LabWindows/CVI Instrument Driver Developers Guide

NATIONAL INSTRUMENTS™
The Software is the Instrument™

February 1998 Edition
Part Number 320684D-01

Internet Support
E-mail: `support@natinst.com`
FTP Site: `ftp.natinst.com`
Web Address: `www.natinst.com`

Bulletin Board Support
BBS United States: 512 794 5422
BBS United Kingdom: 01635 551422
BBS France: 01 48 65 15 59

Fax-on-Demand Support
512 418 1111

Telephone Support (USA)
Tel: 512 795 8248
Fax: 512 794 5678

International Offices
Australia 03 9879 5166, Austria 0662 45 79 90 0, Belgium 02 757 00 20, Brazil 011 288 3336, Canada (Ontario) 905 785 0085, Canada (Québec) 514 694 8521, Denmark 45 76 26 00, Finland 09 725 725 11, France 01 48 14 24 24, Germany 089 741 31 30, Hong Kong 2645 3186, Israel 03 6120092, Italy 02 413091, Japan 03 5472 2970, Korea 02 596 7456, Mexico 5 520 2635, Netherlands 0348 433466, Norway 32 84 84 00, Singapore 2265886, Spain 91 640 0085, Sweden 08 730 49 70, Switzerland 056 200 51 51, Taiwan 02 377 1200, United Kingdom 01635 523545

National Instruments Corporate Headquarters
6504 Bridge Point Parkway Austin, Texas 78730-5039 USA Tel: 512 794 0100

© Copyright 1994, 1998 National Instruments Corporation. All rights reserved.

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 manual 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™, `natinst.com`™, National Instruments™, the National Instruments logo, and The Software is the Instrument™ are trademarks of National Instruments Corporation.

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

WARNING REGARDING MEDICAL AND CLINICAL USE OF NATIONAL INSTRUMENTS PRODUCTS

National Instruments products are not designed with components and testing intended to ensure a level of reliability suitable for use in treatment and diagnosis of humans. Applications of National Instruments products involving medical or clinical treatment can create a potential for accidental injury caused by product failure, or by errors on the part of the user or application designer. Any use or application of National Instruments products for or involving medical or clinical treatment must be performed by properly trained and qualified medical personnel, and all traditional medical safeguards, equipment, and procedures that are appropriate in the particular situation to prevent serious injury or death should always continue to be used when National Instruments products are being used. National Instruments products are NOT intended to be a substitute for any form of established process, procedure, or equipment used to monitor or safeguard human health and safety in medical or clinical treatment.

Contents

About This Manual

Organization of This Manual xxiii
Conventions Used in This Manual xxiv
The LabWindows/CVI Documentation Set xxv
Customer Communication xxvi

Chapter 1
Instrument Driver Overview

What Is an Instrument Driver? 1-1
Historical Evolution of Instrument Drivers 1-1
About Instrument Drivers 1-2
How Users Operate the Instrument Driver 1-3
Instrument Driver Architecture 1-4
    Instrument Driver External Interface Model 1-4
        Functional Body 1-6
        IVI Engine 1-6
        VISA I/O Interface 1-6
        Subroutine Interface 1-6
        Programmatic Developer Interface 1-7
        Interactive Developer Interface 1-7
    Instrument Driver Internal Design Model 1-8
        Component Functions 1-9
        Initialize Functions 1-10
        Configuration Functions 1-10
        Action/Status Functions 1-11
        Data Functions 1-11
        Attribute Functions 1-11
        Utility Functions 1-11
        Close Function 1-12
        Application Functions 1-12
        Callback Functions 1-12

Chapter 2
IVI Architecture Overview

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

Driver Functions and Attribute Model 2-6
Types of Attributes 2-7
Get/Set/Check Functions 2-7
Callbacks 2-8
Creating and Declaring Attributes 2-9
Attribute IDs 2-9
Inherent IVI Attributes 2-9
Class Attributes 2-9
Instrument-Specific Attributes 2-10
Attribute Flags 2-10
Range Tables 2-13
Range Table Structures 2-13
Discrete Range Table Example 2-15
Coerced Range Table Example 2-16
Ranged Range Table Example 2-16
Static and Dynamic Range Tables 2-17
Default Check and Coerce Callbacks 2-17
Comparison Precision 2-18
IVI State-Caching Mechanism 2-19
Initial Instrument State 2-19
Special Cases 2-19
Changing the Value of One Attribute Invalidates Another 2-20
Two Attributes Invalidate Each Other 2-20
Setting or Getting the Values of Two Attributes in One Command 2-20
Instrument Coerces Values 2-21
Enabling and Disabling State-Caching 2-21
Attribute Callback Functions 2-22
Read Callback 2-22
Write Callback 2-23
Check Callback 2-23
Coerce Callback 2-23
Compare Callback 2-25
Range Table Callback 2-25
Session Callback Functions 2-26
Operation Complete Callback 2-26
Check Status Callback 2-26
Instruments without Error Queues 2-27
Buffered I/O Callback 2-27
Channels 2-29
Virtual Channel Names 2-29
Passing Channel Names to IVI Functions 2-30
Coercing and Validating Channel Names 2-30

High-Level Driver Functions 2-30
Range Checking 2-31
Status Checking 2-31
Simulation 2-32
Multithread Safety 2-33
Deferred Updates 2-34
Configuration Entries 2-35
Inherent IVI Attributes 2-36
    Inherent Attribute Reference 2-38
        IVI_ATTR_ATTRIBUTE_CAPABILITIES 2-38
        IVI_ATTR_BUFFERED_IO_CALLBACK 2-38
        IVI_ATTR_CACHE 2-38
        IVI_ATTR_CHECK_STATUS_CALLBACK 2-39
        IVI_ATTR_CLASS_MAJOR_VERSION 2-39
        IVI_ATTR_CLASS_MINOR_VERSION 2-39
        IVI_ATTR_CLASS_PREFIX 2-40
        IVI_ATTR_CLASS_REVISION 2-40
        IVI_ATTR_DEFER_UPDATE 2-40
        IVI_ATTR_DRIVER_MAJOR_VERSION 2-41
        IVI_ATTR_DRIVER_MINOR_VERSION 2-41
        IVI_ATTR_DRIVER_REVISION 2-41
        IVI_ATTR_DRIVER_SETUP 2-41
        IVI_ATTR_ENGINE_MAJOR_VERSION 2-42
        IVI_ATTR_ENGINE_MINOR_VERSION 2-42
        IVI_ATTR_ENGINE_REVISION 2-42
        IVI_ATTR_ERROR_ELABORATION 2-42
        IVI_ATTR_FUNCTION_CAPABILITIES 2-42
        IVI_ATTR_GROUP_CAPABILITIES 2-43
        IVI_ATTR_INTERCHANGE_CHECK 2-43
        IVI_ATTR_IO_SESSION 2-43
        IVI_ATTR_LOGICAL_NAME 2-44
        IVI_ATTR_MODULE_PATHNAME 2-44
        IVI_ATTR_NUM_CHANNELS 2-44
        IVI_ATTR_OPC_CALLBACK 2-45
        IVI_ATTR_PRIMARY_ERROR 2-45
        IVI_ATTR_QUERY_INSTR_STATUS 2-45
        IVI_ATTR_RANGE_CHECK 2-46
        IVI_ATTR_RECORD_COERCIONS 2-46
        IVI_ATTR_RESOURCE_DESCRIPTOR 2-47
        IVI_ATTR_RETURN_DEFERRED_VALUES 2-47
        IVI_ATTR_SECONDARY_ERROR 2-47
        IVI_ATTR_SIMULATE 2-48
        IVI_ATTR_SPECIFIC_PREFIX 2-48

