controls in the common control panel. The first position after
  the controls in the common control panel is one (1). If there is no common control panel,
  the first position is one (1).

• **Data Type** lets you select the data type of the item entered in the input control. The data
  type can be one of any of the data types listed in the *Data Types* section in Chapter 3,
  Developing an Instrument Driver.

• **Default Value** specifies the default for the input control, which must be a valid value,
  a constant name, or any other valid C expression.

• **Control Width** lets you specify the width of the control in pixels. The minimum allowed
  is 24. The maximum allowed is 2,048.

Slide

A slide control looks like a mechanical slide switch. A slide control specifies a parameter value depending upon the position of the cross-bar of the slide control. When you select **Create»Slide**, the dialog box shown in Figure 6-3 appears.
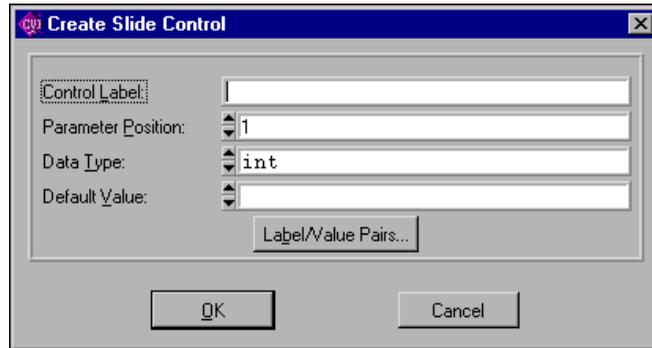
Figure 6-3. Create Slide Control Dialog Box

You see the following items in the dialog box:

- **Control Label** specifies the label that appears above the control on the function panel.

- **Parameter Position** lets you select the location of the control value in the function parameter list. For a control in a common control panel, Parameter Position specifies the control value in the parameter lists of all function panels in a function panel window. The first position is one (1).

  For a control on a function panel, Parameter Position specifies the control value in the parameter list after the controls in the common control panel. The first position after the controls in the common control panel is one (1). If there is no common control panel, the first position is one (1).

- **Data Type** lets you select the data type of the values in the slide control. The data type can be one of any of the data types listed in the *Data Types* section in Chapter 3, *Developing an Instrument Driver*.

- **Default Value** lets you select the default for the slide control, which must be one of the labels specified in the Edit Label/Value Pairs dialog box.

• When you click on the **Label/Value Pairs** button, the Edit Label/Value Pairs dialog box shown in Figure 6-4 appears.
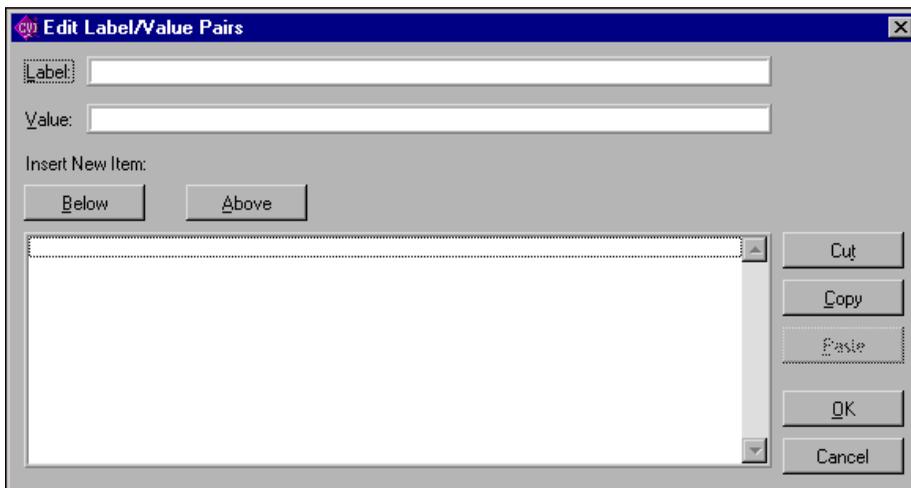
Figure 6-4. Edit Label/Value Pairs Dialog Box

Use this dialog box to specify the label and value associated with each cross-bar position on the slide control. A slide control can have up to 32 labels and associated values.

You see the following items in this dialog box:

– **Label** specifies a label that appears on the slide control.

– **Value** specifies the value, constant name, or expression associated with the label entered in the Label text box.

– The list box below the **Label** and **Value** text boxes displays the labels and the values of items that appear on the slide control.

Adding a Label and Value to the Slide Control List

Complete the following steps to add a label to the slide control list.

1. Type the label in the **Label** text box and press <Enter>. The highlight moves to the **Value** text box.

2. Type the value in the **Value** text box. You can use a constant name or any other valid C expression.

3. Press <Enter> to add the label and value to the slide control list.

LabWindows/CVI adds the label and value after the label and value line that is currently selected in the list box.

Dialog Box Command Buttons

You perform all operations on the items in the list box by entering information into the **Label** and **Value** text boxes and selecting one of the command buttons above or to the right of the list box in the dialog box. You can select the following command buttons:

*   **Below** inserts a blank line below the selected line in the list box.

*   **Above** inserts a blank line above the selected line in the list box.

*   **Cut** removes the selected line from the list and places it in the clipboard.

*   **Copy** copies the selected line to the clipboard.

*   **Paste** inserts the label and value line contained in the clipboard below the selected line in the list box.

*   **OK** accepts the entries in the list box, then removes the dialog box.

*   **Cancel** cancels changes, removes the current dialog box from the screen, and returns you to the Create Slide Control dialog box.

Binary

A binary control operates like a mechanical on/off switch. A binary control gives a parameter value one of two predefined values, depending on whether the control is in the up or down position. When you select **Create»Binary**, the dialog box shown in Figure 6-5 appears.
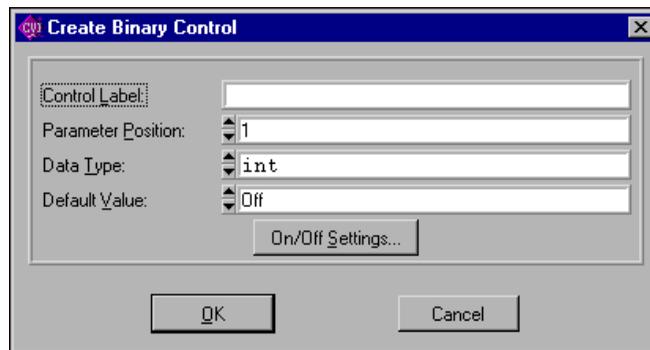
Figure 6-5. Create Binary Control Dialog Box

You see the following items in the Create Binary Control dialog box:

*   **Control Label** specifies the label that appears above the control on the panel.

*   **Parameter Position** lets you select the location of the control value in the function parameter list. For a control in a common control panel, **Parameter Position** specifies the control value in the parameter lists of all function panels in a function panel window. The first position is one (1).

For a control on a function panel, **Parameter Position** specifies the control value in the parameter list after the controls in the common control panel. The first position after the controls in the common control panel is one (1). If there is no common control panel, the first position is one (1).

• **Data Type** lets you select the data type of the values in the binary control. The data type can be one of any of the data types listed in the *Data Types* section in Chapter 3, *Developing an Instrument Driver*.

• **Default Value** lets you select the default for the binary control, which must be either the On or Off label.

• When you select the **On/Off Settings** button, the Edit On/Off Settings dialog box shown in Figure 6-6 appears.
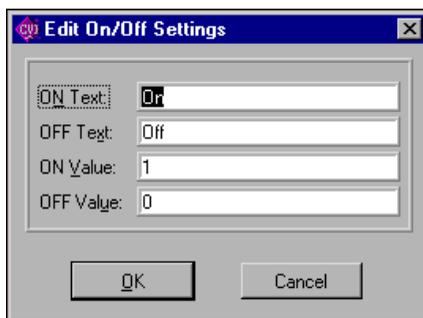
Figure 6-6. Edit On/Off Settings Dialog Box

–   **ON Text** specifies the label that appears next to the upper (on) position of the binary control.

–   **OFF Text** specifies the label that appears next to the lower (off) position of the binary control.

–   **ON Value** specifies the value, constant name, or valid C expression you want to associate with the On label.

–   **OFF Value** specifies the value, constant name, or valid C expression you want to associate with the Off label.

Ring

A ring control shows the user an option list. A ring control displays only one item at a time from its list of options. When you select **Ring** from the **Create** menu, the dialog box shown in Figure 6-7 appears.
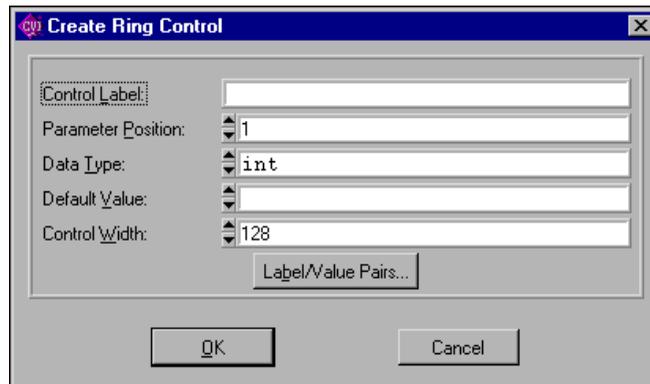
Figure 6-7. Create Ring Control Dialog Box

You see the following items in the Create Ring Control dialog box:

• **Control Label** specifies the label that appears above the control on the function panel.

• **Parameter Position** lets you select the location of the control value in the function parameter list. For a control in a common control panel, **Parameter Position** specifies the control value in the parameter lists of all function panels in a function panel window. The first position is one (1).

 For a control on a function panel, **Parameter Position** specifies the control value in the parameter list after the controls in the common control panel. The first position after the controls in the common control panel is one (1). If there is no common control panel, the first position is one (1).

• **Data Type** lets you select the data type of the values in the ring control. The data type can be one of any of the data types listed in the *Data Types* section in Chapter 3, *Developing an Instrument Driver*.

• **Default Value** lets you select the default for the ring control, which must be one of the labels specified in the Edit Label/Value Pairs dialog box.

• **Control Width** lets you specify the width of the control in pixels. The minimum allowed is 24. The maximum allowed is 2,048.

- When you click on the **Label/Value Pairs** button, the Edit Label/Value Pairs dialog box shown in Figure 6-8 appears.
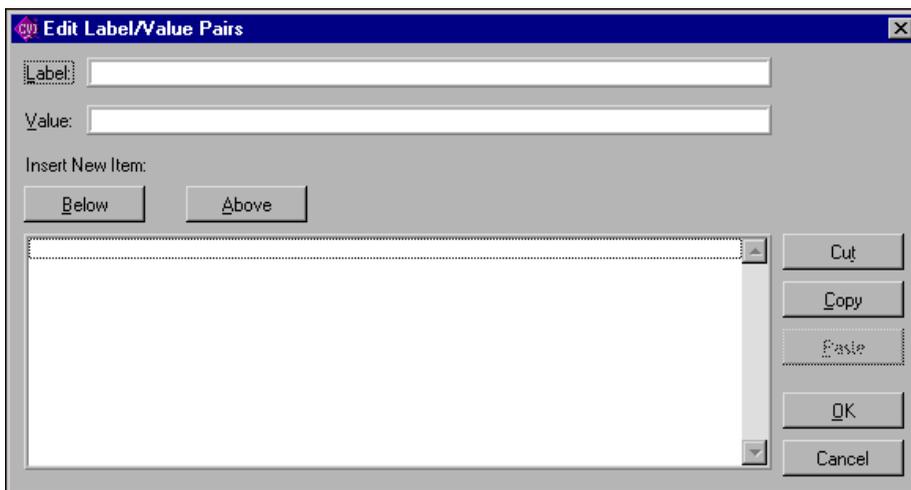
Figure 6-8. Ring Control Edit Label/Value Pairs Dialog Box

Use this dialog box to specify the label and value associated with each entry in the ring control. A ring control can have up to 32,000 labels and associated values.

You see the following items in this dialog box:

– **Label** specifies a label that appears on the ring control.

– **Value** specifies the value, constant name, or expression associated with the label entered in the **Label** text box.

– The list box below the **Label** and **Value** text boxes displays the labels and the values of items that appear on the ring control.

Adding a Label and Value to the Ring Control List

Complete the following steps to add a label to the ring control list:

1. Type the label in the **Label** text box and press <Enter>. The highlight moves to the **Value** text box.

2. Type the value in the **Value** text box. You can use a constant name or any other valid C expression.

3. Press <Enter> to add the label and value to the ring control list.

LabWindows/CVI adds the label and value after the label and value line that is currently selected in the list box.

Dialog Box Command Buttons

You perform all operations on the items in the list box by entering information into the **Label** and **Value** text boxes and selecting one of the command buttons above or to the right side of the list box. You can select the following command buttons:

*   **Below** inserts a blank line below the highlighted line in the list box.

*   **Above** inserts a blank line above the highlighted line in the list box.

*   **Cut** removes the highlighted line from the list and places it in the clipboard.

*   **Copy** copies the highlighted line to the clipboard.

*   **Paste** inserts the label and value line contained in the clipboard below the highlighted line in the list box.

*   **OK** accepts the entries in the list box, then removes the dialog box.

*   **Cancel** cancels changes, removes the current dialog box from the screen, and returns you to the Create Ring Control dialog box.

Numeric

A numeric control is an input control that lets you increment a control using the up and down arrows. When you select **Create»Numeric**, the dialog box shown in Figure 6-9 appears.
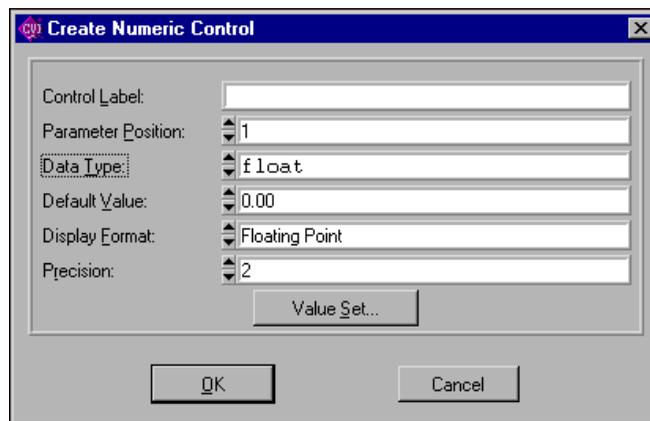
Figure 6-9. Create Numeric Control Dialog Box

You see the following items in the Create Numeric Control dialog box:

*   **Control Label** specifies the label that appears above the control on the function panel.

*   **Parameter Position** lets you select the location of the control value in the function parameter list. For a control in a common control panel, **Parameter Position** specifies

the control value in the parameter lists of all function panels in a function panel window. The first position is one (1).

For a control on a function panel, **Parameter Position** specifies the control value in the parameter list after the controls in the common control panel. The first position after the controls in the common control panel is one (1). If there is no common control panel, the first position is one (1).

• **Data Type** lets you select the data type of the values in the numeric control. You can choose from the following data types:

```
int
short
char
unsigned int
unsigned short
unsigned char
double
float
```

or choose a user-defined data type for which you have specified an intrinsic type.

• **Default Value** lets you select the default for the numeric control, which must be a valid member of the value set.

• **Display Format** lets you select the output format. You can display integers, longs, and shorts in decimal, hexadecimal, octal or ASCII. You can display doubles and floats in either scientific or floating-point notation.

• **Precision** let you select how many digits that the control displays to the right of the decimal point.

• When you click on the **Value Set** button, the Edit Value Set dialog box shown in Figure 6-10 appears.
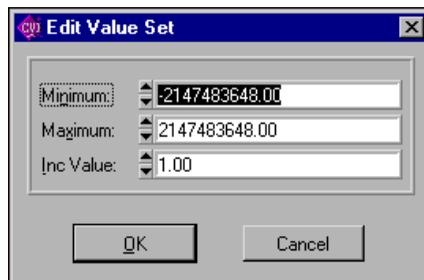
Figure 6-10. Edit Value Set Dialog Box

You see the following items in the Edit Value Set dialog box:

– **Minimum** lets you select the minimum value the numeric control accepts.

– **Maximum** lets you select the maximum value the numeric control accepts.

– **Inc Value** lets you select the amount the numeric control value increments or decrements when the user presses the up or down arrows. The value in **Inc Value** must divide evenly into the range of the numeric control.

Output

An output control displays the results of a function call. When you select **Create»Output**, the dialog box shown in Figure 6-11 appears.
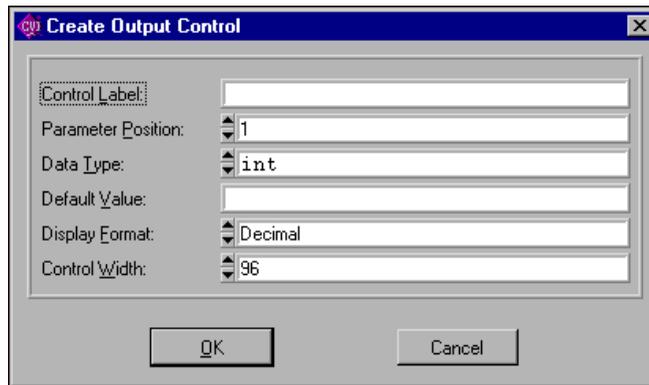
Figure 6-11. Create Output Control Dialog Box

You see the following items in the Create Output Control dialog box:

• **Control Label** specifies the label that appears above the control on the panel.

• **Parameter Position** lets you select the location of the control value in the function parameter list. For a control in a common control panel, **Parameter Position** specifies the control value in the parameter lists of all function panels in a function panel window. The first position is one (1).

For a control on a function panel, **Parameter Position** specifies the control value in the parameter list after the controls in the common control panel. The first position after the controls in the common control panel is one (1). If there is no common control panel, the first position is one (1).

• **Data Type** lets you select the data type of the variable or value displayed in the output control. The data type can be one of any of the data types listed in the *Data Types* section in Chapter 3, *Developing an Instrument Driver*.

- **Default Value** lets you specify the default value that the control displays. you can leave this item blank.

- **Display Format** lets you select the format in which the output control displays values. You can display integers, longs, shorts, and chars in decimal, hexadecimal, octal or ASCII. You can display doubles and floats in either scientific or floating-point notation. If the data type is char *, void *, a meta data type, or an array, the display format control is not valid.

- **Control Width** lets you specify the width of the control in pixels. The minimum allowed is 24. The maximum allowed is 2,084.

Return Value

A return value control displays a value returned from a function. You can use a return value control only if the function has a non-void data type. When you select **Create»Return Value**, the dialog box shown in Figure 6-12 appears.
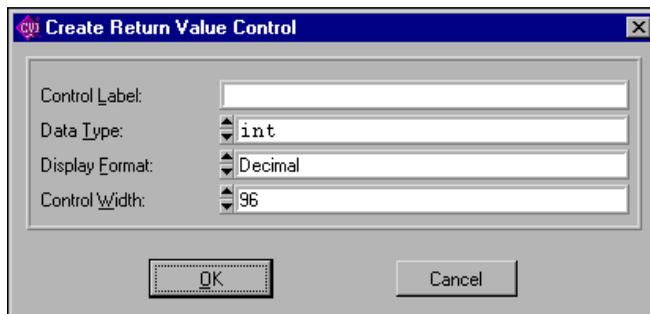
Figure 6-12. Create Return Value Control Dialog Box

You see the following items in the Create Return Value Control dialog box:

- **Control Label** specifies the label that appears above the control on the function panel.

- **Data Type** lets you select the data type of the variable or value displayed in the return value control. The data type can be any data type other than an array type or a meta data type.

- **Display Format** lets you select the format in which the return value control displays values. You can display integers, longs, shorts, and chars in decimal, hexadecimal, octal or ASCII. You can display doubles and floats in either scientific or floating-point notation. If the data type is char * or void *, the display format control is not valid.

- **Control Width** lets you specify the width of the control in pixels. The minimum allowed is 24. The maximum allowed is 2,048.

Global Variable

A global variable control displays the value of a global variable defined in LabWindows/CVI when users operate the function panel. When you select **Create»Global Variable**, the dialog box shown in Figure 6-13 appears.
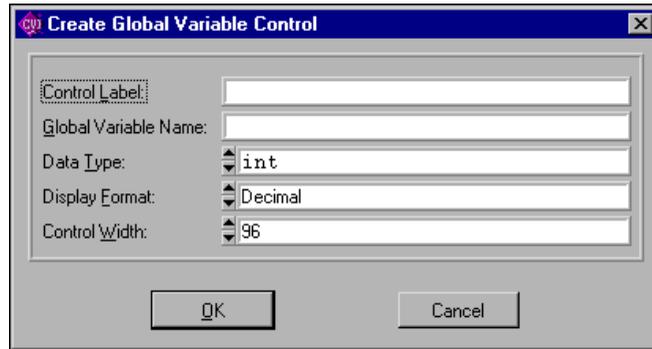
Figure 6-13. Create Global Variable Control Dialog Box

You see the following items in the Create Global Variable Control dialog box:

*   **Control Label** specifies the label that appears above the control on the panel.