IVI_ATTR_SPY 2-48
IVI_ATTR_SUPPORTS_WR_BUF_OPER_MODE 2-49
IVI_ATTR_UPDATING_VALUES 2-49
IVI_ATTR_USE_SPECIFIC_SIMULATION 2-49
IVI_ATTR_VISA_RM_SESSION 2-50

Chapter 3
Developing an Instrument Driver

General Guidelines 3-1
Writing an Instrument Driver 3-2
Naming the Driver 3-3
How to Use the Instrument Driver Development Wizard 3-3
Selecting an Instrument Driver Template 3-5
Running the Preliminary I/O Tests from the Wizard 3-6
Reviewing the Generated Driver Files 3-6
The Generated Function Panels 3-6
The .sub file 3-7
The Source File 3-8
The Include File 3-9
Extended Functions and Attributes 3-9
Attribute Editor 3-9
Instrument Driver Fundamentals 3-9
Defining the Instrument Functions 3-9
Structuring Functions in an Instrument Driver 3-10
Defining the Hierarchy of Functions 3-11
Defining the Function Parameters 3-11
Data Types 3-11
Predefined Data Types 3-12
Intrinsic C Data Types 3-12
Meta Data Types 3-13
Numeric Array 3-13
Any Array 3-13
Any Type 3-13
Var Args 3-14
User-Defined Data Types 3-14
Creating a User-Defined Data Type 3-14
User-Defined Array Data Types 3-15
VISA Data Types 3-16
Input and Output Parameters 3-17
Return Values 3-17
Required Instrument Driver Functions 3-18
Building the Function Tree 3-19

Building the Function Panels......3-19
Writing the Function Code......3-19
Operating the Driver......3-19
Testing the Instrument Driver......3-20
Documenting the Driver......3-20

Chapter 4
Attribute Editor

Invoking the Attribute Editor......4-1
Requirements for Using the Attribute Editor......4-1
Limitations in Updates to Driver Files......4-2
Edit Driver Attributes Dialog Box......4-3
Instrument Attributes List Box......4-4
Restrictions on Modifications to Inherent and Class Attributes......4-4
Attributes List Box Command Buttons......4-5
Adding and Editing Instrument Attributes......4-7
Adding and Editing Range Tables......4-10

Chapter 5
Function Tree Editor

About the Function Tree and Function Tree Editor......5-1
Function Tree Editor Context Menu......5-2
Function Tree Editor Menu Bar......5-4
File......5-4
Edit......5-5
Find......5-5
Replace......5-5
.FP Auto-Load List......5-6
Create......5-6
Instrument......5-7
Class......5-7
Adding a Class to an Empty Tree or Class......5-7
Inserting a Class into an Existing Tree......5-8
Function Panel Window......5-8
Adding a Function to an Empty Tree or Class......5-8
Inserting a Function into an Existing Tree......5-9
Instrument......5-9
Load......5-9
Unload......5-9
Edit......5-10

Tools 5-11
Window 5-12
Options 5-12
Function Tree Editor Examples 5-15
Example—Multiple Classes in a Function Tree 5-15
Example—Cutting and Pasting Functions and Panels 5-16
Using Existing Function Panels in a New Driver 5-17
Example—Editing Items in the Function Tree 5-17

Chapter 6
Function Panel Editor

Invoking the Function Panel Editor 6-1
Invoking from the Function Tree Editor 6-1
Invoking from a Function Panel 6-1
The Function Panel Editor Menu Bar 6-2
File 6-3
Edit 6-3
Cut Controls 6-4
Copy Controls 6-4
Paste 6-4
Cut Panel 6-4
Copy Panel 6-5
Edit Control 6-5
Change Control Type 6-5
Edit Function 6-5
Alignment 6-5
Align Horizontal Centers 6-5
Distribution 6-6
Distribute Vertical Centers 6-6
Find 6-6
Replace 6-6
Control Help 6-6
Function Help or Window Help 6-7
Create 6-7
Function Panel Window, Function Panel, and Common Control Panel 6-7
Control Types 6-8
Input 6-8
Slide 6-9
Adding a Label and Value to the Slide Control List 6-11
Dialog Box Command Buttons 6-11
Binary 6-12

Ring 6-14
Adding a Label and Value to the Ring Control List 6-15
Dialog Box Command Buttons 6-16
Numeric 6-16
Output 6-19
Return Value 6-20
Global Variable 6-20
Message 6-21
View 6-21
Instrument 6-22
Tools 6-22
Window 6-22
Options 6-23
Data Types 6-23
Toolbar 6-24
Default Panel Size 6-24
Panels Movable 6-24
Toggle Scroll Bars 6-24
Edit Function Tree 6-24
Operate Function Panel 6-24
Moving Controls 6-25
Moving Controls between Function Panels 6-25
Selecting Multiple Controls 6-25
Function Panel Editor Examples 6-25
Example—Creating a Function Window 6-26
Example—Changing Control Type 6-30
Example—Cutting and Pasting Controls 6-32

Chapter 7
Adding Help Information

New Style vs. Old Style Help 7-1
Help Options 7-2
Editing Help Information 7-2
File 7-3
Edit 7-4
Tools 7-4
Window 7-4
Instrument Help 7-4
Function Class Help 7-5
Function Help (New Style Help Only) 7-5
Function Panel Window Help (Old Style Help Only) 7-5
Control Help 7-6

Help Information Examples 7-6
Example—Adding Help Information in the Function Tree Editor 7-7
Example—Adding Help Information in the Function Panel Editor 7-9
Example—Copying and Pasting Help Text 7-9

Chapter 8
Programming Guidelines for Instrument Drivers

Generating Driver Files 8-1
Selecting a Template 8-1
Procedures for Customizing Wizard-Generated Driver Files 8-3
Modifying Existing Attributes and Functions 8-3
Deleting the Attributes and Functions 8-4
Adding New Attributes and Functions 8-4
General Modifications 8-5
Instrument Driver Attributes 8-5
Attribute ID Values 8-5
Attribute Value Definitions 8-6
Simulation 8-7
Data Types 8-7
Callbacks 8-8
Read and Coerce Callbacks for ViString Attributes 8-8
Write Callbacks 8-8
Reading Strings From the Instrument 8-9
Using viScanf 8-9
Using viRead 8-9
Range Table Callbacks 8-10
Range Tables 8-11
Attribute Examples 8-11
Attributes That Represent Discrete Settings 8-11
Attributes that Represent a Continuous Range 8-13
Attributes that Represent a Continuous Range with Discrete Settings 8-15
Attributes with a Changing Valid Range 8-16
Using Multiple Static Range Tables 8-17
Using Dynamic Range Tables 8-20
Check, Coerce, and Compare Callbacks 8-21
User-Callable Functions 8-22
Instrument Driver Function Structure 8-22
Locking/Unlocking the Session 8-25
Parameter Checking 8-25
Accessing Attributes 8-26
Performing Direct Instrument I/O 8-26

Simulating Output Parameters 8-27
Checking the Instrument Status 8-27
Functions That Only Set Attributes 8-28
Initialization Functions 8-29
Channel Strings 8-30
Close Functions 8-30
Developing Portable Instrument Drivers 8-30
Instrument Driver Data Types 8-30
Declaring Instrument Driver Functions 8-31
Using Scan and Fmt Functions 8-32
Error-Reporting Guidelines 8-33
General Programming Guidelines 8-35
Function Panels 8-36
Function Tree Hierarchy 8-36
Documentation Guidelines 8-37
Online Help 8-38
The .doc File 8-42
Programming Guidelines for VXI Instruments 8-43
Instrument Driver Checklist 8-43

Chapter 9
Required Instrument Driver Functions

Required Instrument Driver Function Overview 9-1
Required Instrument Driver Function Reference 9-2
Prefix_init 9-3
Prefix_InitWithOptions 9-6
Prefix_IviInit 9-8
Prefix_close 9-10
Prefix_IviClose 9-12
Prefix_reset 9-13
Prefix_self_test 9-15
Prefix_error_query 9-17
Prefix_error_message 9-20
Prefix_revision_query 9-22

Chapter 10
Instrument Driver Examples

Example 1—Creating IVI Instrument Driver Files with the Instrument Driver Development Wizard 10-2
Example 2—Editing the Instrument Driver Attributes 10-13
Customizing the Measurement Function Attribute 10-14
Modifying the Write and Read Callbacks for the Measurement Function Attribute 10-17
Deleting Unused Attributes 10-18
Example 3—Editing High-Level Instrument Driver Functions 10-22
Editing the Fetch Function 10-22
Deleting Functions the Instrument Does Not Use 10-26
Example 4—Adding New Attributes and Functions 10-27
Adding the Hold Enable Attribute 10-27
Adding the Hold Threshold Attribute 10-30
Adding the Configure Hold Function Panel 10-34
Creating the Configure Hold Function Body 10-40
Example 5—Creating Instrument Driver Documentation 10-41
Creating the Instrument Driver .doc file 10-42
Creating Windows Help for the Driver 10-43
Example 6—Modifying an Existing IVI Driver to Work with a New Instrument 10-43

Chapter 11
IVI Library

IVI Library Function Overview 11-1
IVI Library Function Panels 11-1
The IVI Library Function Tree 11-2
Error Reporting 11-8
Error Macros 11-9
checkAlloc(pointer) 11-10
checkErr(status) 11-10
checkWarn(status) 11-10
viCheckAlloc(pointer) 11-10
viCheckErr(status) 11-10
viCheckErrElab(status, elabString) 11-10
viCheckParm(status, parameterPosition, parameterName) 11-11
viCheckWarn(status) 11-11
When to Use the viCheck Macros 11-11
Examples 11-12

IVI Library Function Reference 11-12
Ivi_AddAttributeInvalidation 11-13
Ivi_AddAttributeViAddr 11-15
Ivi_AddAttributeViBoolean 11-18
Ivi_AddAttributeViInt32 11-21
Ivi_AddAttributeViReal64 11-24
Ivi_AddAttributeViSession 11-28
Ivi_AddAttributeViString 11-31
Ivi_AddToChannelTable 11-34
Ivi_Alloc 11-35
Ivi_AttributeIsCached 11-37
Ivi_AttributeUpdateIsPending 11-38
Ivi_BuildChannelTable 11-39
Ivi_CheckAttributeViInt32 11-42
Ivi_CheckAttributeViReal64 11-42
Ivi_CheckAttributeViString 11-42
Ivi_CheckAttributeViBoolean 11-42
Ivi_CheckAttributeViSession 11-42
Ivi_CheckAttributeViAddr 11-42
Ivi_CheckBooleanRange 11-44
Ivi_CheckNumericRange 11-45
Ivi_ClearErrorInfo 11-47
Ivi_ClearInstrSpecificErrorQueue 11-49
Ivi_CoerceBoolean 11-50
Ivi_CoerceChannelName 11-51
Ivi_CompareWithPrecision 11-53
Ivi_DefaultBufferedIOCallback 11-55
Ivi_DefaultCheckCallbackViInt32 11-56
Ivi_DefaultCheckCallbackViReal64 11-58
Ivi_DefaultCoerceCallbackViBoolean 11-60
Ivi_DefaultCoerceCallbackViInt32 11-62
Ivi_DefaultCoerceCallbackViReal64 11-64
Ivi_DefaultCompareCallbackViReal64 11-66
Ivi_DefineClass 11-69
Ivi_DefineDriver 11-71
Ivi_DefineHardware 11-74
Ivi_DefineLogicalName 11-76
Ivi_DefineVInstr 11-78
Ivi_DeleteAttribute 11-82
Ivi_DeleteAttributeInvalidation 11-83
Ivi_DequeueInstrSpecificError 11-84
Ivi_Dispose 11-86
Ivi_DisposeInvalidationList 11-87