*   **Global Variable Name** specifies the name of the variable whose contents are shown in the global control.

*   **Data Type** lets you select the data type of the item entered in the input control. The data type can be one of any of the data types listed in the *Data Types* section in Chapter 3, *Developing an Instrument Driver*.

*   **Display Format** lets you select the format in which the global variable control displays values. You can display integers, longs, shorts, and chars in decimal, hexadecimal, octal or ASCII. You can display doubles and floats in either scientific or floating-point notation. If the data type is char *, void *, a meta data type, or an array, the display format control is not valid.

*   **Control Width** lets you specify the width of the control in pixels. The minimum allowed is 24. The maximum allowed is 2,048.

Message

You can place text anywhere on the panel with a message control. This serves as an online documentation tool for panels. When you select **Create»Message**, a dialog box appears. Enter the desired text into the message text control and click on the **OK** button. To enter a new line in the message text control, press <Ctrl-Enter>. The text appears on the panel, and you can position it like any other control.

View

Use the **View** menu commands to view the current instrument driver function panels or the most recently used function panels. The commands give easy access to function panels within an instrument driver. Refer to Chapter 5, *Using Function Panels*, in the *LabWindows/CVI User Manual*, for more information on the **View** menu.

Instrument

Use the **Instrument** menu to load and edit instrument drivers and to specify which instrument driver function panel to edit. The **Instrument** menu operates identically to the **Instrument** menu on the Function Tree Editor menu bar. Refer to Chapter 5, *Function Tree Editor*, for more information about the **Instrument** menu.

Tools

The Tools menu presents the following options:

- **Create ActiveX Automation Controller** generates a new instrument driver for an ActiveX Automation Server. When you select the **Create ActiveX Automation Controller** command, LabWindows/CVI displays the ActiveX Automation Controller Wizard.

- **Create IVI Instrument Driver** initiates the instrument driver developer wizard. Refer to Chapter 3, *Developing an Instrument Driver*, for more information on the instrument driver developer wizard.

- **Edit Instrument Attributes** initiates the attribute editor. Refer to Chapter 4, *Attribute Editor*, for more information on the attribute editor.

- **Enable Auto Replace** enables the updating of instrument driver source files to reflect changes to function names in the function tree. This option is global to LabWindows/CVI, so enabling it in one Function Tree Editor window enables it for all Function Tree Editor windows. This command is dimmed for function trees that have not yet been saved.

  When this option is enabled and not dimmed, LabWindows/CVI updates the instrument driver .c, .h, and .sub files to reflect changes you make to function names or the instrument prefix in the Function Tree Editor window or Function Panel Editor window. When you change a function name, LabWindows/CVI prompts you for permission to update your instrument driver to reflect the new name. When you change the instrument prefix, LabWindows/CVI prompts you for permission to update your instrument driver to reflect the new prefix.

- **Generate Source For Function Panel** generates function definitions and declarations in your driver source and header files. If a function definition already exists, LabWindows/CVI prompts you for permission to update it. You can replace, insert above or below, or skip without updating. Your choice also applies to the declaration.

- **Go To Definition** opens the driver source file and jumps to the definition of the function that is currently selected in the function tree.

- **Go To Declaration** opens the driver include file and jumps to the declaration of the function that is currently selected in the function tree.

Window

The **Window** menu lets you select which window to make active. The **Window** menu operates like the **Window** menu of the Project window. Refer to Chapter 3, *Project Window*, in the *LabWindows/CVI User Manual*, for more information about the **Window** menu.

Options

The **Options** menu lets you invoke the Function Tree Editor or operate the current function panel. The **Options** menu includes the following items:

- Data Types
- Toolbar
- Default Panel Size
- Panels Movable
- Toggle Scroll Bars
- Edit Function Tree
- Operate Function Panel

Data Types

The **Data Types** command lets you specify the names of user-defined data types. Data types you specify with the **Data Types** command appear in the Data Type Ring control on the Edit Control dialog boxes for input, slide, binary, ring, output, and global variable controls.

✎ **Note**   The .h file for the instrument driver must define the types that you specify with the **Data Types** command.

When you select **Options»Data Types,** the dialog box in Figure 6-14 appears.
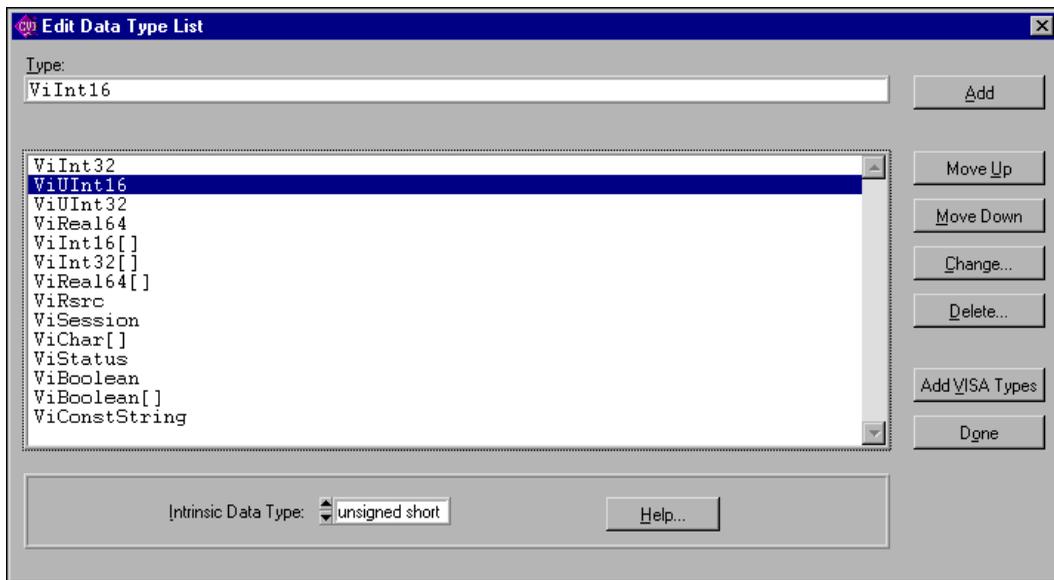
Figure 6-14. Edit Data Type List Dialog Box

The items in the Edit Data Type List dialog box are as follows:

•   **Type** specifies the name of a user-defined data type.

•   **Intrinsic Data Type** allows you to associate each user defined data type with one of the intrinsic C data types that you can use in a numeric control. If you select an item other than None, you can use the user-defined data type as the data type for a numeric control.

•   **Add** places the name in the **Type** control in the **Data Type** list.

•   **Move Up** moves the selected entry up one line in the **Data Type** list.

•   **Move Down** moves the selected entry down one line in the **Data Type** list.

•   **Change** displays a dialog box that prompts you to change the selected entry in the **Data Type** list.

•   **Delete** removes an entry in the **Data Type** list.

•   **Add VISA Types** adds the special set of data types defined by the VISA I/O library.

•   **Done** accepts edits to the **Data Type** list and returns to the Function Panel Editor.

Toolbar

The **Toolbar** command displays a dialog box that prompts you to select which icons appear in the Function Panel Editor tool bar.

Default Panel Size

The **Default Panel Size** command sizes and positions the function panel so that it exactly fills up the default function panel window size.

Panels Movable

The **Panels Movable** command lets you specify whether panels are movable within a Function Panel Editor window. Panels are never movable in operate mode.

Toggle Scroll Bars

The **Toggle Scroll Bars** command adds or removes horizontal and vertical scroll bars from a function panel.

Edit Function Tree

The **Edit Function Tree** command invokes the Function Tree Editor.

Operate Function Panel

The **Operate Function Panel** command lets you operate the current function panel window.

Moving Controls

When you create a control, the new control always appears in the same location on the function panel. You can position a control anywhere on a function panel.

Complete the following steps to move a control using the keyboard.

1. Press <Page Up> and <Page Down> to move the highlight to the function panel that contains the control.

2. Press the <Tab> key to move the highlight to the control.

3. Press the arrow keys to move the control up, down, left, or right to the location you want. Press <Ctrl> and the arrow keys to position the control precisely.

To move a control using the mouse, click the mouse button on the control you want to move and drag the control to the desired location.

Moving Controls between Function Panels

You can move a control from one function panel page to another using the clipboard.

Complete the following steps to move a control from one page to another.

1. Select the desired control.

2. Select **Edit»Cut Controls**.

3. Move to the new function panel.

4. Select **Edit»Paste**.

Selecting Multiple Controls

To select multiple controls, click and drag the mouse selector box around the controls you want to select.

Function Panel Editor Examples

The following examples teach you about creating and editing function panel windows, specifically creating a function panel window with one function panel, creating controls on a function panel, changing the type of control, and cutting and pasting controls on a panel and between panels.

Example—Creating a Function Window

In this example, you create a function panel without writing any code. The example panel controls an oscilloscope with two channels and configures the vertical sensitivity, coupling, and invert setting of the oscilloscope.

Complete the following steps to create a new instrument and panel.

1. Select **File»New»Function Tree (*.fp)**.

2. Select **Create»Instrument**.

3. Enter `Function Panel Examples` as the **Name** and `panel` as the **Prefix**. Click on **OK**.

4. Select **Create»Function Panel Window**.

5. Enter `Configure` as the **Name** and `config` as the **Function Name**. Click on **OK**.

6. Select the `Configure` node in the function tree and select **Edit»Edit Function Panel**. A new function panel window that contains a single function panel appears on the screen. Notice that the code name of the function appears in the Generated Code window, preceded by the prefix.

7. Select **Create»Binary**.

8. Complete the Create Binary Control dialog box as shown in Figure 6-15.
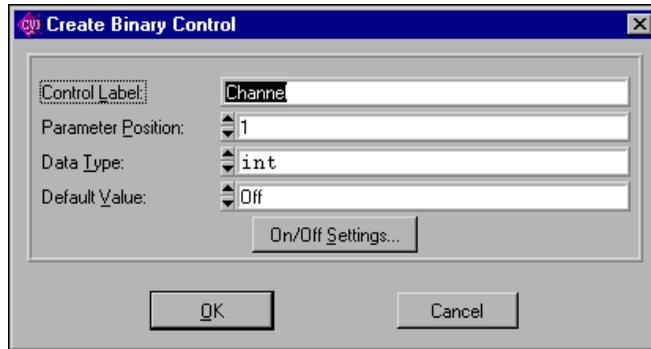
Figure 6-15. Channel Create Binary Control Dialog Box

9. Click on he **On/Off Setting** button and complete the Edit On/Off Settings dialog box as shown in Figure 6-16. Click on **OK** in both dialog boxes and position the control on the panel.
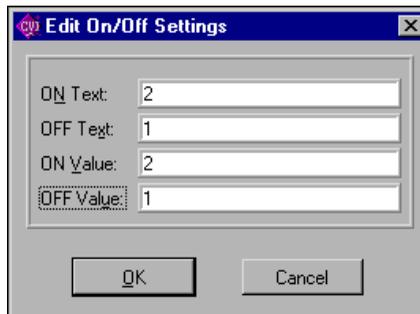
Figure 6-16. Channel Edit On/Off Settings Dialog Box

10. Select **Create»Input**.

11. Complete the Create Input Control dialog box as shown in Figure 6-17. Click on **OK** and position the control on the panel.
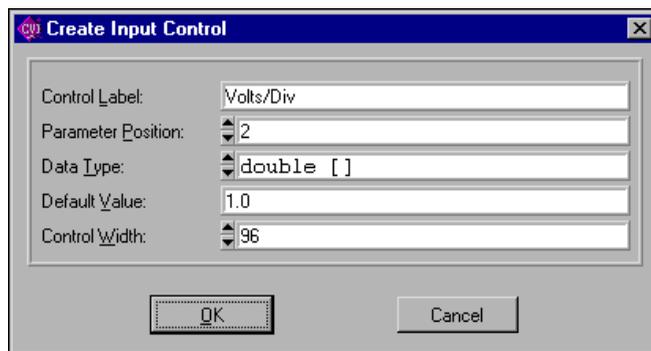
Figure 6-17. Volts/Div Create Input Control Dialog Box

12. Select **Create»Slide**.

13. Complete the Create Slide Control dialog box as shown in Figure 6-18.
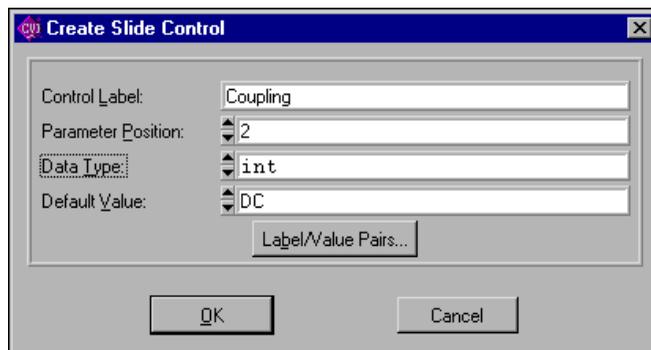
Figure 6-18. Coupling Create Slide Control Dialog Box

14. Click on **Label/Value Pairs** and complete the Edit Label/Value Pairs dialog box as shown in Figure. Click on **OK** in both dialog boxes and position the control on the panel.
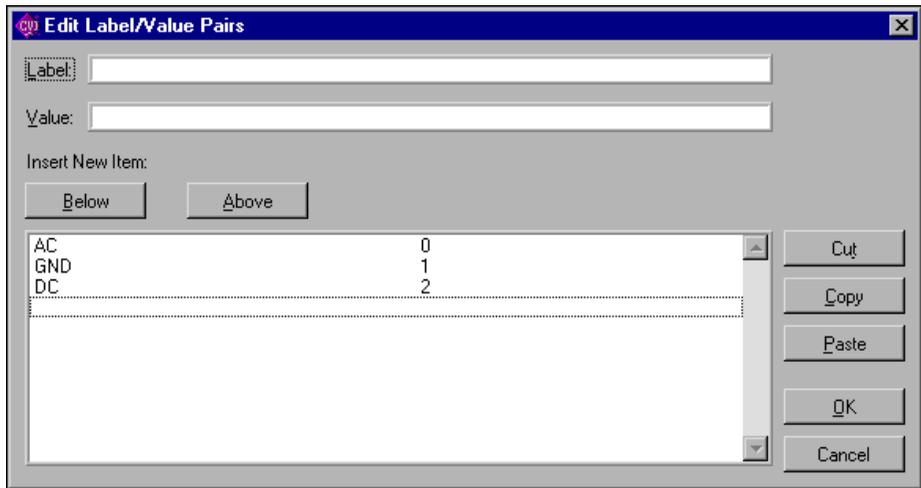
Figure 6-19. Coupling Edit Label/Value Pairs Dialog Box

15. Select **Create»Binary**.

16. Complete the Create Binary Control dialog box as shown in Figure 6-20.
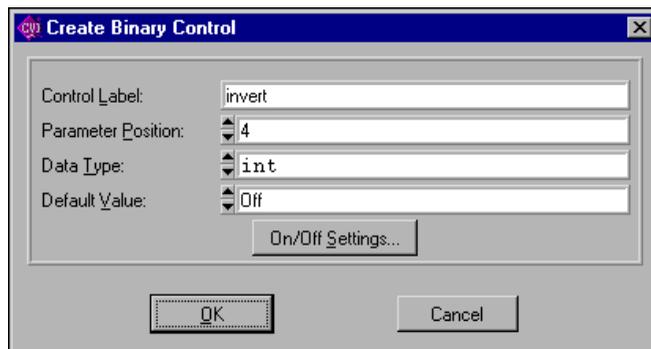
Figure 6-20. Invert Create Binary Control Dialog Box

17. Click on the **On/Off Settings** button and complete the Edit On/Off Settings dialog box as shown in Figure 6-21. Click on **OK** in both dialog boxes and position the control on the panel.
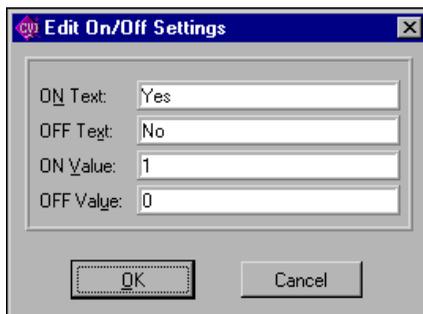
Figure 6-21. Invert Edit On/Off Settings Dialog Box

You now see the function panel shown in Figure 6-22.
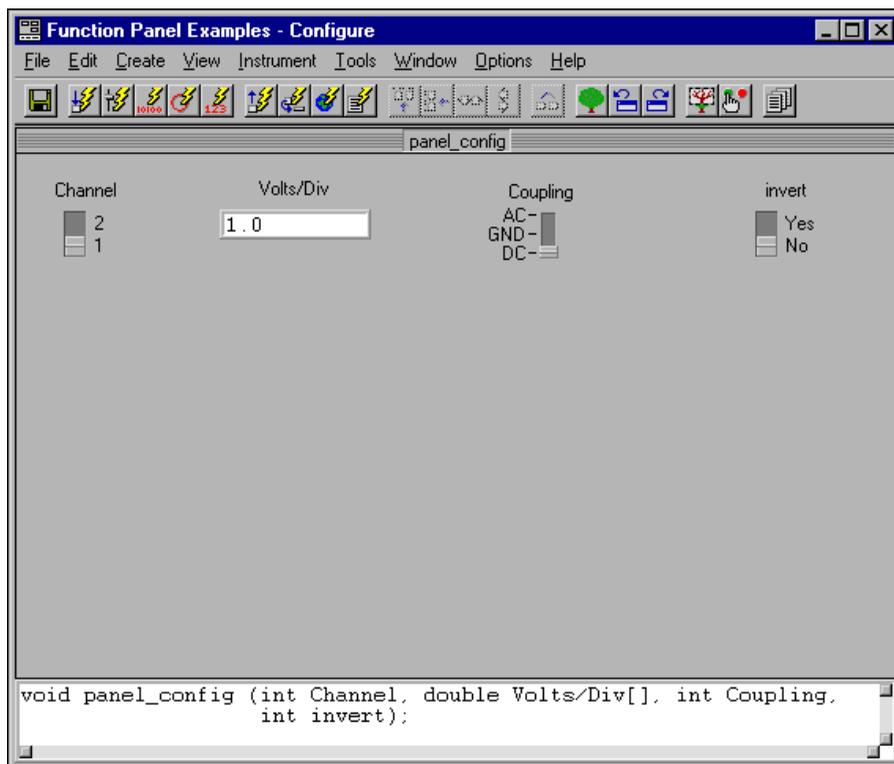
Figure 6-22. Function Panel Window

Example—Changing Control Type

In this example, you change the type of the Volts/Div control from an input control to a slide control. Follow these steps:

1. Be sure the function panel window from the previous example is active, in edit mode. Position the selection on the **Volts/Div** control.

2. Select **Edit»Change Control Type**. The dialog box shown in Figure 6-23 appears.
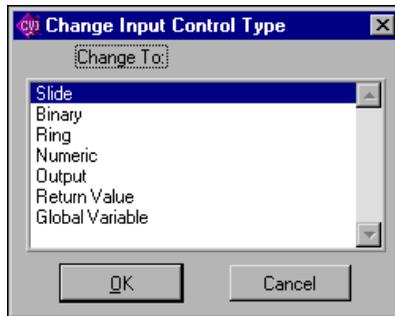
Figure 6-23. Change Input Control Type Dialog Box

3. Select **Slide** and click on **OK**. The Edit Slide Control dialog box appears.

4. Click on **Label/Value Pairs**. The Edit Label/Value Pairs dialog box appears.

5. Complete the dialog box as shown in Figure 6-24 and click on **OK**.
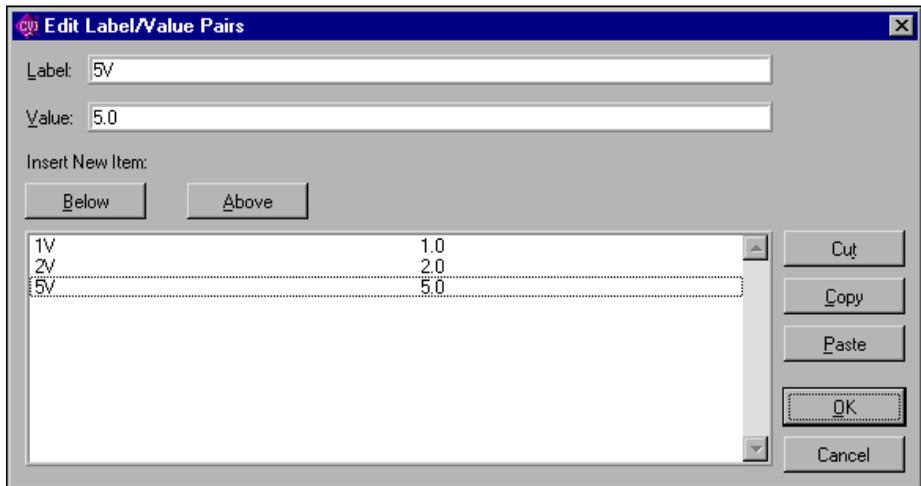
Figure 6-24. Volts/Div Edit Label/Value Pairs Dialog Box

6. Click on **OK** of the Edit Slide Control dialog box to replace the Volts/Div input control with a slide control.

Suppose that you meant this control to be a ring control instead of a slide control. Follow these steps:

1. Position the selection on the **Volts/Div** control.

2. Select **Edit»Change Control Type**.

3. Select **Ring**. The Edit Ring dialog box appears.

4. Click on **Label/Value Pairs**, leaving all other items unchanged. The Edit Label/Value Pairs dialog box appears. Notice that the slide control label value pairs remain.