Ivi_DisposeLogicalNamesList 11-88
Ivi_Free 11-89
Ivi_FreeAll 11-90
Ivi_GetAttrComparePrecision 11-91
Ivi_GetAttributeFlags 11-92
Ivi_GetAttributeName 11-93
Ivi_GetAttributeType 11-95
Ivi_GetAttributeViInt32 11-96
Ivi_GetAttributeViReal64 11-96
Ivi_GetAttributeViBoolean 11-96
Ivi_GetAttributeViSession 11-96
Ivi_GetAttributeViAddr 11-96
Ivi_GetAttributeViString 11-99
Ivi_GetAttrMinMaxViInt32 11-102
Ivi_GetAttrMinMaxViReal64 11-104
Ivi_GetAttrRangeTable 11-106
Ivi_GetErrorInfo 11-108
Ivi_GetErrorMessage 11-110
Ivi_GetInvalidationList 11-111
Ivi_GetIviIniDir 11-113
Ivi_GetLogicalNamesList 11-114
Ivi_GetNextCoercionInfo 11-116
Ivi_GetNthAttribute 11-118
Ivi_GetNthChannelString 11-119
Ivi_GetNthLogicalName 11-121
Ivi_GetNumAttributes 11-123
Ivi_GetRangeTableNumEntries 11-124
Ivi_GetSpecificDriverStatusDesc 11-125
Ivi_GetStoredRangeTablePtr 11-127
Ivi_GetStringFromTable 11-129
Ivi_GetUserChannelName 11-131
Ivi_GetValueFromTable 11-133
Ivi_GetViInt32EntryFromCmdValue 11-135
Ivi_GetViInt32EntryFromCoercedVal 11-137
Ivi_GetViInt32EntryFromIndex 11-139
Ivi_GetViInt32EntryFromString 11-141
Ivi_GetViInt32EntryFromValue 11-143
Ivi_GetViReal64EntryFromCmdValue 11-145
Ivi_GetViReal64EntryFromCoercedVal 11-147
Ivi_GetViReal64EntryFromIndex 11-149
Ivi_GetViReal64EntryFromString 11-151
Ivi_GetViReal64EntryFromValue 11-153
Ivi_InstrSpecificErrorQueueSize 11-155

Ivi_InterchangeCheck 11-157
Ivi_InvalidateAllAttributes 11-158
Ivi_InvalidateAttribute 11-159
Ivi_IOSession 11-161
Ivi_LockSession 11-162
Ivi_NeedToCheckStatus 11-165
Ivi_QueryInstrStatus 11-167
Ivi_QueueInstrSpecificError 11-168
Ivi_RangeChecking 11-170
Ivi_RangeTableFree 11-171
Ivi_RangeTableNew 11-172
Ivi_ReadInstrData 11-175
Ivi_ReadToFile 11-177
Ivi_RestrictAttrToChannels 11-179
Ivi_SetAttrCheckCallbackViInt32 11-181
Ivi_SetAttrCheckCallbackViReal64 11-181
Ivi_SetAttrCheckCallbackViString 11-181
Ivi_SetAttrCheckCallbackViBoolean 11-181
Ivi_SetAttrCheckCallbackViSession 11-181
Ivi_SetAttrCheckCallbackViAddr 11-181
Ivi_SetAttrCoerceCallbackViInt32 11-184
Ivi_SetAttrCoerceCallbackViReal64 11-184
Ivi_SetAttrCoerceCallbackViString 11-184
Ivi_SetAttrCoerceCallbackViBoolean 11-184
Ivi_SetAttrCoerceCallbackViSession 11-184
Ivi_SetAttrCoerceCallbackViAddr 11-184
Ivi_SetAttrCompareCallbackViInt32 11-187
Ivi_SetAttrCompareCallbackViReal64 11-187
Ivi_SetAttrCompareCallbackViString 11-187
Ivi_SetAttrCompareCallbackViBoolean 11-187
Ivi_SetAttrCompareCallbackViSession 11-187
Ivi_SetAttrCompareCallbackViAddr 11-187
Ivi_SetAttrComparePrecision 11-190
Ivi_SetAttributeFlags 11-192
Ivi_SetAttributeViInt32 11-194
Ivi_SetAttributeViReal64 11-194
Ivi_SetAttributeViString 11-194
Ivi_SetAttributeViBoolean 11-194
Ivi_SetAttributeViSession 11-194
Ivi_SetAttributeViAddr 11-194
Ivi_SetAttrRangeTableCallback 11-198
Ivi_SetAttrReadCallbackViInt32 11-200
Ivi_SetAttrReadCallbackViReal64 11-200

Ivi_SetAttrReadCallbackViString 11-200
Ivi_SetAttrReadCallbackViBoolean 11-200
Ivi_SetAttrReadCallbackViSession 11-200
Ivi_SetAttrReadCallbackViAddr 11-200
Ivi_SetAttrWriteCallbackViInt32 11-203
Ivi_SetAttrWriteCallbackViReal64 11-203
Ivi_SetAttrWriteCallbackViString 11-203
Ivi_SetAttrWriteCallbackViBoolean 11-203
Ivi_SetAttrWriteCallbackViSession 11-203
Ivi_SetAttrWriteCallbackViAddr 11-203
Ivi_SetErrorInfo 11-206
Ivi_SetIviIniDir 11-209
Ivi_SetNeedToCheckStatus 11-210
Ivi_SetRangeTableEnd 11-212
Ivi_SetRangeTableEntry 11-213
Ivi_SetStoredRangeTablePtr 11-215
Ivi_SetValInStringCallback 11-217
Ivi_Simulating 11-218
Ivi_SpecificDriverNew 11-219
Ivi_Spying 11-222
Ivi_UndefClass 11-223
Ivi_UndefDriver 11-224
Ivi_UndefHardware 11-225
Ivi_UndefLogicalName 11-226
Ivi_UndefVInstr 11-227
Ivi_UnlockSession 11-228
Ivi_Update 11-229
Ivi_UseSpecificSimulation 11-232
Ivi_ValidateAttrForChannel 11-233
Ivi_ValidateRangeTable 11-235
Ivi_ValidateSession 11-236
Ivi_WriteFromFile 11-237
Ivi_WriteInstrData 11-239
Ivi_WriteRunTimeDefinesToFile 11-240
Status Codes 11-241

Appendix A
Customer Communication

Glossary

Index

Figures

Figure 1-1. Instrument Driver External Interface Model 1-5
Figure 1-2. Instrument Driver Internal Design Mode 1-8

Figure 2-1. IVI Driver Operation Diagram 2-4

Figure 3-1. Instrument Driver Wizard Selection Panel... 3-5
Figure 3-2. Select Attribute Constant Dialog Box 3-7

Figure 4-1. Edit Driver Attribute Dialog Box 4-3
Figure 4-2. Edit Attribute Dialog Box 4-7
Figure 4-3. Edit Attribute Advanced Dialog Box 4-9
Figure 4-4. Range Tables Dialog Box.. 4-10
Figure 4-5. Edit Range Table Dialog Box 4-11

Figure 5-1. Function Tree 5-2
Figure 5-2. Function Tree Context Menu 5-3
Figure 5-3. Edit Instrument Dialog Box 5-10
Figure 5-4. Sample Function Tree 5-16

Figure 6-1. Function Panel Editor.. 6-2
Figure 6-2. Control Types 6-8
Figure 6-3. Create Input Control Dialog Box 6-8
Figure 6-4. Create Slide Control Dialog Box.. 6-9
Figure 6-5. Edit Label/Value Pairs Dialog Box 6-10
Figure 6-6. Create Binary Control Dialog Box 6-12
Figure 6-7. Edit On/Off Settings Dialog Box 6-13
Figure 6-8. Create Ring Control Dialog Box 6-14
Figure 6-9. Ring Control Edit Label/Value Pairs Dialog Box 6-15
Figure 6-10. Create Numeric Control Dialog Box.. 6-16
Figure 6-11. Edit Value Set Dialog Box 6-18
Figure 6-12. Create Output Control Dialog Box... 6-19
Figure 6-13. Create Return Value Control Dialog Box 6-20
Figure 6-14. Create Global Variable Control Dialog Box 6-21
Figure 6-15. Edit Data Type List Dialog Box... 6-23
Figure 6-16. Channel Create Binary Control Dialog Box... 6-26
Figure 6-17. Channel Edit On/Off Settings Dialog Box 6-27
Figure 6-18. Volts/Div Create Input Control Dialog Box... 6-27
Figure 6-19. Coupling Create Slide Control Dialog Box.. 6-28
Figure 6-20. Coupling Edit Label/Value Pairs Dialog Box 6-28
Figure 6-21. Invert Create Binary Control Dialog Box 6-29
Figure 6-22. Invert Edit On/Off Settings Dialog Box... 6-29
Figure 6-23. Function Panel Window 6-30

Figure 6-24. Change Input Control Type Dialog Box 6-31
Figure 6-25. Volts/Div Edit Label/Value Pairs Dialog Box 6-31

Figure 7-1. Help Editor Dialog Box 7-3
Figure 7-2. Sample Function Tree 7-7

Figure 8-1. Fluke 45 Digital Multimeter Function Tree 8-37
Figure 8-2. Fluke 45 Instrument Help 8-38
Figure 8-3. Fluke 45 Function Class Help 8-39
Figure 8-4. Fluke 45 Function Panel Help 8-40
Figure 8-5. Fluke 45 Function Panel Control Help 8-41
Figure 8-6. Fluke 45 Function Panel Status Control Help 8-42

Figure 10-1. Select an Instrument Driver Panel 10-2
Figure 10-2. General Information Panel 10-3
Figure 10-3. General Command Strings Panel 10-4
Figure 10-4. Standard Operations Panel 10-5
Figure 10-5. ID Query Panel 10-6
Figure 10-6. Reset Panel 10-7
Figure 10-7. Self-Test Panel 10-8
Figure 10-8. Revision Panel 10-9
Figure 10-9. Test Panel 10-10
Figure 10-10. IVI Test Results Panel 10-11
Figure 10-11. Finish Panel 10-12
Figure 10-12. Edit Driver Attributes Dialog Box 10-13
Figure 10-13. Edit Attribute Dialog Box 10-14
Figure 10-14. Edit Range Table Dialog Box 10-15
Figure 10-15. Edit Range Table Dialog Box With Modifications 10-16
Figure 10-16. Generate Code Dialog Box 10-19
Figure 10-17. Range Tables Dialog Box 10-19
Figure 10-18. Range Tables Dialog Box with Modifications 10-20
Figure 10-19. Fluke 45 Function Tree 10-22
Figure 10-20. Function Tree Editor Context Menu 10-23
Figure 10-21. Edit Group Dialog Box 10-27
Figure 10-22. Hold Enable Edit Attribute Dialog Box 10-29
Figure 10-23. Hold Threshold Edit Range Table Dialog Box 10-32
Figure 10-24. Hold Threshold Edit Attribute Dialog Box 10-33
Figure 10-25. Edit Function Panel Window Node Dialog Box 10-35
Figure 10-26. Create Binary Control Dialog Box 10-36
Figure 10-27. Edit On/Off Setting Dialog Box 10-36
Figure 10-28. Edit Slide Control Dialog Box 10-37
Figure 10-29. Edit Label/Value Pairs Dialog Box 10-37
Figure 10-30. Configure Hold Function Panel Window 10-40

Figure 10-31. Generate Documentation Dialog 10-42
Figure 10-32. Fluke 45 Instrument Driver Documentation Window 10-42
Figure 10-33. LabWindows/CVI Message Dialog Box 10-43
Figure 10-34. Select an Instrument Driver Panel 10-44
Figure 10-35. General Information Panel 10-45

Tables

Table 2-1. IVI Attribute Flags50 2-10
Table 2-2. Possible Values for the msg Parameter 2-28

Table 3-1. VISA Data Types 3-16

Table 7-1. Types of Help Information 7-2

Table 8-1. Instrument Driver Class Templates 8-2
Table 8-2. Data Types You Can Use for Attributes 8-7
Table 8-3. Attribute Callbacks That Can Call the Appropriate Default Callback 8-22
Table 8-4. VISA Data Types 8-31
Table 8-5. VISA I/O Library Macros 8-31
Table 8-6. Suggested Error Values 8-33
Table 8-7. Instrument Driver Completion and Warning Codes 8-33
Table 8-8. Instrument Driver Error Codes 8-34

Table 10-1. Multi-Point Operations 10-26
Table 10-2. Range Table Entry Information 10-31

Table 11-1. IVI Library Function Tree 11-3
Table 11-2. Module Types on Different Platforms 11-73
Table 11-3. optionsString Values 11-81
Table 11-4. optionsString Values 11-220
Table 11-5. Status Code Ranges 11-241
Table 11-6. Default Values of Defined Constants 11-242
Table 11-7. IVI Errors and Warnings 11-242
Table 11-8. Common Errors and Warnings 11-245
Table 11-9. Most Often Encountered VISA Errors and Warnings 11-246

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 customers 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* guide, the *LabWindows/CVI User Manual*, and the *LabWindows/CVI Standard Libraries Reference Manual*, 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.

Organization of This Manual

The *LabWindows/CVI Instrument Driver Developer Guide* is organized as follows:

- Chapter 1, *Instrument Driver Overview*, introduces the concept of instrument drivers, explains how they have evolved, and then explains the implementation of them in LabWindows/CVI.
- Chapter 2, *IVI Architecture Overview*, contains a general overview of the concepts of the IVI (Intelligent Virtual Instrumentation) instrument drivers and the IVI engine. It also contains detailed descriptions of the inherent IVI attributes.
- Chapter 3, *Developing an Instrument Driver*, explains the proper procedure for developing an instrument driver.
- Chapter 4, *Attribute Editor*, describes the operation of the Attribute Editor.
- Chapter 5, *Function Tree Editor*, explains the function tree and the Function Tree Editor, and describes the Function Tree Editor menu bar, menus, and commands.