5. Click on **OK**.

A ring control replaces the **Volts/Div** slide control on the function panel.

Example—Cutting and Pasting Controls

You frequently might want to cut and paste controls. In this example, you copy controls from one panel to another. Complete the following steps to copy a control.

1. Be sure the function panel from the previous example is active and in the Edit mode. Position the selection on the **Volts/Div** control.

2. Select **Edit»Control Help** f or click the secondary mouse button on the control.

3. Enter the following text in the Help Editor dialog box:

    ```
    This control specifies the volts per division setting of the
    oscilloscope.
    ```

4. Select **File»Save .FP File** and then select **File»Close** in the Help Editor dialog box.

5. With the selection still on the **Volts/Div** control, select **Edit»Copy Controls**.

6. Select **Edit»Paste**.

7. With the selection on the new control, select **Edit»Edit Control**.

8. Change the Ring Control Label to Volts/Div 2 and the parameter position to 2.

Notice in the Generated Code window that the config function now has an additional parameter, Volts/Div 2.

Complete the following steps to create a new function panel and copy a control to the panel.

1. Select **Option»Edit Function Tree**.

2. Create a function panel window with the following parameters. Type New Panel in the **Name** box and new_panel in the **Function Name** box.

3. Position the selection on the Configure node.

4. Select Edit**»Edit Function Panel Window** to return to the Configure panel.

5.   Position the selection on the control **Volts/Div 2**.

6.   Select **Edit»Cut Controls**.

7.   Press <Ctrl-Page Down> to move to the New Panel function panel.

8.   Select **Edit»Paste**.

The control appears on the panel. View the help information by selecting **Edit»Control Help**. Notice that the help information is copied with the control.

7

Adding Help Information

This chapter describes the types of help information available from an instrument driver and how you can create help information.

New Style Versus Old Style Help

LabWindows/CVI has two styles of online help for instrument drivers: new (Recommended) and old (LabWindows DOS). The old help style maintains compatibility with help information created in LabWindows version 2.3 or earlier. This help style uses the DOS/IBM character set so it can display special extended ASCII characters used by older instrument drivers.

The new help screen style uses the standard Windows character set and automatically displays the control help with control name and data type information.

There is also a difference in the type of help information that can be displayed. In either new or old style help, you can view instrument help, function class help, and control help. However, the help information for functions is displayed differently between the two styles. This difference has an effect only when you have multiple function panels on a single function panel window. In the new style, you can access function help for each function panel. In the old style, you can access the function panel window help, which describes all the functions contained in that function panel window.

National Instruments recommends that you use the new help style for all help information for instrument drivers that you create in LabWindows/CVI. Chapter 5, *Function Tree Editor*, gives more information on new and old style help. Most of the discussion in this chapter assumes you are using the new style help.

Help Options

The user of an instrument driver can view the following types of help information listed in Table 7-1.

Table 7-1. Types of Help Information

Type of Help	Location of Help
Instrument help	function class and function help dialog boxes
Function class help	dialog box that appears when a user selects an instrument from the **Instrument** menu
Function help (New style help only)	**Help** menu in the function panel window menu bar
Function panel window help	dialog box that appears when a user selects an instrument from the **Instrument** menu (Directly editable only in old style help. In the new style help, it is generated from the function help for each function in the window)
Control help	**Help** menu in the function panel window menu bar

Editing Help Information

There are four types of help information that you can enter: instrument, class, function, and control. You can edit instrument and class help from the Function Tree Editor and function and control help from the Function Panel Editor. Each of the editors has an **Edit** menu in the menu bar. Select **Edit»Edit Help** in the Function Tree Editor to add instrument and class help. Select **Edit»Function Help** and **Edit»Control Help** in the Function Panel Editor to add function panel and control help.

Complete the following steps to add help information:

1. From either the Function Tree Editor or the Function Panel Editor, select the item to which you want to add help information.

2. Select **Edit»Edit Help**, **Edit»Function Help**, or **Edit»Control Help** in the menu bar. The Help Editor window shown in Figure 7-1 appears.
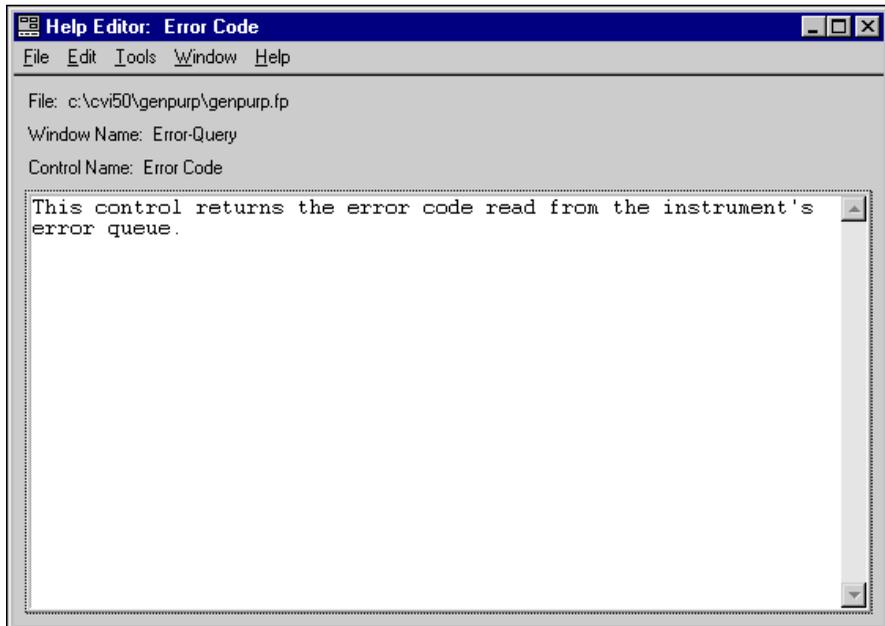
Figure 7-1. Help Editor Dialog Box

The Help Editor window contains a scrollable text box. You can scroll through the displayed text using the arrow keys or the scroll bars.

You see the following items on the Help Editor window menu bar:

• **File** lets you load, save, and manipulate files.

• **Edit** lets you edit the help text in the window.

• **Tools** lets you jump back to the function panel or function tree node that the help text in the window applies to.

• **Window** lets you specify which window to make active.

File

The **File** menu lets you create a new function tree, edit an existing function tree, save function panel information into a `.fp` and `.sub` file on disk, or add function panels to a project. The **File** menu operates like the **File** menu of the Project window. Refer to Chapter 3, *The Project Window*, in the *LabWindows/CVI User Manual*, for more information about the **File** menu.

For each IVI instrument driver, a `.sub` file accompanies the `.fp` file. The `.sub` file contains the information about the instrument driver attributes. You edit this information using the attribute editor. When you save the contents of a `.fp` file, LabWindows/CVI also saves the contents of the `.sub` file automatically.

Edit

You see the following items in the **Edit** menu:

- **Cut** deletes the selected text in the window and copies the text to the clipboard.

- **Copy** copies the selected text in the window to the clipboard without deleting the selected text.

- **Paste** inserts the contents of the clipboard into the window at the location of the cursor.

- **Delete** discards the selected text in the window without copying it to the clipboard.

- **Find** locates a particular text string in the Help Editor window.

- **Replace** replaces particular text in the Help Editor window with other text.

- **Revert** returns the most recently saved version of help text to the window.

Tools

You see the following items in the **Tools** menu:

- **Function Tree** brings up the Function Tree Editor window and jumps to the function tree node that contains the current help text.

- **Function Panel** brings up the Function Panel Editor window for the function panel that contains the current help text. If the help text applies to a particular control on the function panel, the **Function Panel** command selects the control.

Window

The **Window** menu lets you select which window to make active. The **Window** menu operates like the **Window** menu of the Project window. Refer to Chapter 3, *Project Window*, in the *LabWindows/CVI User Manual*, for more information about the **Window** menu.

Instrument Help

When you are viewing help information for a function class or function panel window, click the **Instrument Help** button to see help information about the instrument driver as a whole.

You can add instrument help information in the Function Tree Editor. Complete the following steps to enter the help information for the instrument.

1. In the Function Tree Editor, select the instrument node at the top of the function tree.

2. Select **Edit»Edit Help**. The Help Editor window appears. Alternatively, you can click on the instrument node with the right mouse button to display the Help Editor window.

3. Enter the help text into the Help Editor window.

Function Class Help

To display help information about a class of function panel windows, select the class in the Select Function Panel dialog box and click on the **Help** button.

You enter function class help information from the Function Tree Editor. Complete the following steps to add help information:

1. Select the class node in the function tree.

2. Select **Edit»Edit Help** in the Function Tree Editor menu bar. The Help Editor window appears. Alternatively, you can click on the class node with the right mouse button to display the Help Editor window.

3. Enter the help text into the Help Editor window.

Function Help (New Style Help Only)

When you use the new help style, you can display help information that pertains to a specific function panel by selecting **Help»Function** in the Function Panel menu bar. Alternatively, you can click on the background of the function panel with the right mouse button to display the function panel help.

When you use the new help style, you enter function panel help information from the Function Panel Editor. Complete the following steps to add function panel help:

1. Activate the function panel.

2. Select **Edit»Function Help** in the Function Panel Editor menu bar. The Help Editor window appears. Alternatively, you can click on the background of the function panel with the right mouse button to display the Help Editor window.

3. Enter the help text into the Help Editor window.

Refer to Chapter 5, *Function Tree Editor*, for more information on changing between the new and old style help modes.

Function Panel Window Help (Old Style Help Only)

When you use the old help style, you can display help information that pertains to a function panel window by selecting **Help»Window** in the Function Panel menu bar. Alternatively, you can click on the background of the function panel window with the right mouse button to display the function panel help.

When you use the old help style, you enter function panel window help information from the Function Panel Editor. Complete the following steps to add function panel window help.

1. Select **Edit»Window Help** in the Function Panel Editor menu bar. The Help Editor window appears. Alternatively, you can click on the background of the function panel window with the right mouse button to display the Help Editor window.

2. Enter the help text into the Help Editor window.

Refer to Chapter 5, *Function Tree Editor*, for more information on changing between the new and old style help modes.

Control Help

You can display help information for a specific function panel control by selecting the control and selecting **Help»Control** in the Function Panel menu bar. Alternatively, you can click on the control with the right mouse button to display the control help.

You enter control help information from the Function Panel Editor.

Complete the following steps to add help information for a function panel control.

1. Select the control.

2. Select **Edit»Control Help** in the Function Panel Editor menu bar. The Help Editor window appears. Alternatively, you can click on the control with the right mouse button to display the Help Editor window.

3. Enter the help text into the Help Editor window.

Help Information Examples

The following examples teach you about creating and editing help information, specifically adding instrument and panel help information from the Function Tree Editor, adding panel and control help information from the Function Panel Editor, and cutting and pasting help information between controls.

Example—Adding Help Information in the Function Tree Editor

In this example, you add instrument and function class help information to a function tree. Complete the following steps to create a new instrument and function tree.

1. Choose **File»New»Function Tree (*.fp)**.

2. Choose **Create»Instrument**.

3. Type Help Information Examples as the Name and help as the Prefix. Click on **OK**.

4. Choose **Create»Class**.

5. Enter Class 1 as the Name. Click on **OK**.

6. Select the line beneath the name Class 1.

7. Choose **Create»Function Panel Window**.

8. Enter Function 1 as the Name and fun1 as the Function Name. Click on **OK**.

The new function tree appears as shown in Figure 7-2.
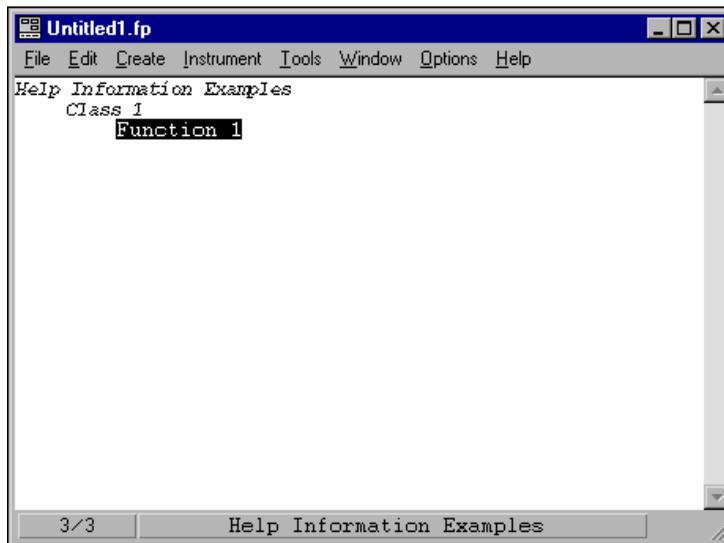
Figure 7-2. Sample Function Tree

The first level of help information is associated with the name of the instrument driver.

Complete the following steps to add help information to the top level of the tree.

1. Select the name Help Information Examples.

2. Select **Edit»Edit Help**, or click on the instrument name with the right mouse button. The Help Editor window appears.

3. Enter the following help information.

    ```
    This driver was created to illustrate how to add help text to an
    instrument driver.
    ```

4. Select **File»Save .FP File** then **File»Close** to save the text and remove the Help Editor window.

Complete the following steps to add help information to `Class 1`.

1. Select the name `Class 1`.

2. Select **Edit»Edit Help** or click on the class name with the right mouse button. The Help Editor window appears.

3. Enter the following help information.

    ```
    An example function class. The functions in this class are:
    Function 1—The only function in the class.
    ```

4. Select **File»Save .FP File** then **File»Close** to save the text and remove the Help Editor window.

Complete the following steps to view the help information.

1. Select **Help Information Examples** from the **Instrument** menu. The Select Function Panel dialog box appears.

2. Select `Class 1` and click on **Help** to display the Class Help window.

3. Click on **Instrument Help** to display the Instrument Help window.

4. Click on **Done** to exit the Instrument Help window.

5. Click on **Done** to exit the Class Help window.

6. Click on **Cancel** to exit the Select Function Panel dialog box.

Example—Adding Help Information in the Function Panel Editor

In this example, you add help information to function panels and function panel controls from the Function Panel Editor. Double-click on `Function 1` from the previous example.

Complete the following steps in the Function Panel Editor to modify the help information for the function panel.

1. Select **Edit»Function Help** from the **Edit** menu. The Help Editor window appears.

2. Enter the following help information.

    ```
    This function is the only function in Function Class.
    ```

3. Select **File»Save .FP File** then **File»Close** to save the text and remove the Help Editor window.

Help information also is associated with each of the controls in a function.

Complete the following steps to add a control to the current panel.

1. Select **Create»Input**.

2. Enter `Input Control` for the **Control Label**.

3. Click on **OK**.

Complete the following steps to add help information to the control.

1. Select the control.

2. Select **Edit»Control Help**. Alternatively, click the right mouse button on the control. The Help Editor window appears.

3. Enter the following text in the Help Editor window.

4. This control is an input control on the Function 1 function panel.

5. Select **File»Save .FP File** then **File»Close** to save the text and remove the Help Editor window.

You have now added help information to all possible locations. Select **Options»Operate Function Panel** and then view the help information for the function panel.

Example—Copying and Pasting Help Text

In this exercise, you copy text between function panels, controls, and instruments. The clipboard retains its contents as you move between controls, function panels, and even instruments. Help text also stays with a control or function panel that is cut, copied, or pasted.

Complete the following steps to copy the help information between controls on different panels.

1. Create a new function panel window from the Function Tree Editor. Type `Function 2` in the Name box and `fun2` in the Function Name box.

2. The `Function 1` function panel should be on the screen in Edit mode. Double-click on `Function 1` in the Function Tree Editor.

3. Select **Create»Global Variable**.

4. Type `Status` in the **Control Label** box and `ibsta` in the **Global Variable Name** box. Leave all other items at their default settings. Click on **OK**.

5. Add the following help information to the Global Control.

   ```
   This control displays the status of GPIB function calls.

   Errors:
   0         Success
   non-zero  See the STATUS control on any GPIB Library
             function panel
   ```

6. Select **File»Save .FP file** then **File»Close** to save the text.

7. Select the Status control. Select **Edit»Copy Controls**.

8. Press <Ctrl-Page Down> to display the `Function 2` function panel.

9. Select **Edit»Paste**. The Status control appears on the function panel.

10. Select **Options»Operate Function Panel** and view the help information. Notice that the help information stays with a control when you copy that control.

Complete the following steps to copy the help text *without copying the control*.

1. Select **Options»Edit Function Panel** Window.

2. Select **Create»Global Variable**.

3. Complete the Create Global Variable Control dialog box as follows. Type `Error` in the **Control Label** box and `iberr` in the **Global Variable Name** box. Leave all other items at their default settings. Click on **OK**.

4. Select the **Status** control.

5. Select **Edit»Control Help** or click the right mouse button on the control.

6. Select all the text in the dialog box.

7. Select **Edit»Copy**.

8. Select **File»Close**.

9. Select the Error control.

10. Select **Edit»Control Help** or click the right mouse button on the control.

11. Select **Edit»Paste**. The help information appears in the window.

12. Modify the text so it reads as follows:

```
This control displays the value of the GPIB global error variable.
The control displays the value of the error only when the STATUS
control is non-zero.

Errors:
0         Success
non-zero  See the ERROR control on any GPIB Library
          function panel
```

In these examples, you have learned to copy or move text from one control to another. Use the same methods to copy and move help text between various locations. For example, copying and moving panel, instrument, window, and control help within an instrument driver or across instrument drivers.

8

Programming Guidelines for Instrument Drivers

This chapter contains general procedures and guidelines for creating IVI instrument 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. This chapter shows you how to handle common situations you might encounter. This chapter contains specific guidelines for GPIB, VXI, and RS-232 instruments. However, you can apply this information to instruments that use other I/O interfaces.

The examples in the *LabWindows/CVI Online Help* illustrate many of the procedures and guidelines in this chapter.

Generating Driver Files

Always use the Instrument Driver Development Wizard to create your initial instrument driver files. You can use the wizard to create your driver files from a template or an existing driver. Refer to Chapter 3, *Developing an Instrument Driver*, for information about using the Instrument Driver Development Wizard to create your instrument driver files.

Selecting a Template

Table 8-1 lists the templates that the Instrument Driver Development Wizard uses. Select a template that best matches the capabilities of your instrument.

Table 8-1. Instrument Driver Class Templates

Template	Description
General Purpose Template	Use this template only for instruments types for which there is no class template. The template contains all the functions and attributes that IVI and VXI*plug&play* require. It also has utility routines that implement typical low-level driver operations
Digital Multimeter Template	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 Template	Controls basic operations such as outputting standard waveforms. It also includes the ability to generate arbitrary waveforms and configure the modulation.
Oscilloscope Template	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 Template	Controls basic operations such as outputting DC power and configuring the over-voltage and over-current protection. It also includes advanced features such as producing transient waveforms and monitoring the output voltage and current.
Switch Template	Controls basic channel connect and disconnect operations. It also includes advanced switch features such as scanning.

Note   The instrument class templates incorporate all the features of the general purpose template as well as the features in Table 8-1.

Procedures for Customizing Wizard-Generated Driver Files

After you create the instrument driver files, you customize them to match the requirements of your instrument. To customize the instrument driver files, you can 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 later in this chapter for more information and guidelines regarding attributes and functions.

Modifying Existing Attributes and Functions

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

Complete the following steps to modify an attribute.

1. Edit the range table with the Range Tables dialog box of 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, fill in 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, you must 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 of 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, you must 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 did 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. Fill in 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** command 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 give you 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. There are three types of attributes in your instrument driver:

• Attributes that the IVI engine defines

• Attributes that the instrument class defines

• Attributes that only your instrument driver defines

You 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
#define FL45_ATTR_RESOLUTION            IVIDMM_ATTR_RESOLUTION
```

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 as well. 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, you 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 example below 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, you increment the offset.

Attribute Value Definitions

You define values for public attributes in your instrument driver header file. Typically, the instrument class defines values for the attributes that it defines. You 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, you must 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_DRIVER_EXT_BASE + 1)
```

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

The IVI engine allows you to create attributes with only a subset of the data types you can use with instrument drivers. Table 8-2 shows what data types you can use for attributes.

Table 8-2. 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 you must 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. You use an alternative mechanism in read and coerce callbacks for `ViString` attributes. For these attributes, you 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;
}
```

You 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 only has 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. This is particularly important if you want your instrument driver to support deferred updates with buffered I/O. When processing deferred updates, the buffered I/O callback buffers the command strings from multiple `Ivi_SetAttribute` operations into one command string. 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 `viScan` and `viRead` functions.

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.

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

You 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. 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 later 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 usually 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 *Attributes that Represent a Continuous Range* and *Attributes that Represent a Continuous Range with Discrete Settings* 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 later in this chapter for information on how to structure check and coerce callbacks.

The following sections illustrate how to use multiple static range tables and use a dynamic range table.

Using Multiple Static Range Tables

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

*   Create a static range table for each configuration.
*   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.

You 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;
}
```

You 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, do the following:

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 8-3 shows the attribute callbacks for which you can use this approach.

Table 8-3. 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

You 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 them in a fashion that is 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 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 Online 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;
   ViChar    rdBuffer[BUFFER_SIZE];
   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( viWrite (io, "VAL1?", 5, VI_NULL));
       viCheckErr( viRead (io, rdBuffer, BUFFER_SIZE-1, &retCnt));
       rdBuffer[retCnt] = 0;

       if (Scan (rdBuffer, "%f", reading) != 1)
```

```
                    viCheckErr( VI_ERROR_INV_RESPONSE );
                }
        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** You must not attempt to lock or unlock the IVI session in the `Prefix_init`,
`Prefix_InitWithOptions`, `Prefix_IviInit`, and `Prefix_IviClose` functions.

Refer to the*LabWindows/CVI Online 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. You 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`. You 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. You always perform this type of checking regardless of the value that the `Ivi_RangeChecking` function returns. If the address is NULL, you 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:

• 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. This rule does not apply to the following functions:

*Prefix*_init
*Prefix*_InitWithOptions
*Prefix*_close
*Prefix*_IviClose
*Prefix*_error_query
*Prefix*_error_message

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 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 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
 * Purpose:    Configures the common attributes of the DMM.
 *******************************************************************/

ViStatus _VI_FUNC Fl45_Configure (ViSession vi, ViInt32 measFunction,
                        ViReal64 range, ViReal64 resolution,
                        ViReal64 acMinFreq, ViReal64 acMaxFreq)
{
   ViStatus    error = VI_SUCCESS;

   checkErr( Ivi_LockSession (vi, VI_NULL));

   viCheckParm( Ivi_SetAttributeViInt32 (vi, VI_NULL,
                        FL45_ATTR_FUNCTION, 0, measFunction), 2,
                        "Measurement Function");

   viCheckParm( Ivi_SetAttributeViReal64 (vi, VI_NULL,
                        FL45_ATTR_RANGE, 0, range), 3, "Range");

   viCheckParm( Ivi_SetAttributeViReal64 (vi, VI_NULL,
                        FL45_ATTR_RESOLUTION, 0, resolution), 4,
                        "Resolution");
      /*
         Set the AC min/max frequencies only if configuring an AC
         measurement
      */
   switch (measFunction)
      {
      case FL45_VAL_AC_VOLTS:
      case FL45_VAL_AC_CURRENT:
      case FL45_VAL_AC_PLUS_DC_VOLTS:
      case FL45_VAL_AC_PLUS_DC_CURRENT:
```

```
                viCheckParm( Ivi_SetAttributeViReal64 (vi, VI_NULL,
                             FL45_ATTR_AC_MIN_FREQ, 0, acMinFreq), 5,
                             "AC Min Frequency");

                viCheckParm( Ivi_SetAttributeViReal64 (vi, VI_NULL,
                             FL45_ATTR_AC_MAX_FREQ, 0, acMaxFreq), 6,
                             "AC Max Frequency");
            break;
        }
    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 multi-channel 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, do the following:

- Use the front panel channel names as the channel strings.

- 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

An important consideration in developing an instrument driver is making the driver portable across multiple compilers and operating systems. There are established guidelines 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 8-4 for a list of VISA data types.

Table 8-4. VISA Data Types

VISA Type Name	Definition
ViInt16	Signed 16-bit integer
ViUInt16	Unsigned 16-bit integer
ViInt32	Signed 32-bit integer
ViUInt32	Unsigned 32-bit integer
ViReal64	64-bit floating-point number
ViInt16[]	An array of ViInt16 values
ViInt32[]	An array of ViInt32 values
ViReal64[]	An array of ViReal64 values

Table 8-4. VISA Data Types (Continued)

VISA Type Name	Definition
`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
`ViBoolean[]`	An array of `ViBoolean` values

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 8-5 contains the macro definition for each platform.

Table 8-5. VISA I/O Library Macros

Macro	LabWindows/ CVI Environment (Windows 3.1)	Outside the LabWindows/CVI Environment (Windows 3.1)	Windows 95/NT	UNIX
_VI_FUNC	_pascal	_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, the instrument driver developer usually transfers 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. Specifically, LabWindows/CVI is a multi-platform application and must account for byte ordering on different platforms. 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 in relation to Intel 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 8-6 presents a scheme for determining error values. It lists predefined error codes for instrument drivers.

Table 8-6. 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 Online 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 8-7 and 8-8 are resolved in the file vpptype.h. By including the file vpptype.h in your instrument driver header file, you can use these defined names in your instrument driver, and users of your driver can use them in their application programs.

Table 8-7. 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

Table 8-7. Instrument Driver Completion and Warning Codes (Continued)

Completion Code	Description	Error Number
VI_WARN_NSUP_REV_QUERY	Revision query not supported	0x3FFC0105L
—	Instrument-specific warnings	0x3FFC0800 to 0x3FFC0FFF

Table 8-8. 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_PARAMETER1	Parameter 1 out of range	0xBFFC0001L
VI_ERROR_PARAMETER2	Parameter 2 out of range	0xBFFC0002L
VI_ERROR_PARAMETER3	Parameter 3 out of range	0xBFFC0003L
VI_ERROR_PARAMETER4	Parameter 4 out of range	0xBFFC0004L
VI_ERROR_PARAMETER5	Parameter 5 out of range	0xBFFC0005L
VI_ERROR_PARAMETER6	Parameter 6 out of range	0xBFFC0006L
VI_ERROR_PARAMETER7	Parameter 7 out of range	0xBFFC0007L
VI_ERROR_PARAMETER8	Parameter 8 out of range	0xBFFC0008L
—	Instrument-specific errors	0xBFFC0800 to 0xBFFC0FFF

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 *Error Reporting* and *Error Macros* sections in the *LabWindows/CVI Online 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 to precede all user-callable function and global variable names. The prefix 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 `vpptype.h` in the include file for your instrument driver. Include the file `visa.h` in the source code for your instrument driver.

- Declare `void` any function that does not return a value. 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.

- Test all the instrument drivers you create. Test them in LabWindows/CVI and in standalone applications.

- You must not declare 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 only uses 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. When your function panels resemble others in the library, users feel more comfortable with your instrument driver.

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 8-1 shows the Fluke 45 instrument driver function tree.
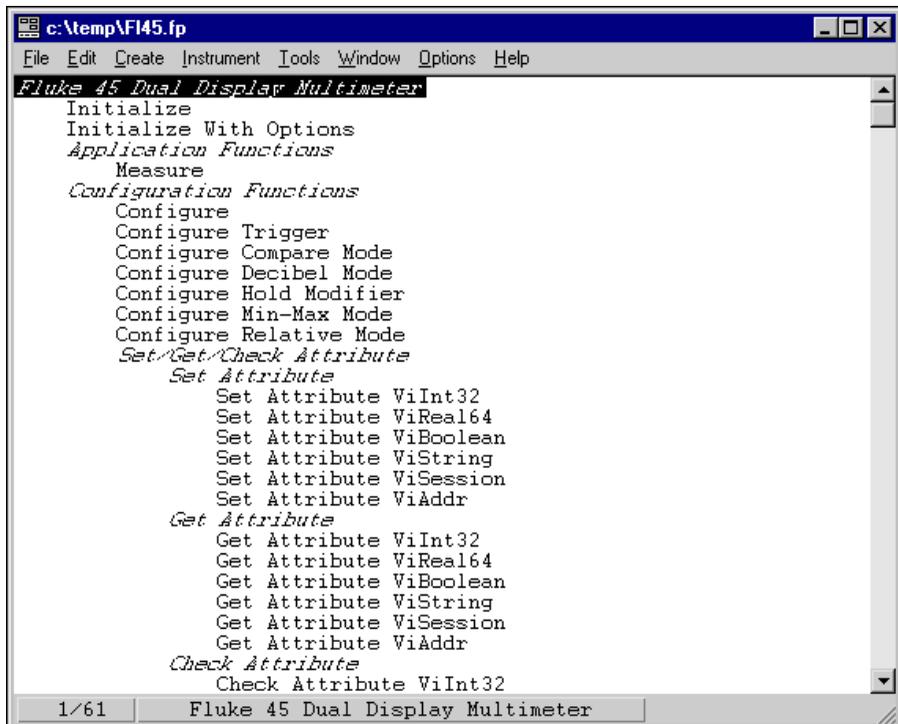
Figure 8-1. Fluke 45 Digital Multimeter Function Tree

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 so that users feel familiar with your instrument driver.

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:

- Online help from within LabWindows/CVI function trees and function panels.

- A `.doc` file you distribute with the instrument driver files. It 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. Relevant help information in a consistent format makes using the instrument driver easier. 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 8-2 shows instrument help for the Fluke 45 instrument driver.
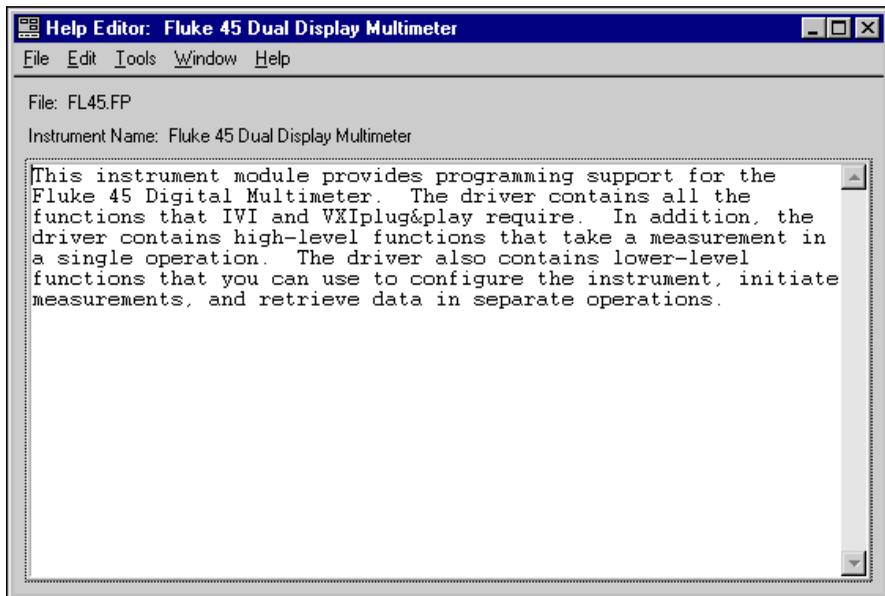
Figure 8-2. Fluke 45 Instrument Help

- Function class help briefly describes all the functions and subclasses beneath the selected function class. Figure 8-3 shows function class help from the Fluke 45 instrument driver.
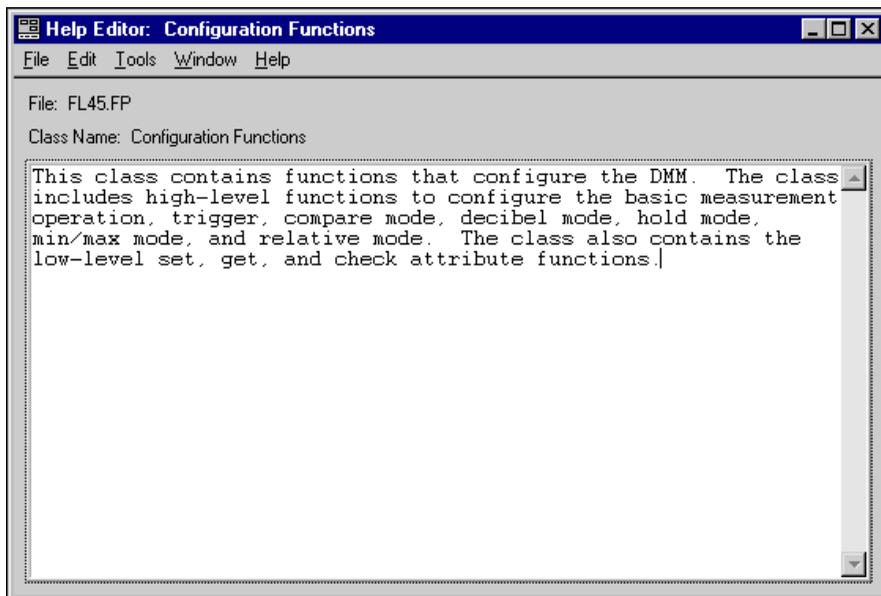
Figure 8-3. Fluke 45 Function Class Help

•   Function panel help describes the function call. Figure 8-4 shows the function panel help from the Fluke 45 instrument driver.
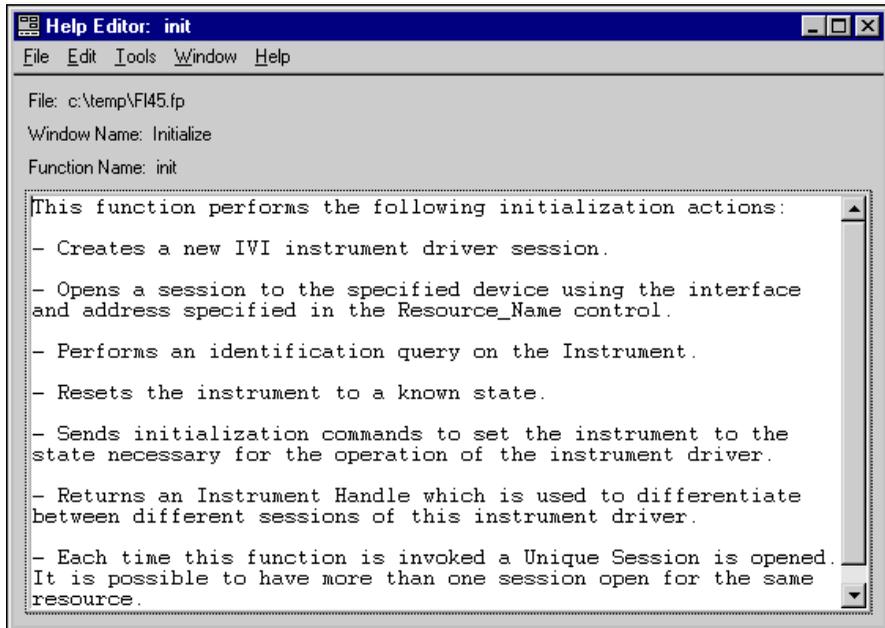
Figure 8-4. Fluke 45 Function Panel Help

- Control help contains a description of the parameter, the valid range, and the default value. Figure 8-5 shows an example of function panel control help from the Fluke 45 instrument driver.
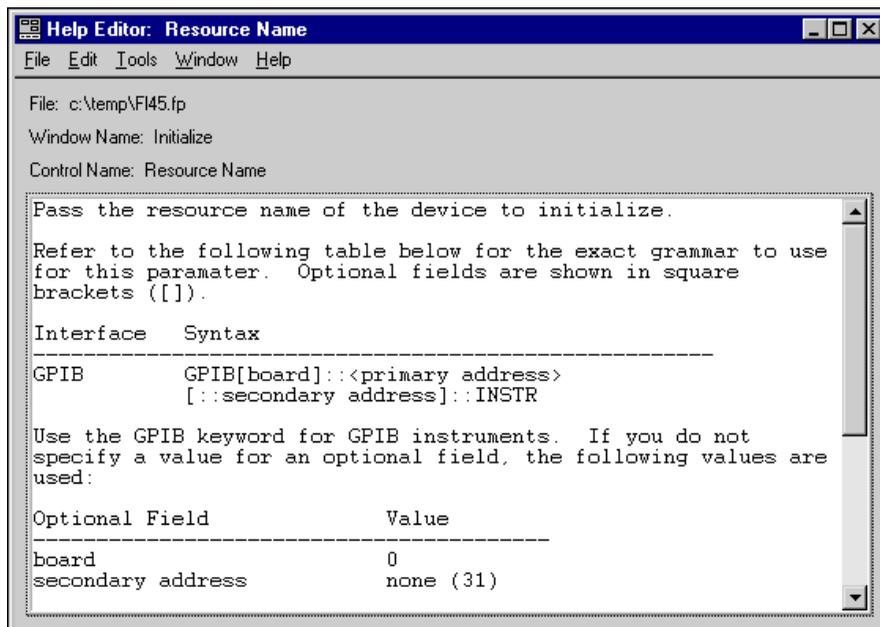
Figure 8-5. Fluke 45 Function Panel Control Help

•   Status help contains a description of the parameter and the possible error values.
    Figure 8-6 shows an example of status control help from the Fluke 45 instrument driver.
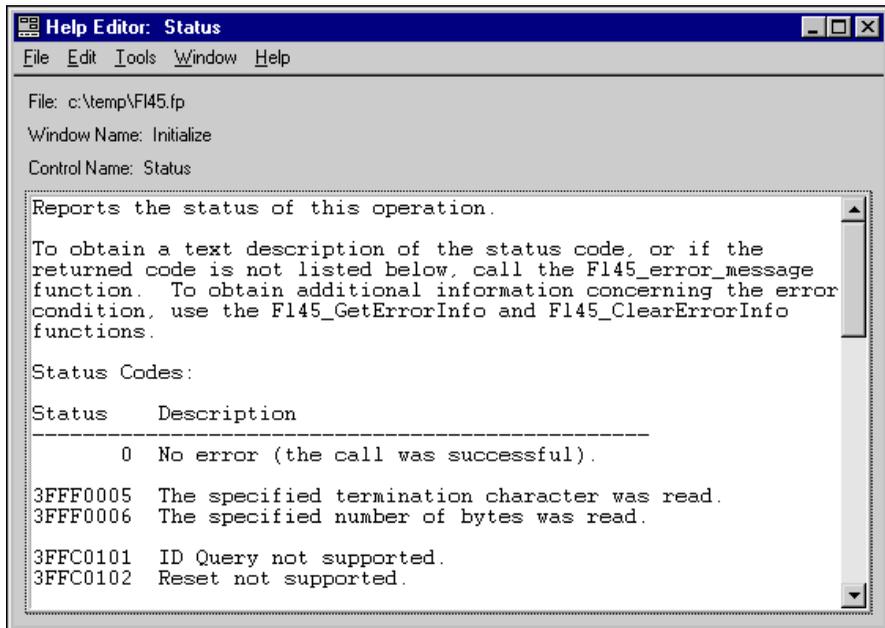
Figure 8-6. Fluke 45 Function Panel Status Control Help

.doc File

The .doc file is an ASCII text file that contains 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

You should give the `.doc` file 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.