- Chapter 6, *Function Panel Editor*, describes how to create and modify instrument driver function panels using the Function Panel Editor.
- Chapter 7, *Adding Help Information*, describes the types of help information available from an instrument driver and how you can create help information.
- Chapter 8, *Programming Guidelines for Instrument Drivers*, 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 may 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.
- Chapter 9, *Required Instrument Driver Functions*, contains information and function descriptions for required instrument driver functions.
- Chapter 10, *Instrument Driver Examples*, shows you how to create an IVI instrument driver. The examples in this chapter can serve as models for your instrument driver development.
- Chapter 11, *IVI Library*, describes the functions in the LabWindows/CVI IVI (Intelligent Virtual Instruments) Library. The *IVI Library Function Overview* section contains general information about the IVI Library functions and panels. The *IVI Library Function Reference* section contains an alphabetical list of function descriptions.
- Appendix A, *Customer Communication*, contains forms you can use to request help from National Instruments or to comment on our products and manuals.
- The *Glossary* contains an alphabetical list and description of terms used in this manual.
- The *Index* contains an alphabetical list of key terms and topics in this manual, including the page where you can find each one.

Conventions Used in This Manual

The following conventions are used in this manual:

<> Angle brackets enclose the name of a key on the keyboard—for example, <shift>.

	This icon to the left of bold italicized text denotes a note, which alerts you to important information.
	This icon to the left of bold italicized text denotes a caution, which advises you of precautions to take to avoid injury, data loss, or a system crash.
bold	Bold text denotes the names of menus, menu items, parameters, dialog boxes, and dialog box buttons.
bold italic	Bold italic text denotes an activity objective, note, caution, or warning.
italic	Italic text denotes variables, emphasis, a cross reference, or an introduction to a key concept. This font also denotes text for which you supply the appropriate word or value.
`monospace`	Text in this font denotes text or characters that should literally 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 for statements and comments taken from programs.
`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.
paths	Paths in this manual are denoted using backslashes (\) to separate drive names, directories, folders, and files.

The LabWindows/CVI Documentation Set

For a detailed discussion of the best way to use the LabWindows/CVI documentation set, see the section *The LabWindows/CVI Documentation Set* in the *About This Manual* section of the *Getting Started with LabWindows/CVI* manual.

Customer Communication

National Instruments wants to receive your comments on our products and manuals. We are interested in the applications you develop with our products, and we want to help you if you have problems with them. To make it easy for you to contact us, this manual contains comment and technical support forms for you to complete. These forms are in Appendix A, *Customer Communication*, at the end of this manual.

1

Instrument Driver Overview

This chapter introduces the concept of instrument drivers, explains how they have evolved, and then explains the implementation of them in LabWindows/CVI.

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

However, the VXI*plug&play* model is not the last word in instrument drivers. The IVI (Intelligent Virtual Instrumentation) model takes the old model a step further without introducing additional complexity or performance overhead. Although IVI instrument drivers comply with the VXI*plug&play* standard, they have many additional features. Two of the most important features are:

- 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. Through the IVI attribute model, IVI drivers automatically cache the current state of the instrument. An IVI instrument 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.
- Instrument simulation. IVI drivers can simulate the operation of instruments when they are not available. IVI drivers 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 and coerced to proper values when out of range.

For a comprehensive list of IVI features, see 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 may convert a binary array of two-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 five files:

- The instrument driver program, which can be a `.lib`, `.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 **Edit Instruments Attributes** command in the **Tools** menu.
- An ASCII text file (`.doc`), which contains documentation for the instrument driver

The five filenames consist of the driver name, followed by the appropriate extension. For example, if the instrument driver name for the Tektronix 2430A digitizing oscilloscope is tek2430a, its files are named `tek2430a.c` (`.obj`, `.lib`, or `.dll`), `tek2430a.h`, `tek2430a.fp`, `tek2430a.sub`, and `tek2430a.doc`.

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-specific 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 representing 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 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. The first step in creating a standard for instrument drivers, therefore, is to define a model to explain how the instrument driver interacts with the rest of the system.

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

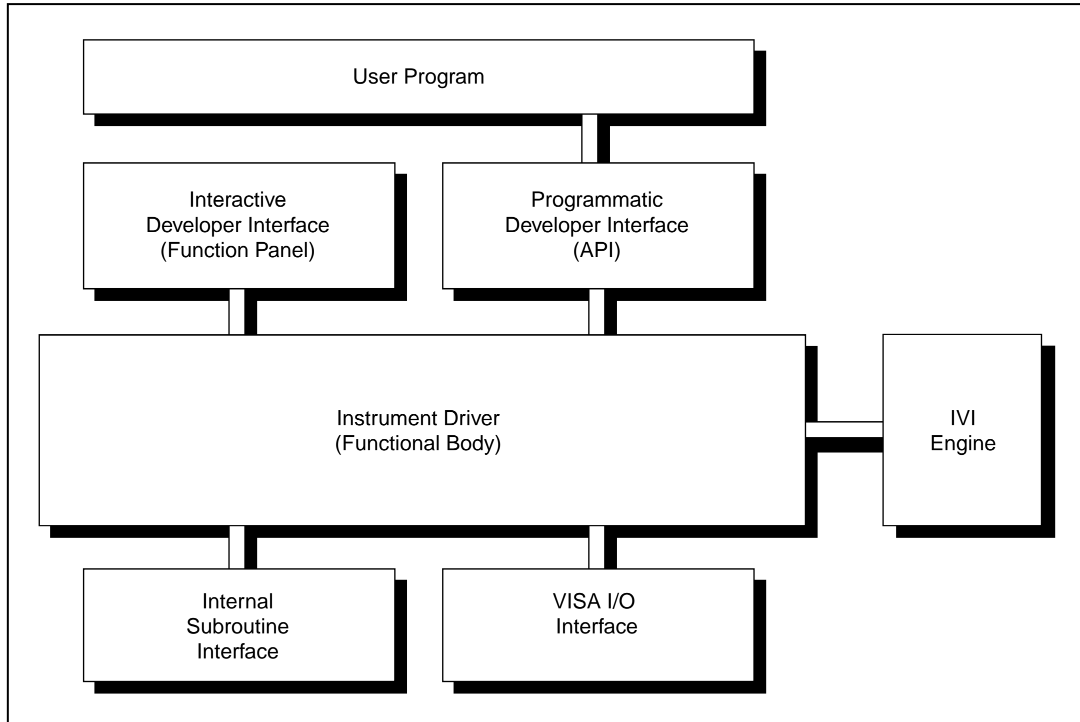


Figure 1-1. Instrument Driver External Interface Model

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 *Intelligent Virtual Instrumentation 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 *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 may include operating system calls or calls to other unique libraries such as formatting and analysis functions.

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

Functional Body

The functional body of a LabWindows/CVI instrument driver is a library of C functions for controlling a specific instrument. Because the functional body is developed with the standard tools provided in the LabWindows/CVI environment, users can easily view instrument driver source code and optimize it for their application. The details of the functional body are based on the instrument driver internal design model. Chapter 8, *Programming Guidelines for Instrument Drivers*, describes the guidelines for creating the instrument driver functional body.

IVI Engine

IVI, an acronym for Intelligent Virtual Instruments, 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 (Virtual Instrument Software Architecture) I/O interface is the National Instruments next-generation I/O architecture. VISA includes a single interface library for controlling GPIB, VXI, RS-232, 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.

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. This interface is accomplished through standard software function calls, 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, the programmatic developer interface to the instrument driver can be enhanced 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.

Instrument Driver Internal Design Model

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

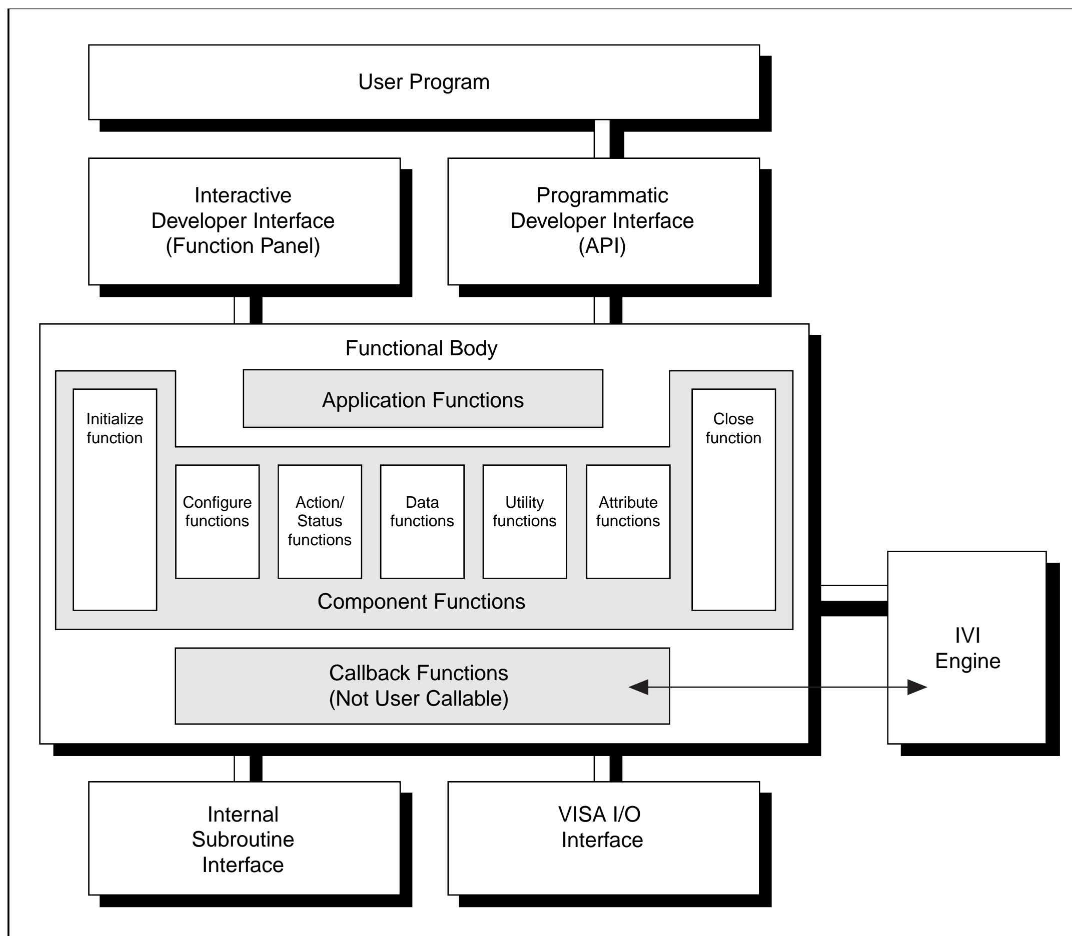


Figure 1-2. Instrument Driver Internal Design Mode

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 callback functions that the IVI engine invokes to perform such operations as reading and writing instrument settings or querying the state of an instrument. Non-IVI drivers do not have 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. A user can, for example, 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. The specific routines in each category are further categorized as either *required functions* or *developer-specified functions*.

The required functions are instrument driver functions that are common to the majority of instruments. These functions perform the following instrument operations:

- Initialize
- Close
- Reset
- Self-Test
- Revision Query
- Error Query
- Error Message
- Get/Clear Error Info
- 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 architecture, on the other hand, provides a framework for instrument class definitions. An instrument class definition defines the external interface and functionality of all instrument drivers for a particular instrument type. It defines some functionality as required, and some as optional. It also allows instrument drivers to define additional functions specific to a particular instrument. You can use the IVI architecture without complying with a class definition, and class definitions do not exist for all instrument types. Nevertheless, you gain the following benefits if you create an instrument driver according to a class definition:

- The class definition makes most of the design decisions for you.
- The class definition provides template files that you can use with the **Create IVI Instrument Driver** command in the **Tools** menu. 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 all of the attributes and high-level functions that the class defines to the instrument driver files. Refer to Chapter 3, *Developing an Instrument Driver*, for more information.
- By following a class definition, 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.
- The standard class definitions also make it possible to create *class instrument drivers* that work on all instruments of the same type.

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. They 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.

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.

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	The reset function places the instrument in a default state.
Revision Query	The revision query function returns the revision of the instrument driver and the firmware revision of the instrument.
Error Query	The error query function queries the status of the instrument and returns the instrument-specific error information
Error Message	The error message function translates the error return value from an instrument driver function to a user-readable string.

Get/Clear Error Info	The get and clear error info functions allow you to get and clear the extra error information that IVI drivers provide.
Lock/Unlock Session	The lock and unlock session functions allow you to protect a sequence of calls to an IVI instrument driver from interference by other execution threads.
Write/Read Instrument Data	The write/read instrument data functions allow the end-user to perform instrument I/O directly.

Close Function

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

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, an acronym for Intelligent Virtual Instruments, 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. Following are the major features of the IVI instrument driver architecture:

- 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 and thereby increase 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.