Programming Guidelines for VXI Instruments

When developing drivers for VXI register-based devices, you must 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 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 that are 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

____ 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, whichever 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 includes the following:

    ____ 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 has all error/warning status codes that the function might return

    ____ 3. Have deleted all modification instructions

III. Source File

____ A. Includes standard instrument driver comments:

    ____ 1. Instrument manufacturer and name

    ____ 2. Author identification

    ____ 3. Modifications history

____ B. Includes the `visa.h` and instrument driver header files

____ 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

_____ 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

_____ 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. You have deleted all modification instructions

IV. Include file

____ A. Includes the `vpptype.h` and `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

V. Document file

____ A. The LabWindows/CVI-generated document file is properly edited

____ B. Unsupported platform information is removed

____ C. The document file contains no redundant information such as variable name and variable type

9

Instrument Driver Examples

This chapter shows you how to create an IVI instrument driver. The following examples in this chapter can 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 instrument 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 Instrument Driver Files with the Instrument Driver Development Wizard

To create your IVI instrument driver, you first use the Instrument Driver Development Wizard to create the driver files. To invoke the wizard, select **Tools»Create IVI Instrument Driver**. 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. Click on the **Next** button on the welcome panel to begin. The Select an Instrument Driver panel appears as shown in Figure 9-1.
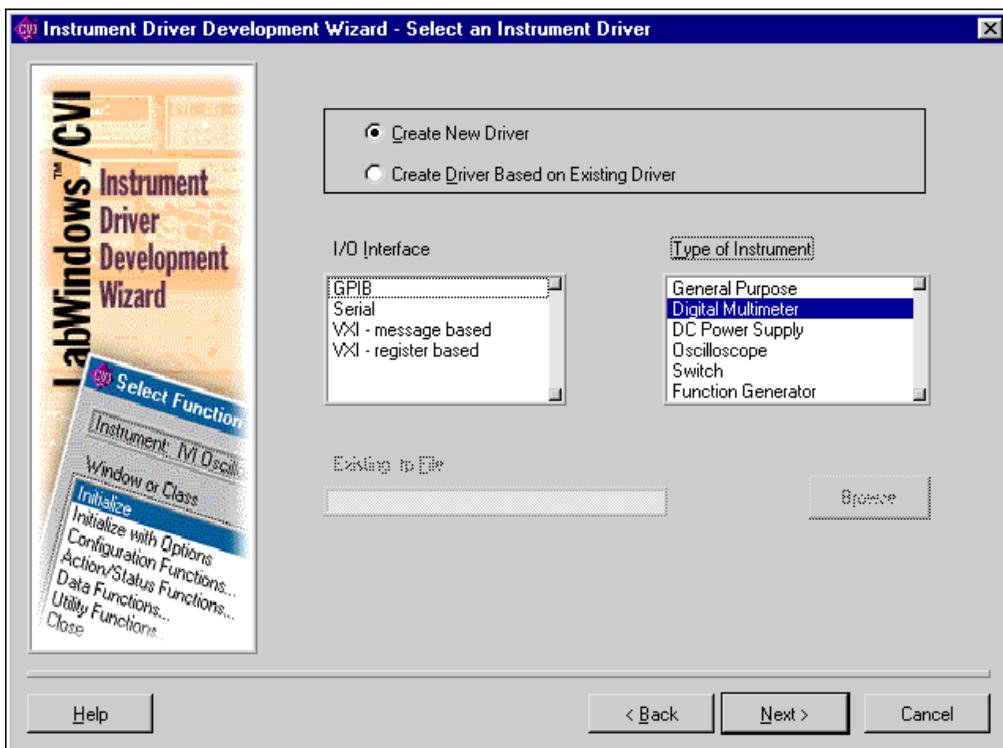
Figure 9-1. Select an Instrument Driver Panel

Enter the following information in the wizard panel.

- Select the **Create New Driver** option.
- Select GPIB in the **I/O Interface** list box.
- Select Digital Multimeter in the **Type of Instrument** list box

Click on the **Next** button to continue. At any time, you can click on the **Back** button to return to a previous panel and change the information.

The General Information panel appears as shown in Figure 9-2.
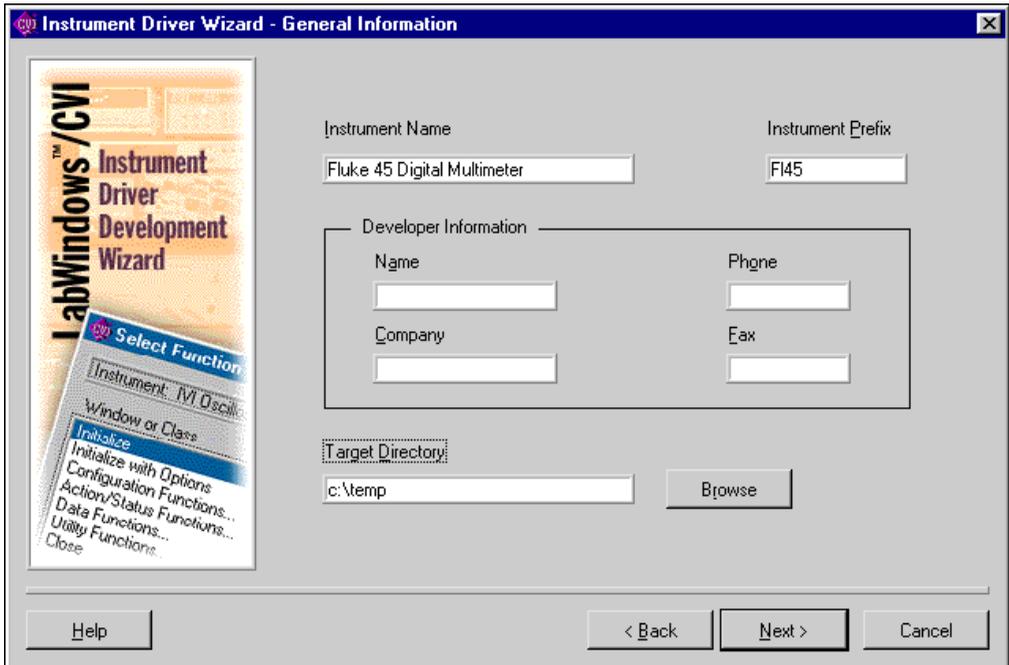
Figure 9-2. General Information Panel

Enter the following general instrument driver information.

- Enter Fluke 45 Digital Multimeter in the **Instrument Name** control.

- Enter Fl45 in the **Instrument Prefix** control.

- Enter your name, company, phone number, and fax number in the Developer Information section.

- Click on the **Browse** button to select a target directory for the new instrument driver.

Click on the **Next** button to continue.

The General Command Strings panel appears as shown in Figure 9-3.
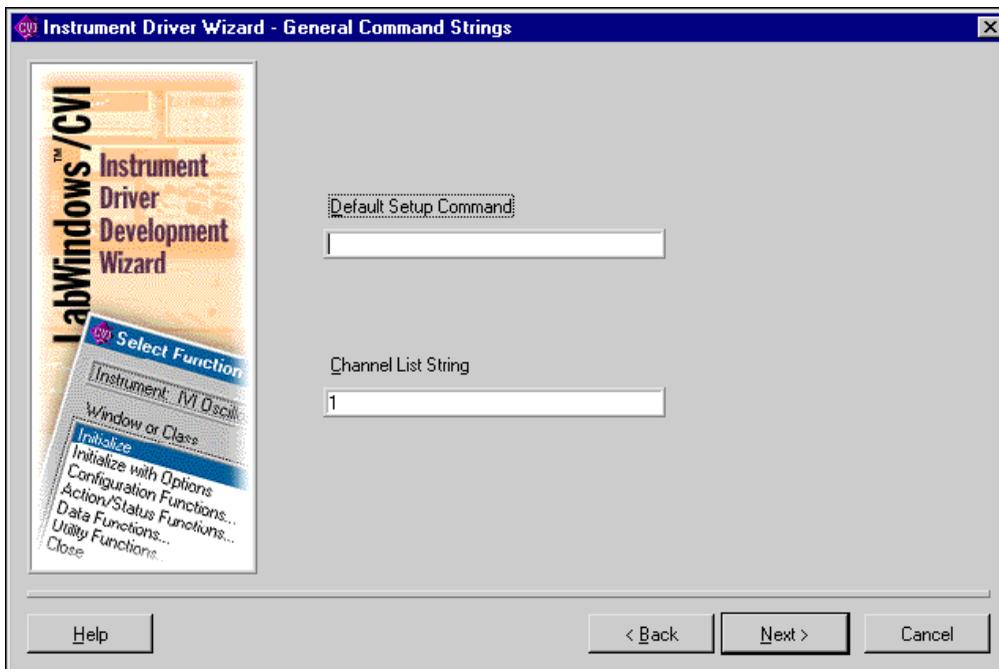
Figure 9-3. General Command Strings Panel

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. You 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. 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 multi-channel 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, you might want to use the channel names from the front panel.

For the Fluke 45 and all other single-channel devices, enter 1 in the **Channel List String** control.

Click on the **Next** button to continue.

The Standard Operations panel, shown in Figure 9-4, allows you to select which standard operations your instrument supports.
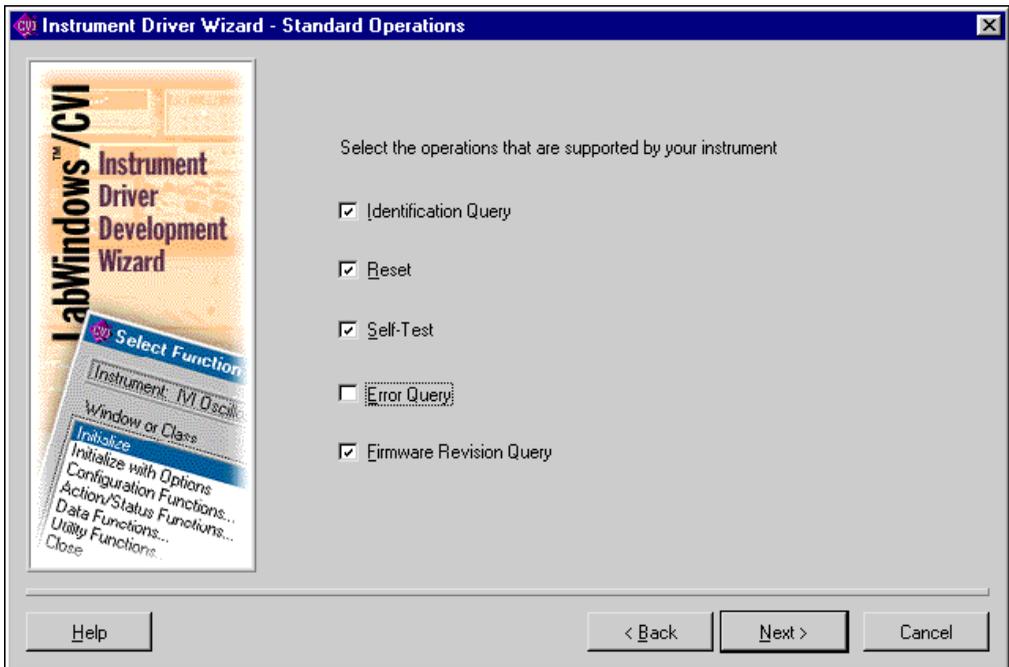
Figure 9-4. Standard Operations Panel

The Fluke 45 supports the identification query, reset, self-test, and firmware revision query operations, but it does not support an error query operation. Deselect the **Error Query** option.

Click on the **Next** button 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 9-5.
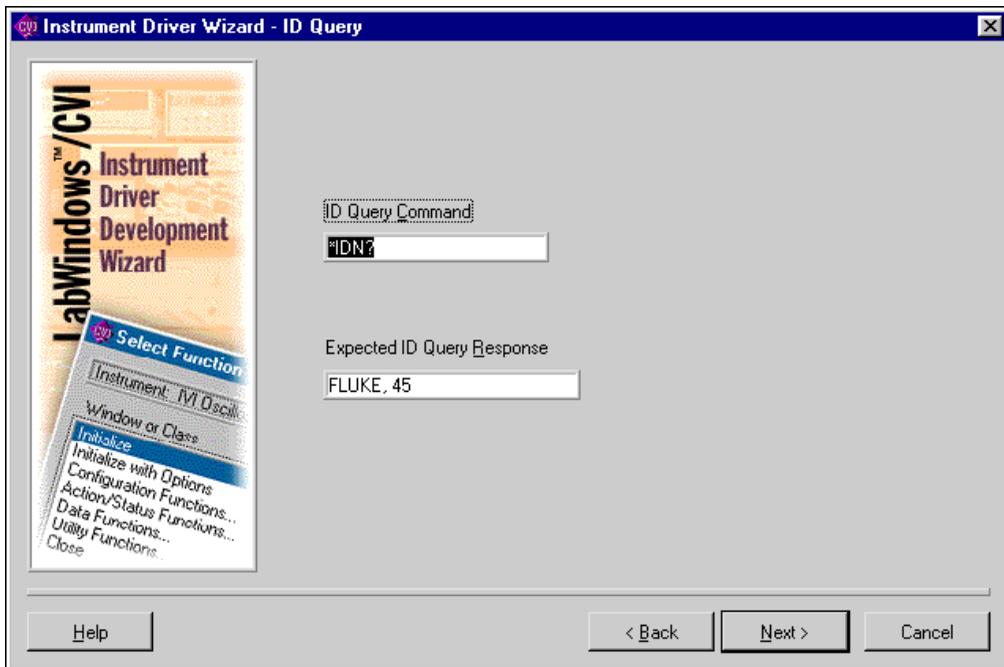
Figure 9-5. ID Query Panel

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.

- Enter *IDN? in the **ID Query Command** control. This is the default.
- Enter FLUKE, 45 in the **Expected ID Query Response** control.

Click on the **Next** button to continue.

The Reset panel appears as shown in Figure 9-6.
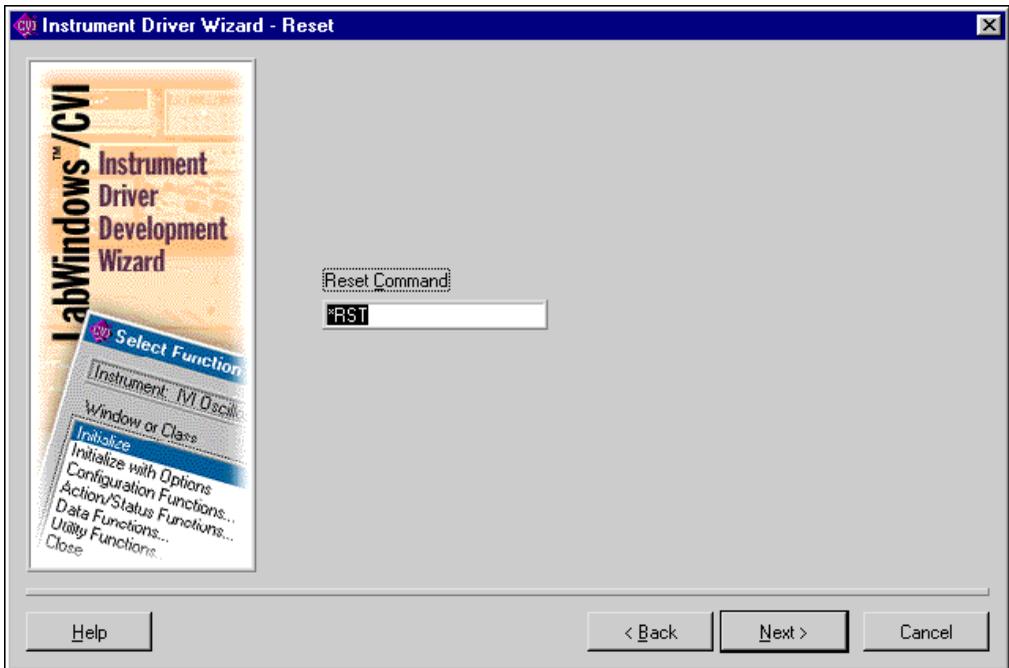
Figure 9-6. Reset Panel

The *RST command resets the Fluke 45. Enter *RST in the **Reset Command** control. This is the default.

Click on the **Next** button to continue.

The Self Test panel appears as shown in Figure 9-7.
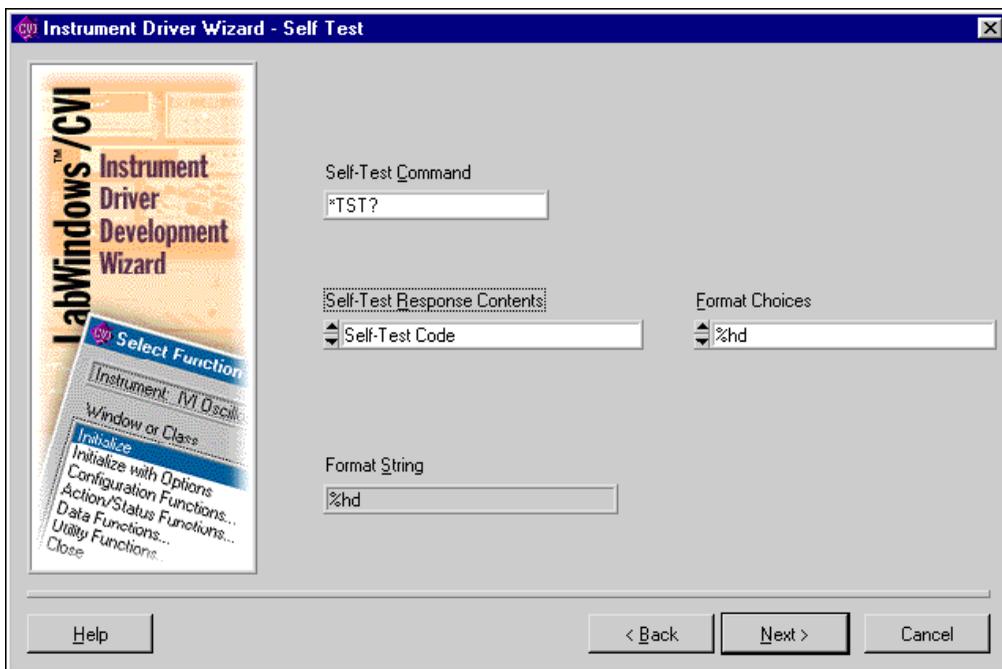
Figure 9-7. Self-Test Panel

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.

- Enter *TST? in the **Self-Test Command** control. This is the default.

- Select **Self-Test Code** from the **Self-Test Response Contents** ring control.

- 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.

Click on the **Next** button to continue.

The Revision panel appears as shown in Figure 9-8.
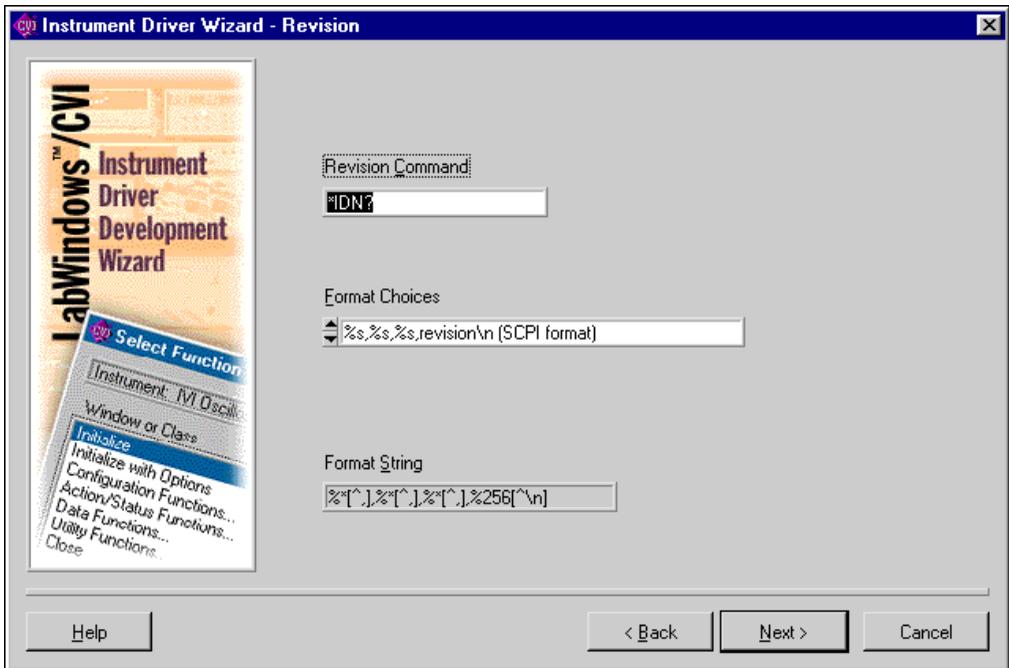
Figure 9-8. Revision Panel

The response to the *IDN? command also includes the revision of the Fluke 45 instrument.

- Enter *IDN? in the **Revision Command** control. This is the default.

- Select %s,%s,%s,revision\n(SCPI format) from the **Format Choices** ring control. This 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.

Click on the **Next** button to continue.

The Test panel appears as shown in Figure 9-9.
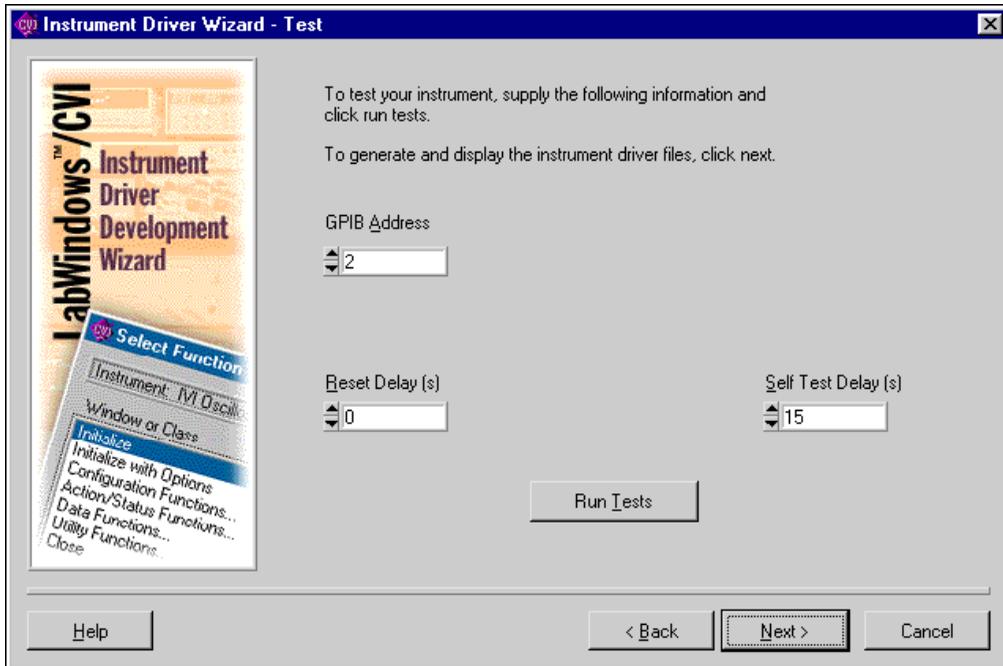
Figure 9-9. Test Panel

If you have a Fluke 45 instrument available and can connect it to the computer, the wizard tests the commands you enter in the previous panels by sending the commands to the instrument and displaying the responses.

Run the test as follows:

- Enter the address of your instrument in the **GPIB Address** control.

- Enter 0 in the **Reset Delay(s)** control.

- The Fluke 45 takes 15 seconds to perform the self-test operation. Enter 15 in the **Self-Test Delay(s)** control.

- Click on the **Run Tests** button.

Figure 9-10 shows the Test Results panel. This panel displays the operations the test performs and the results of these operations. Click on the **Done** button to return to the previous panel.
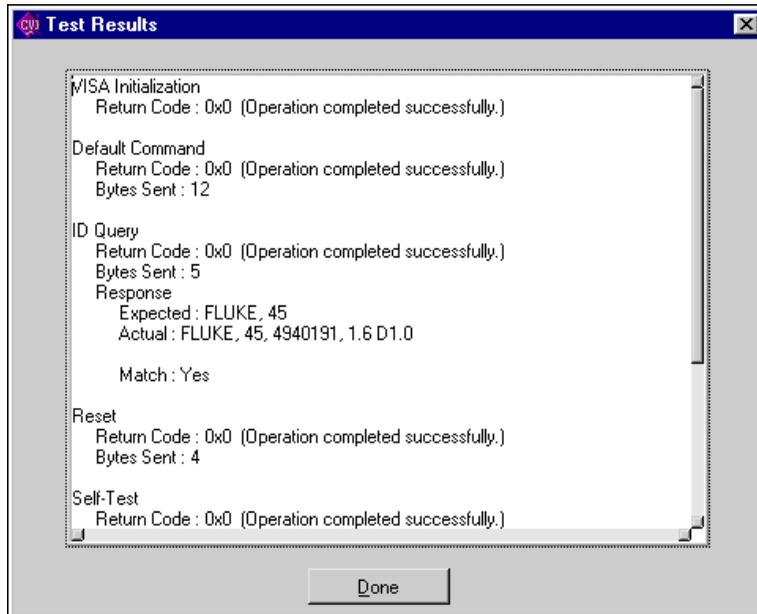
Figure 9-10. IVI Test Results Panel

If any of the operations generate errors or the responses are not as you expect, you can click on the **Back** button to return to the corresponding panel and change the information. You can then click on the **Next** button to return to the Test panel to verify the new information.

When you click on the **Next** button 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 Online 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 digital

multimeters. These functions and attributes have example code with instructions on how to modify the code for a specific DMM.

Figure 9-11 shows the final panel of the wizard. Select the **Launch Attribute Editor** option and click on the **Finish** button to invoke the attribute editor.
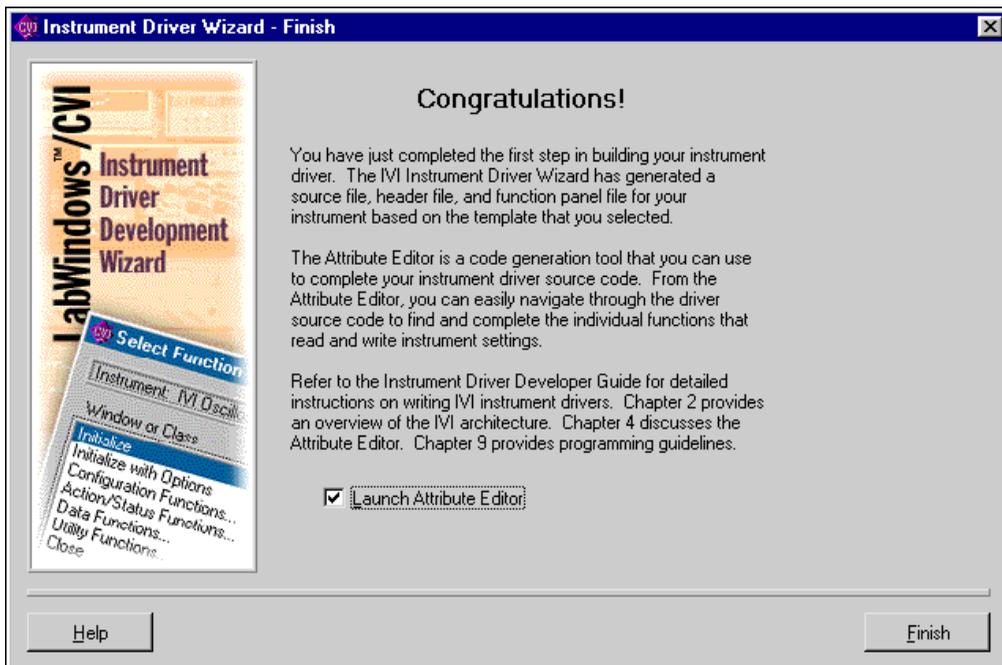
Figure 9-11. Finish Panel

Example 2—Editing the Instrument Driver Attributes

As a final step, the Instrument Driver Development Wizard allows you to launch the attribute editor. However, you can launch the attribute editor at any time by selecting **Tools»Edit Instrument Attributes**. Figure 9-12 shows all the attributes that the Instrument Driver Development Wizard created for the Fluke 45 instrument driver.
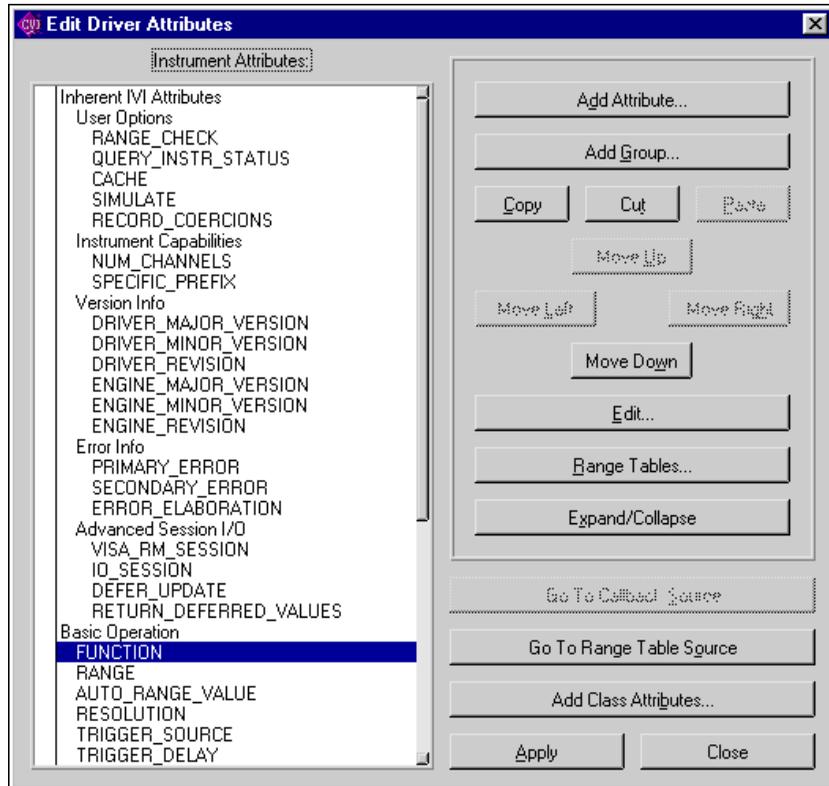
Figure 9-12. Edit Driver Attributes Dialog Box

The Fluke 45 driver you create 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. They also include 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 complete the following steps:

1. Customize each attribute's example implementation for the Fluke 45.

2. Delete the attributes that the Fluke 45 does not use.

The following section shows how to customize the measurement function attribute and delete the attributes that the Fluke 45 does not use.

Customizing the Measurement Function Attribute

From the Edit Driver Attributes dialog box, select the FUNCTION attribute and click on the **Edit** button. The Edit Attribute dialog box appears as shown in Figure 9-13.

Notice that the Instrument Driver Development Wizard has already filled in the attribute information.
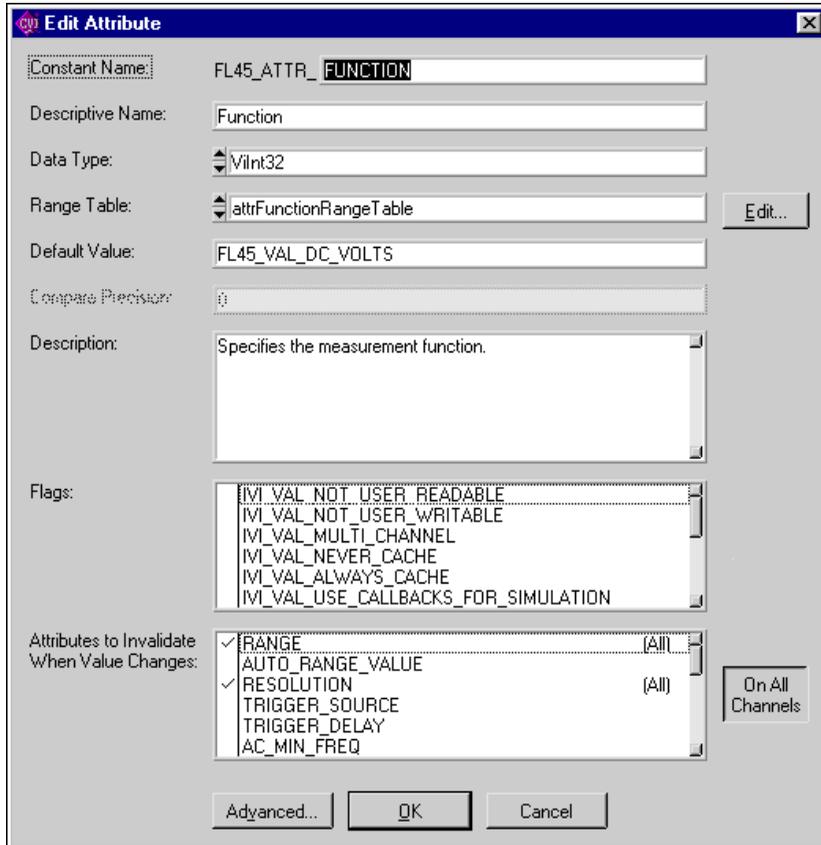
Figure 9-13. Edit Attribute Dialog Box

Click on the **Edit** button to edit the range table for the attribute. The Edit Range Table dialog box is shown in Figure 9-14.
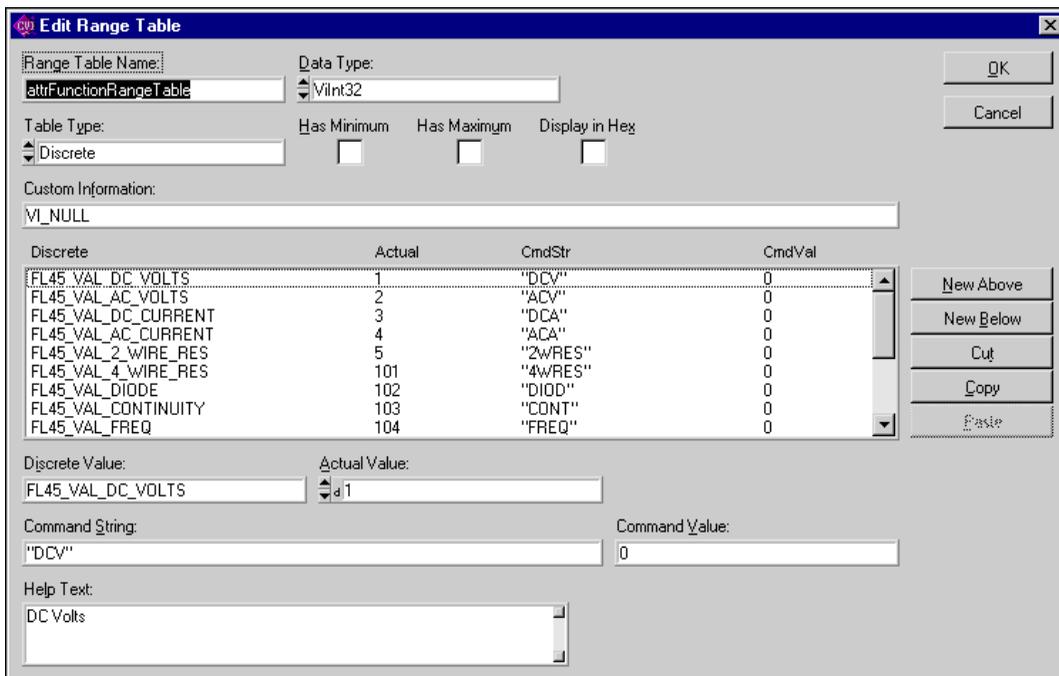
Figure 9-14. Edit Range Table Dialog Box

Complete the following steps to modify the range table.

1. Delete the range table entries that contain the following values. These entries correspond to measurement functions that the Fluke 45 does not support.

    ```
    FL45_VAL_4_WIRE_RES
    FL45_VAL_PERIOD
    FL45_VAL_TEMP_C
    FL45_VAL_TEMP_F
    FL45_VAL_SIEMENS
    FL45_VAL_COULOMBS
    ```

2. Change the contents of the **Command String** control for the remaining entries as follows:

    ```
    FL45_VAL_DC_VOLTS           "VDC"
    FL45_VAL_AC_VOLTS           "VAC"
    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 9-15.
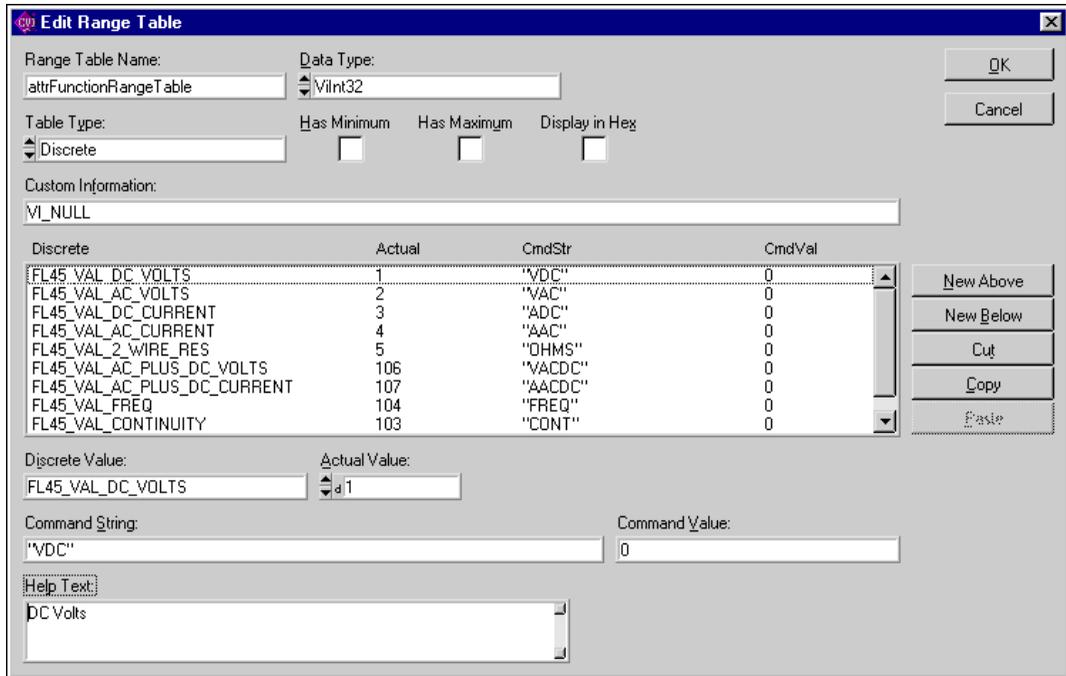
Figure 9-15. Edit Range Table Dialog Box With Modifications

3.   Click on the **OK** button to return to the Edit Attribute dialog box. Click on the **OK** button to return to the Edit Driver Attributes dialog box. Click on the **Apply** button to commit the changes to the range table. Click on the **Close** button on 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.

Complete the following steps to delete the unused measurement function values that the `fl45.h` header file defines.

1. Open the file `fl45.h`.

2. 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_TEMP_C        IVIDMM_VAL_TEMP_C
#define FL45_VAL_TEMP_F        IVIDMM_VAL_TEMP_F
#define FL45_VAL_SIEMENS       IVIDMM_VAL_SIEMENS
#define FL45_VAL_COULOMBS      IVIDMM_VAL_COULOMBS
```

Modifying the Write and Read Callbacks
for the Measurement Function Attribute

Select **Tools»Edit Instrument Attributes** to return to the Edit Driver Attributes dialog box. Select the `FUNCTION` attribute and click on the **Expand/Collapse** button. Select the **Write Callback** for the `FUNCTION` attribute and click on the **Go To Callback Source** button.

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:

1. Uses the range table to look up the instrument-specific command that corresponds to the measurement function the callback receives in the **value** parameter.

2. Writes the command string to the DMM.

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;
}
```

Select **Tools»Edit Instrument Attributes** to return to the Edit Driver Attributes dialog box. Select the **Read Callback** for the FUNCTION attribute and click on the **Go To Callback Source** button.

The read callback for the FUNCTION attribute queries the instrument for the present measurement function setting. The callback performs the following operations:

1. Sends the FUNC1? query to the DMM. This command instructs the Fluke 45 to return the measurement function it is currently using.

2. Reads the response from the instrument.

3. 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. Select **Tools»Edit Instrument Attributes** to invoke the attribute editor.

Complete the following steps to delete the attributes that the Fluke 45 does not use, perform the following:

1. For each of the following attributes, select the attribute and click on the **Cut** button.

   SAMPLE_COUNT
   SAMPLE_TRIGGER
   SAMPLE_INTERVAL
   TRIGGER_COUNT
   TRIGGER_SLOPE
   MEAS_COMPLETE_DEST
   AUTO_ZERO
   POWERLINE_FREQ

2. Select the **Advanced Triggering** group and click on the **Cut** button.

3. Click on the **Apply** button to commit 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 9-16.
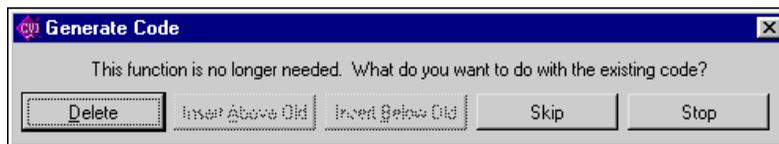
Figure 9-16. Generate Code Dialog Box

4. Click on the **Delete** button for each callback.

5. You also must delete the range tables that correspond to the attributes. From the Edit
    Driver Attributes dialog box, click on the **Range Tables** button. The Range Tables dialog
    box appears as shown in Figure 9-17.
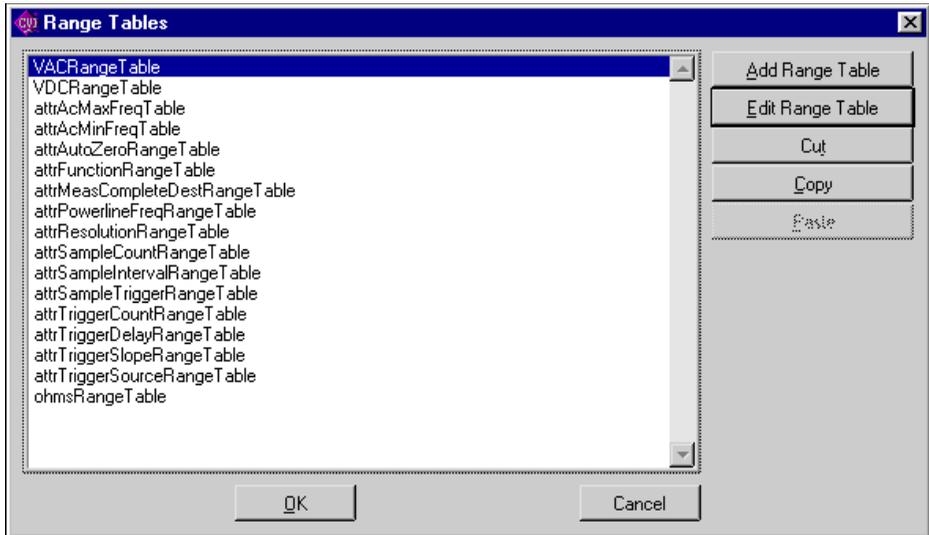
Figure 9-17. Range Tables Dialog Box

6. Delete each of the following range tables by selecting it in the list box and clicking on
    the **Cut** button.

```
attrAutoZeroRangeTable
attrMeasCompleteDestRangeTable
attrPowerlineFreqRangeTable
attrSampleCountRangeTable
attrSampleIntervalRangeTable
attrSampleTriggerRangeTable
attrTriggerCountRangeTable
attrTriggerSlopeRangeTable
```

The Range Tables dialog box appears as shown in Figure 9-18.
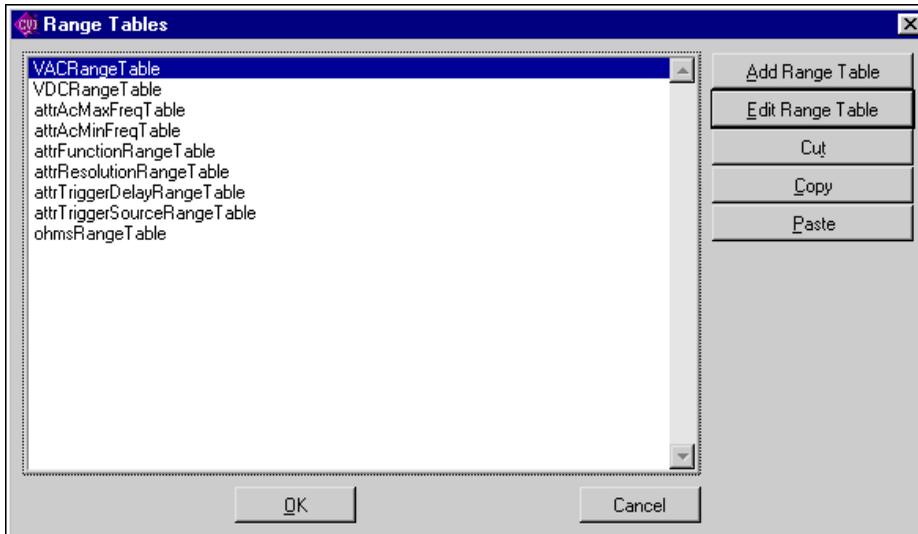
Figure 9-18. Range Tables Dialog Box with Modifications

7.   Click on the **OK** button to return to the Edit Driver Attributes dialog box.

8.   Click on the **Apply** button to commit the range tables changes in the instrument driver files.

9.   Click on the **Close** button on the Edit Driver Attributes dialog box to close the attribute editor.

You also must delete the values that the fl45.h header file defines for the range tables.

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_OFFIVIDMM_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

Select **File»Open** to open the fl45.fp function panel file. The function tree appears as shown in Figure 9-19.
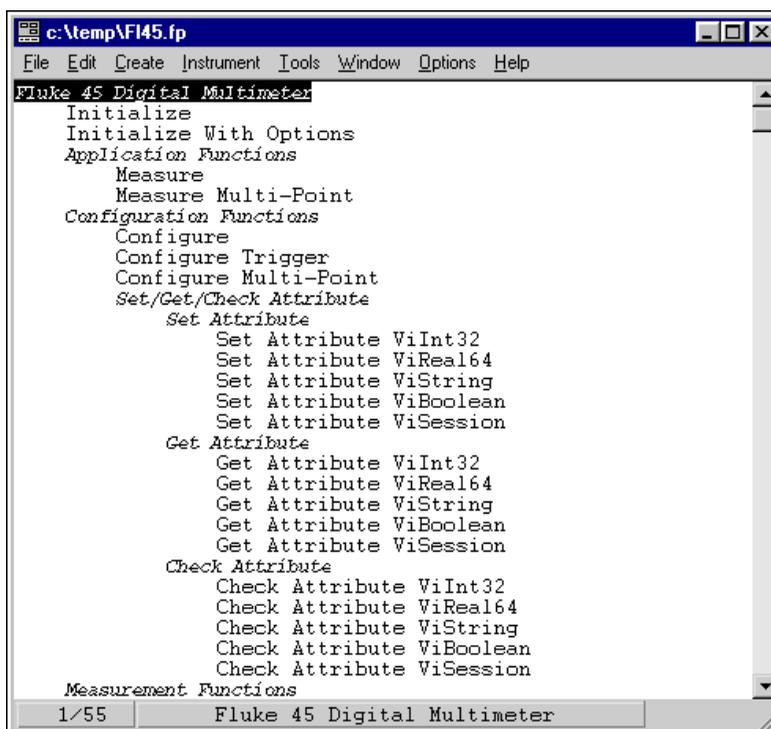
Figure 9-19. Fluke 45 Function Tree

Scroll down to the Fetch function. Right-click on the Fetch function to display the context menu as shown in Figure 9-20.
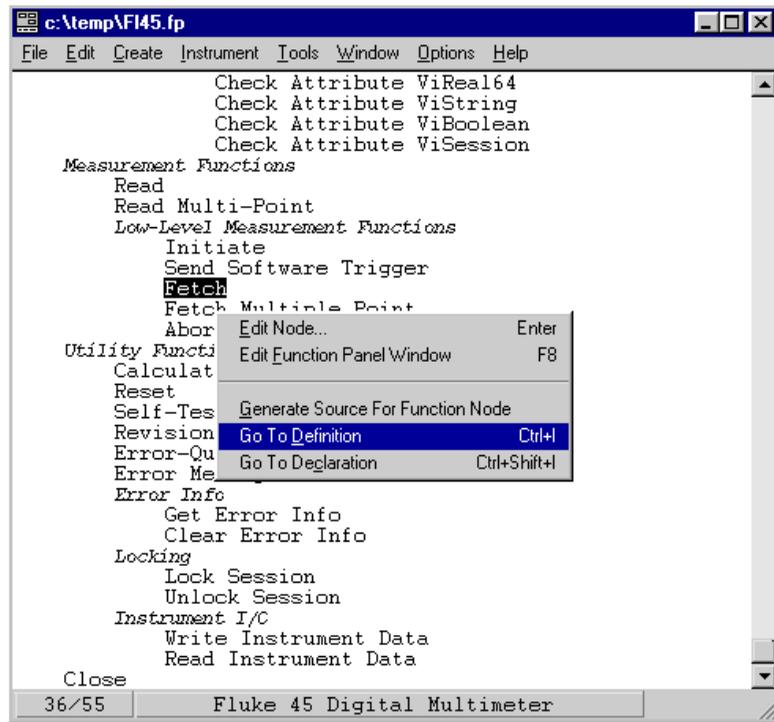
Figure 9-20. Function Tree Editor Context Menu

Choose the **Go To Definition** command 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. 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

```
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?;"));
        viCheckErr (viScanf (io, "%lf", &reading));
            /* 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 multi-point operation, as shown in Table 9-1.

Table 9-1. Multi-Point Operations

Function	Function Panel Name
Fl45_MeasureMultiPoint	Measure Multi-Point
Fl45_ReadMultiPoint	Read Multi-Point
Fl45_FetchMultiPoint	Fetch Multi-Point
Fl45_ConfigureMultiPoint	Configure Multi-Point

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. Edit the function tree for the `fl45.fp` function panel file.

2. Right-click on the Measure Multi-Point function to display the context menu. Select the **Go To Declaration** command to go to the function's declaration in the `fl45.h` file.

3. Delete the declaration.

4. Right-click in the source window and select the **Edit Function Tree** command from the context menu to return to the Function Tree Editor.

5. Right-click on the **Measure Multi-Point** function and select the **Go To Definition** command from the context menu to go to the function's definition in the `fl45.c` file.

6. Delete the entire function body.

7. Right-click in the source window and select the **Edit Function Tree** command from the context menu to return to the Function Tree Editor.

8. Select the **Measure Multi-Point** function. Select **Tools»Cut** 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. This example adds two attributes and a high-level function that configures the hold capability of the Fluke 45. The hold feature enables the Fluke 45 to take a new measurement only when it detects a stable input signal. This example adds the following:

- 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.

Adding the Hold Enable Attribute

Select **Tools»Edit Instrument Attributes** to launch the attribute editor. Select the Advanced Triggering group. Click on the **Add Group** button to create a group for the new attributes. Enter the following information in the Edit Group dialog box.

- Enter `Hold Modifier Attributes` in the **Name** control.

- 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.

After you enter this information, the dialog box appears as shown in Figure 9-21.
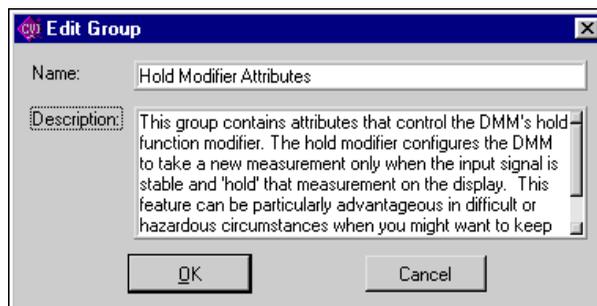
Figure 9-21. Edit Group Dialog Box

Click on the **OK** button to return to the Edit Driver Attributes dialog box. Select the line below the Hold Modifier Attributes class. Click on the **Add Attribute** button to create a new attribute. Enter the following information in the Edit Attribute dialog box:

- Enter HOLD_ENABLE in the **Constant Name** control.

- Enter Hold Enable in the **Descriptive Name** control.

- Select ViBoolean from the **Data Type** ring control.

- Enter VI_FALSE in the **Default Value** control.

- 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.
  ```

After you enter this information, the Edit Attribute dialog box appears as shown in Figure 9-22.
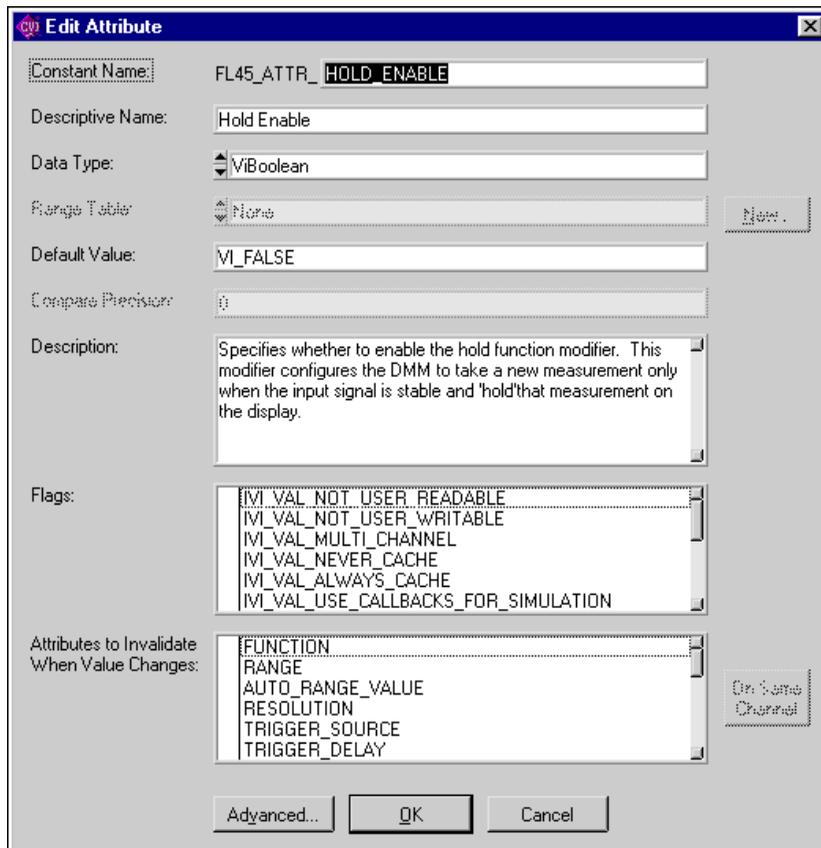
Figure 9-22. Hold Enable Edit Attribute Dialog Box

Click on the **OK** button to return to the Edit Driver Attributes dialog box. Select the HOLD_ENABLE attribute and click on the **Expand/Collapse** button. Click in the left margin of the list box next to the Read Callback and Write Callback entries to create read and write callbacks for the attribute. Click on the **Apply** button to commit the changes.

Select the Write Callback for the HOLD_ENABLE attribute and click on the **Go To Callback Source** button. 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;
}
```

Select **Tools»Edit Instrument Attributes** to return to the Edit Driver Attributes dialog box. Select the Read Callback for the HOLD_ENABLE attribute and click on the **Go To Callback Source** button.

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

Select **Tools»Edit Instrument Attributes** to launch the attribute editor. Select the line below the HOLD_ENABLE function and click on the **Add Attribute** button to create a new attribute. Enter the following information in the Edit Attribute dialog box:

• Enter HOLD_THRESHOLD in the **Constant Name** control.

• Enter Hold Threshold in the **Descriptive Name** control.

• Select ViInt32 from the **Data Type** ring control.

• Click on the **New** button to create a range table for the attribute.

Enter the following information for the range table:

- Enter `holdThresholdRangeTable` in the **Range Table Name** control.

- Select `ViInt32` from the **Data Type** ring control.

- Select `Discrete` from the **Table Type** ring control.

- Deselect the **Has Minimum** and **Has Maximum** controls.

- Enter the information in Table 9-2 for the range table entries.

Table 9-2. 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>.

After you enter this information, the Edit Range Table dialog box appears as shown in Figure 9-23.

Figure 9-23. Hold Threshold Edit Range Table Dialog Box

Click on the **OK** button to return to the Edit Attribute dialog box. Complete the dialog box with the following information:

- Enter FL45_VAL_HOLD_STABLE in the **Default Value** control.

- 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.

After you enter this information, the Edit Attribute dialog box appears as shown in Figure 9-24.
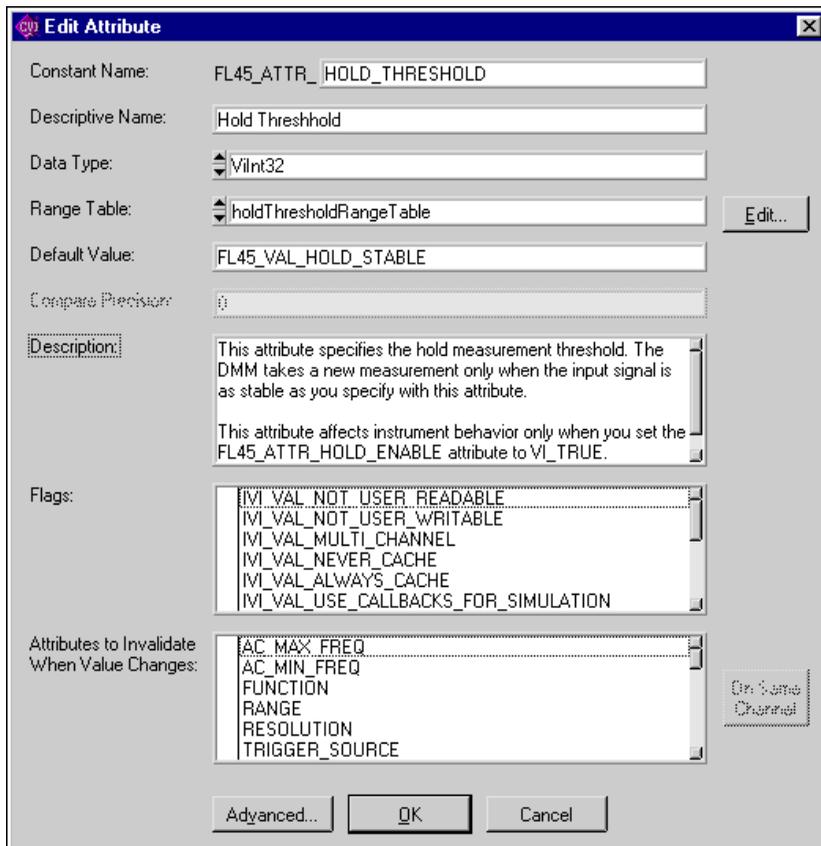
Figure 9-24. Hold Threshold Edit Attribute Dialog Box

Click on the **OK** button to return to the Edit Driver Attributes dialog box. Select the HOLD_THRESHOLD attribute and click on the **Expand/Collapse** button. Click in the left of the list box next to the Read Callback and Write Callback entries to create read and write callbacks for the attribute. Click on the **Apply** button to commit the changes.

Select the Write Callback for the HOLD_THRESHOLD attribute and click on the **Go To Callback Source** button. 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;
}
```

Choose the **Edit Instrument Attributes** command from the **Tools** menu to return to the Edit
Driver Attributes dialog box. Select the Read Callback for the HOLD_THRESHOLD attribute
and click on the **Go To Callback Source** button.

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

Select **File»Open** to open the f145.fp function panel file. Select the **Configure Trigger**
function. Select **Create»Function Panel Window** and enter the following information:

- Enter Configure Hold Modifier in the **Name** control.
- Enter ConfigureHold in the Function **Name** control.

After you enter this information, the Edit Function Panel Window Node dialog box appears as shown in Figure 9-25.
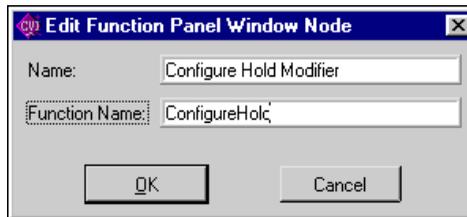
Figure 9-25. Edit Function Panel Window Node Dialog Box

Click on the **OK** button to return to the Function Tree Editor.

Complete the following steps to edit the function panel.

1. Select the **Configure Hold Modifier** function and select **Edit»Edit Function Panel Window**.

2. Choose the **Function Help** command from the **Edit** menu.

3. 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.
   ```

4. Select **File»Close** in the Help Editor.

Complete the following steps to add an Instrument Handle control to the function panel.

1. Press <Ctrl-Page Up> to display the Configure Trigger function panel.

2. Select the **Instrument Handle** control.

3. Select **Edit»Copy Controls**.

4. Press <Ctrl-Page Down> to display the Configure Hold Modifier function panel.

5. Select **Edit»Paste** to place a copy of the **Instrument Handle** control on the Configure Hold Modifier panel.

6. Position the **Instrument Handle** control in the lower left corner of the panel.

Complete the following steps to add a control to specify whether to enable the Hold Modifier.

1. Select **Create»Binary**.

2. Complete the Edit Binary Control and Edit On/Off Settings dialog boxes as shown in Figures 9-26 and 9-27.
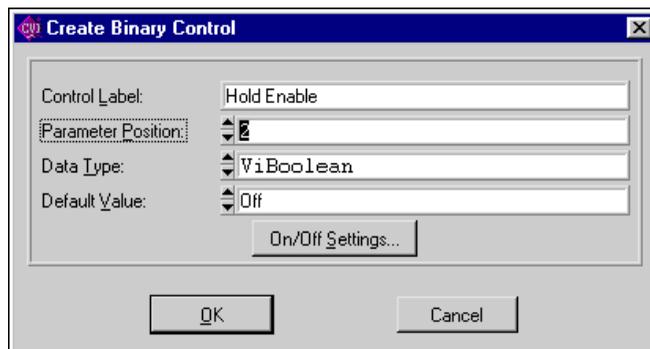
Figure 9-26. Create Binary Control Dialog Box
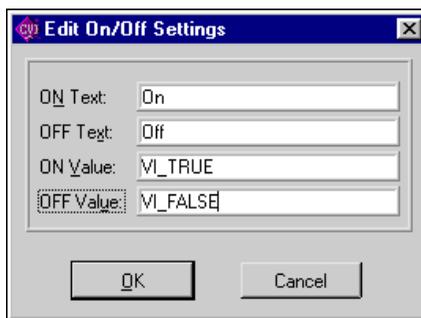
Figure 9-27. Edit On/Off Setting Dialog Box

3. Click on the **OK** button twice to return to the function panel window. Position the **Hold Enable** control in the upper left portion of the panel.

Complete the following steps to add help to the **Hold Enable** control.

1. Select the **Hold Enable** control and select **Edit»Control Help**.

2. 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
```

3. Select **File»Close** to return to the function panel window.

Complete the following steps to add a control to specify the hold threshold.

1. Select **Create»Slide**.
2. Complete the Edit Slide Control dialog box as shown in Figure 9-28.
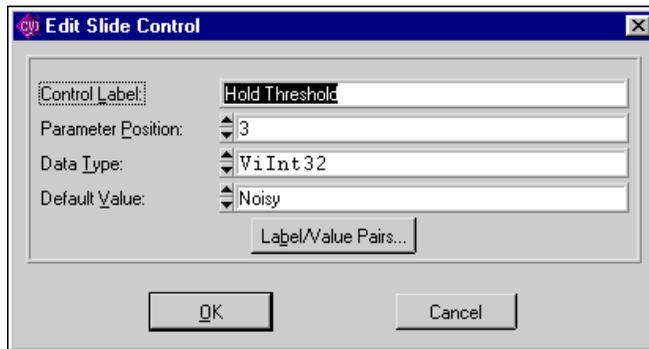
Figure 9-28. Edit Slide Control Dialog Box

3. Click on the **Label/Value Pairs** button and complete the Edit Label/Value Pairs dialog box as shown in Figure 9-29.
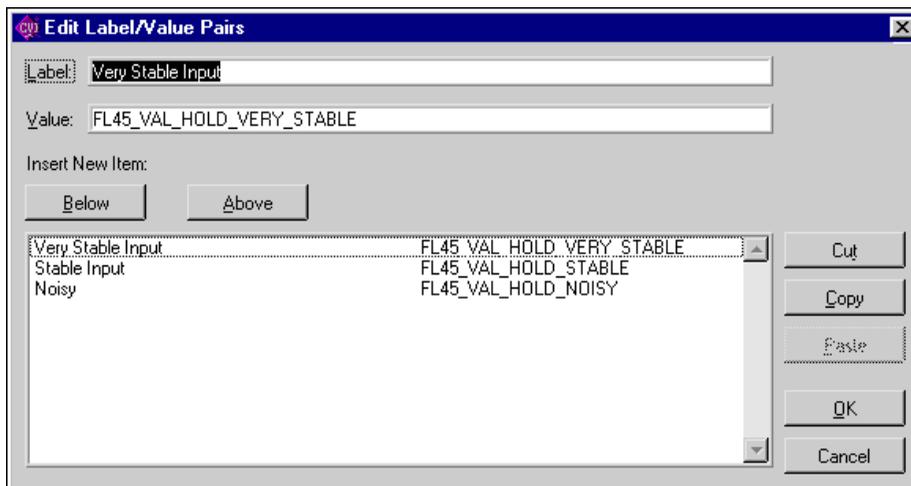
Figure 9-29. Edit Label/Value Pairs Dialog Box

4. Click on the **OK** button twice to return to the Function Panel Editor. Position the **Hold Threshold** control in the upper right portion of the panel.

Complete the following steps to add help to the **Hold Threshold** control.

1. Select the **Hold Threshold** control and select **Edit»Control Help** .

2. 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
    ```

3. Select **File»Close** in the Help editor.

Complete the following steps to add a return control to indicate the status of the function.

1. Press <Ctrl-Page Up> to display the Configure Trigger function panel.

2. Select the **Status** control.

3. Select **Edit»Copy Controls**.

4. Press <Ctrl-Page Down> to display the Configure Hold Modifier function panel.

5. Select **Edit»Paste** to place a copy of the **Status** control on the Configure Hold Modifier panel.

6. Position the **Status** control in the lower right corner of the panel.

Complete the following steps to add help to the **Status** control.

1. Select the **Status** return value and select **Edit»Control Help**.

2. Modify the text in the Help Editor dialog box so that it appears as follows:

```
Reports the status of this operation.

To obtain a text description of the status code, or if the returned
code is not listed below, call the Fl45_error_message function. To
obtain additional information concerning the error condition, use
the Fl45_GetErrorInfo and Fl45_ClearErrorInfo functions.

Status Codes:

Status      Description
-------------------------------------------------
      0     No error (the call was successful).
BFFA0001    Instrument error. Call Fl45_error_query.
BFFA000C    Invalid attribute.
BFFA000D    Attribute is not writable.
BFFA000F    Invalid parameter.
BFFA0010    Invalid value.
BFFA0012    Attribute not supported.
BFFA0013    Value not supported.
BFFA0014    Invalid type.
BFFA0015    Types do not match.
BFFA0016    Attribute already has a value waiting to be updated.
BFFA0018    Not a valid configuration.
BFFA0019    Requested item does not exist or value not available.
BFFA001A    Requested attribute value not known.
BFFA001B    No range table.
BFFA001C    Range table is invalid.
BFFA001F    No channel table has been built for the session.
BFFA0020    Channel name specified is not valid.
BFFA0044    Channel name required.
BFFA0045    Channel name not allowed.
```

```
BFFA0046   Attribute not valid for channel.
BFFA0047   Attribute must be channel based.
BFFC0003   Parameter 3 (Hold ThresHold) out of range.
BFFF0000   Miscellaneous or system error occurred.
BFFF000E   Invalid session handle.
BFFF0015   Timeout occurred before operation could complete.
BFFF0034   Violation of raw write protocol occurred.
BFFF0035   Violation of raw read protocol occurred.
BFFF0036   Device reported an output protocol error.
BFFF0037   Device reported an input protocol error.
BFFF0038   Bus error occurred during transfer.
BFFF003A   Invalid setup (attributes are not consistent).
BFFF005F   No listeners condition was detected.
BFFF0060   This interface is not the controller in charge.
BFFF0067   Operation is not supported on this session.
```

3. Select **File»Close** in the help editor.

The Configure Hold Modifier function panel window now appears as shown in Figure 9-30.
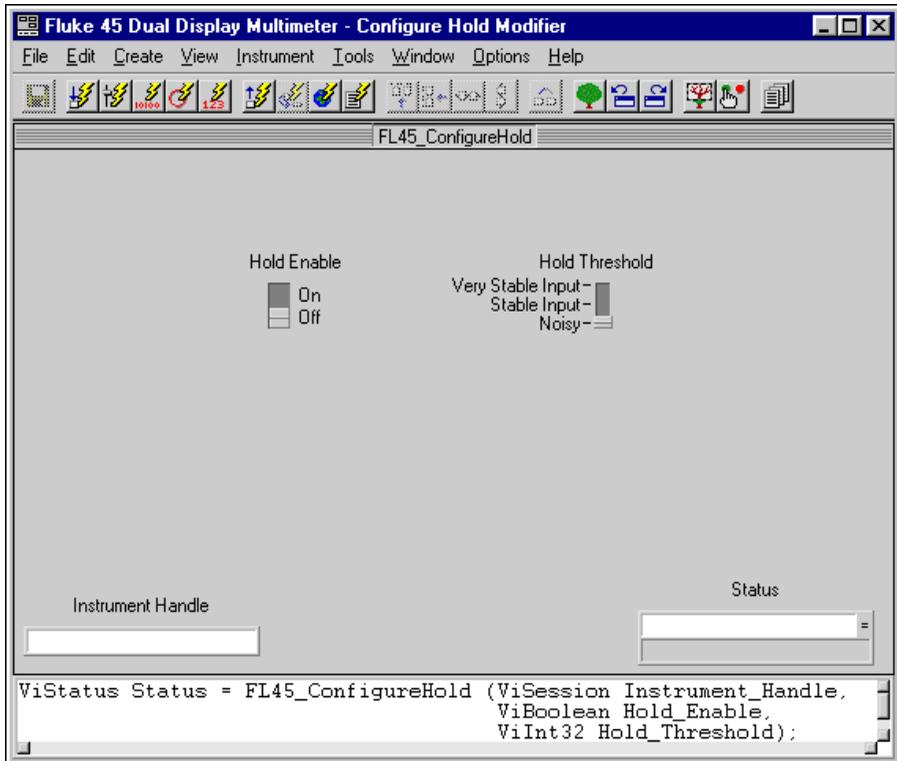
Figure 9-30. Configure Hold Function Panel Window

Creating the Configure Hold Function Body

Complete the following steps to create the Configure Hold Function Body.

1. Select **File»Open** to open the fl45.fp function panel file.

2. Right-click on the Configure Hold 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 on the Configure Hold function and choose the **Go To Definition** command 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.

```c
/****************************************************************
 * 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;

    viCheckParm (Ivi_SetAttributeViBoolean (vi, VI_NULL,
                          FL45_ATTR_HOLD_ENABLE, 0, holdEnable), 2,
                          "Hold Enable");

    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

Open the fl45.fp file and edit the function tree. Select the **Options»Generate
Documentation**. The dialog box shown in Figure 9-31 lets you choose the programming
language for which you wish to generate documentation.
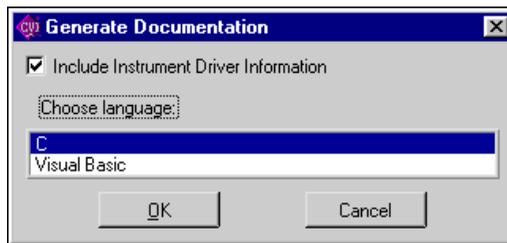
Figure 9-31. Generate Documentation Dialog

Select the language you want and click on the **OK** button. The instrument driver documentation appears as shown in Figure 9-32.
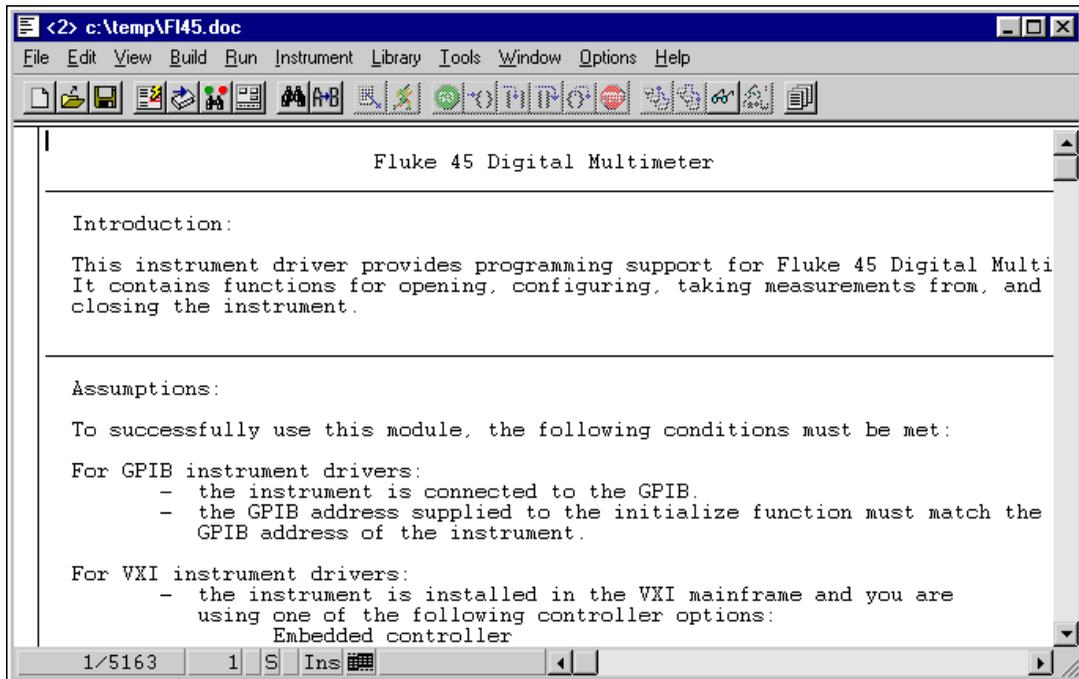
Figure 9-32. Fluke 45 Instrument Driver Documentation Window

Select **File»Save** and save the file as fl45.doc.

Creating Windows Help for the Driver

Open the file fl45.fp and edit the function tree. Select **Options»Generate Windows Help**. Select the programming language for which you want to generate the Windows help from the Generate Windows Help dialog box. Click on the **OK** button. A message appears as shown in Figure 9-33.

Figure 9-33. LabWindows/CVI Message Dialog Box

Run `cvi/sdk/bin/hcrtf -x fl45.hpj` to generate the help file. The `hcrtf.exe` program installs with LabWindows/CVI.

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. To invoke the wizard, select **Tools»Create IVI Instrument Driver**. Click on the **Next** button on the welcome panel to begin.
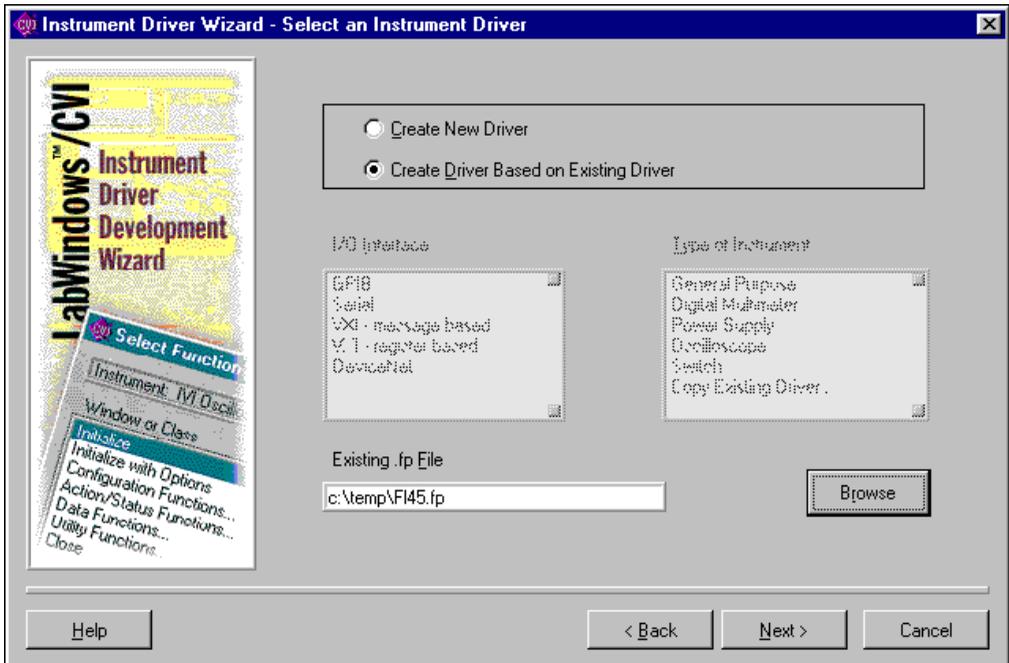
Figure 9-34. Select an Instrument Driver Panel

Enter the following information in the wizard panel.

- Select the **Create Driver Based on Existing Driver** option.
- Click on the **Browse** button 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 enter this information, the panel appears as shown in Figure 9-34. Click on the **Next** button to continue.
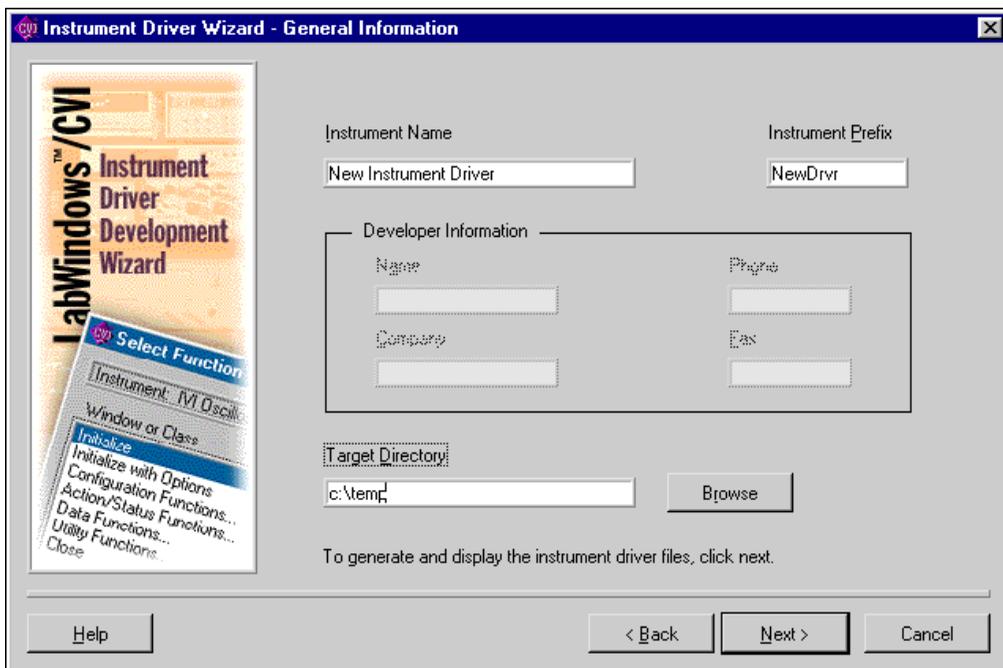
Figure 9-35. General Information Panel

Enter the name, prefix, and target directory for the new driver. After you enter this information, the panel appears as shown in Figure 9-35. Click on the **Next** button 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 Examples 2 and 3 earlier in this chapter.

*   Adding new attributes and functions. Refer to Example 4 earlier in this chapter.

*   Creating the instrument driver documentation. Refer to Example 5 earlier in this chapter.

A

Technical Support Resources

This appendix describes the comprehensive resources available to you in the Technical Support section of the National Instruments Web site and provides technical support telephone numbers for you to use if you have trouble connecting to our Web site or if you do not have internet access.

NI Web Support

To provide you with immediate answers and solutions 24 hours a day, 365 days a year, National Instruments maintains extensive online technical support resources. They are available to you at no cost, are updated daily, and can be found in the Technical Support section of our Web site at `www.ni.com/support`

Online Problem-Solving and Diagnostic Resources

- **KnowledgeBase**—A searchable database containing thousands of frequently asked questions (FAQs) and their corresponding answers or solutions, including special sections devoted to our newest products. The database is updated daily in response to new customer experiences and feedback.

- **Troubleshooting Wizards**—Step-by-step guides lead you through common problems and answer questions about our entire product line. Wizards include screen shots that illustrate the steps being described and provide detailed information ranging from simple getting started instructions to advanced topics.

- **Product Manuals**—A comprehensive, searchable library of the latest editions of National Instruments hardware and software product manuals.

- **Hardware Reference Database**—A searchable database containing brief hardware descriptions, mechanical drawings, and helpful images of jumper settings and connector pinouts.

- **Application Notes**—A library with more than 100 short papers addressing specific topics such as creating and calling DLLs, developing your own instrument driver software, and porting applications between platforms and operating systems.

Software-Related Resources

- **Instrument Driver Network**—A library with hundreds of instrument drivers for control of standalone instruments via GPIB, VXI, or serial interfaces. You also can submit a request for a particular instrument driver if it does not already appear in the library.

- **Example Programs Database**—A database with numerous, non-shipping example programs for National Instruments programming environments. You can use them to complement the example programs that are already included with National Instruments products.

- **Software Library**—A library with updates and patches to application software, links to the latest versions of driver software for National Instruments hardware products, and utility routines.

Worldwide Support

National Instruments has offices located around the globe. Many branch offices maintain a Web site to provide information on local services. You can access these Web sites from `www.ni.com/worldwide`

If you have trouble connecting to our Web site, please contact your local National Instruments office or the source from which you purchased your National Instruments product(s) to obtain support.

For telephone support in the United States, dial 512 795 8248. For telephone support outside the United States, contact your local branch office:

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 (Ontario) 905 785 0085, Canada (Québec) 514 694 8521, China 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, Mexico (D.F.) 5 280 7625, Mexico (Monterrey) 8 357 7695, Netherlands 0348 433466, Norway 32 27 73 00, Poland 48 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 2377 1200, United Kingdom 01635 523545

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, 1-11

Add Attribute button, Edit Driver Attributes dialog box, 4-4

Add Class Attributes button, Edit Driver Attributes dialog box, 4-5

Add Group button, Edit Driver Attributes dialog box, 4-5

Align Horizontal Centers command, Edit menu, 6-6

Alignment command, Edit menu, 6-5

Any Array data type, 3-13

Any Type data type, 3-13 to 3-14

application functions
    initialization and close functions not called by (note), 1-12
    purpose and use, 1-12

Apply command, Edit Driver Attributes dialog box, 4-4 to 4-6

architecture. *See* instrument driver architecture.

array data types, user-defined, 3-15

ASCII text file, as component of instrument drivers, 1-3

Attach and Edit Source command, Edit Instrument dialog box, 5-10

Attribute Editor, 4-1 to 4-13
    adding and editing
        instrument attributes, 4-7 to 4-9
        range tables, 4-10 to 4-13
    Edit Attribute dialog box, 4-7 to 4-9
        Advanced dialog box (figure), 4-9
        entering information, 4-7 to 4-9
        illustration, 4-7
    Edit Driver Attributes dialog box, 4-3 to 4-6
        command buttons, 4-4 to 4-6
        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, 4-10 to 4-13
        Edit Range Table dialog box, 4-11 to 4-13
        illustration, 4-10
    requirements for using, 4-1 to 4-2
    reviewing files generated by Instrument Driver Development Wizard, 3-9

attribute functions
    callback functions. *See* callback functions.
    purpose and use, 1-12

attributes. *See also* state-caching, IVI.
    accessing with user-callable functions, 8-27
    adding new attributes
      (example), 9-29 to 9-44
        Configure Hold function panel, 9-36 to 9-43
        creating Configure Hold Function Body, 9-43 to 9-44
        Hold Enable attribute, 9-29 to 9-32
        Hold Threshold attribute, 9-32 to 9-36
    class attributes
        constant name for ID, 2-10
        definition, 2-7
    comparison precision, 2-18 to 2-19
    creating and declaring, 2-9 to 2-19
    customizing Wizard-generated attributes, 8-3 to 8-5
      adding new attributes, 8-4 to 8-5
      deleting attributes, 8-4
      general modifications, 8-5
      modifying existing attributes, 8-3