- The standard class definitions also make it possible to create *class instrument 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 definition.
- LabWindows/CVI command to create a C++ wrapper for an IVI instrument driver.
- Multiplatform capability. The IVI engine will be available under Windows 95/NT, Windows 3.1, Sun Solaris 1, Sun Solaris 2, and HP-UX. Under Windows 95/NT, 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. The balance of this chapter 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 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 Chapter 11, *IVI Library*, 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, or serial instruments or other devices. With an instrument driver, you can easily 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, IVI adds many features to instrument drivers, including:

- State-caching to eliminate redundant commands being sent 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

Note ***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, as well as *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_ConfigureVertical` 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_ConfigureVertical` 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_ConfigureVertical 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_ConfigureVertical.

The following diagram illustrates this process.

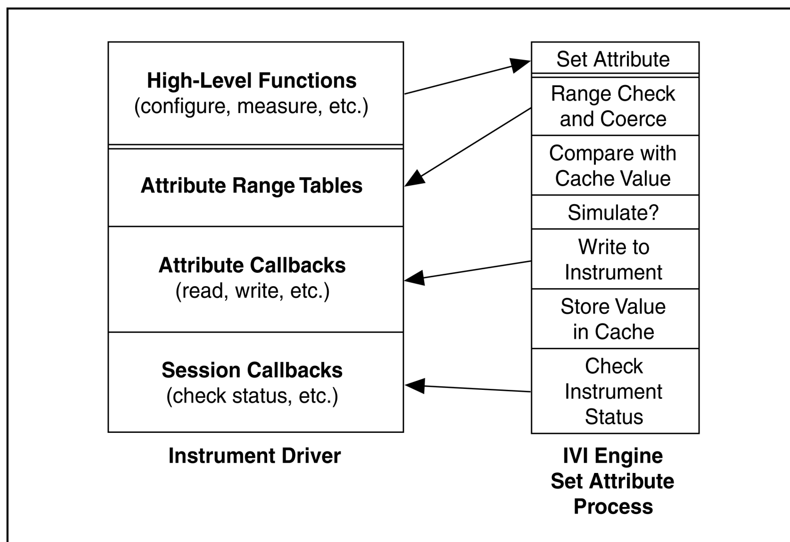


Figure 2-1. IVI Driver Operation Diagram

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 particular 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_ReadInstrData
Prefix_WriteInstrData
```

Refer to Chapter 9, *Required Instrument Driver Functions*, 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 also 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

The interface to many of the standard capabilities in the IVI engine is through attributes. The IVI engine defines a set of attributes that are present for all IVI sessions. 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 Chapter 11, *IVI Library*.

Chapter 11, *IVI Library*, 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

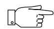
An instrument driver creates the specific and class attributes it uses by calling the `Ivi_AddAttribute` functions. The IVI Library provides a separate `Ivi_AddAttribute` function for each of the six data types, for example, `Ivi_AddAttributeViInt32` and `Ivi_AddAttributeViBoolean`. In the include file 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* **Create IVI Instrument Driver** *and* **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`, except 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, then 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 Flags50

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

Table 2-1. IVI Attribute Flags50 (Continued)

Bit	Value	Flag
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:

1. The instrument always coerces the same value in the same way.
2. 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 an 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, and 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, and 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 which 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 **Edit Instrument Attributes** command, 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 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 do computers. 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

  if $|b| < 10^{-(d-1)}$

    then a == b.

else /* a != 0 */

  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 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 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 1,00.0. 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 1,00.0 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 *Coerce Callback* and *Range Tables* sections in this chapter.

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 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 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, and a third might accept a continuous range of values and coerce them to still another discrete set. Standard class definitions handles 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 which might seem to require value coercion but which, 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 install 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.

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 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 through the Ivi_GetAttrRangeTable function. 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 read 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 of 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 of 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 of 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 the following table:

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 Chapter 11, *IVI Library*.

Channels

Many instrument 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 assign specific driver channel strings to virtual channel names in the `ivi.ini` configuration file. 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 an 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 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 vertical*, 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 can also 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 example code illustrating how high-level functions implement the user capabilities. Also, if you use the **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 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 of 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 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 an 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 that 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 which 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 a 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 insure 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 at which the program configures the instrument and the time at which 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. It can later trigger all of 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. Generally, 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. You can defer updates in an application program, but only when using the *Prefix*_SetAttribute functions. Do not enable IVI_ATTR_DEFER_UPDATE when calling the high-level functions of an instrument driver.

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 of 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.

Refer to the function description for Ivi_Update in Chapter 11, *IVI Library*, for detailed information on what Ivi_Update and Ivi_DeferredBufferedIOCallback do.

Configuration Entries

When an 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.

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.

Contact National Instruments for more information on using IVI drivers for instrument interchangeability.

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 table shows the IVI attributes in their categories.

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`
Use Specific Simulation	`IVI_ATTR_USE_SPECIFIC_SIMULATION`
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 Capability Strings	
Class Group Capabilities	`IVI_ATTR_GROUP_CAPABILITIES`
Class Function Capabilities	`IVI_ATTR_FUNCTION_CAPABILITIES`
Class Attribute Capabilities	`IVI_ATTR_ATTRIBUTE_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_ATTRIBUTE_CAPABILITIES

Data Type: ViString
Restrictions: Not writable by user

A comma-separated string that identifies the class attributes that the specific instrument driver implements. The class driver creates the value of this attribute based on the state of the IVI_VAL_NOT_SUPPORTED flag for each attribute in the session.

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 the value of this attribute.

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 or not 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. 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_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 IVI_ERROR_NOT_CREATED_BY_CLASS 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 IVI_ERROR_NOT_CREATED_BY_CLASS 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`, then `IviDmm` is the prefix for that 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_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 `IVI_ERROR_NOT_CREATED_BY_CLASS` error whenever you attempt to set or get this attribute.

IVI_ATTR_DEFER_UPDATE

Data Type: `ViBoolean`
Restrictions: None

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.

Generally, 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 through a class driver, the user can also 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 concerning 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. The class driver sets the value of this attribute based on the functions it finds in 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_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. The class driver sets the value of this attribute based on the functions and attributes in 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_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 IVI_ERROR_NOT_CREATED_BY_CLASS 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.

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_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 IVI_ERROR_NOT_CREATED_BY_CLASS 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 checks 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 parameters 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 very 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 Ivi_GetNextCoercionInfo 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: None

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.

Generally, only instrument driver developers use this attribute. 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 concerning 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 or not 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, then 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_USE_SPECIFIC_SIMULATION

Data Type: ViBoolean
Restrictions: None

This attribute specifies whether the specific driver or the class driver simulates I/O operations when simulation is enabled. A value of VI_TRUE specifies that the specific driver simulates I/O operations. A value of VI_FALSE specifies that the class driver simulates I/O operations through a special simulation driver.

The user specifies the value of IVI_ATTR_USE_SPECIFIC_SIMULATION.

When the user opens an instrument session through a class driver, the default value is VI_FALSE, but the user can override the default value by specifying a value in the ivi.ini

configuration file. When the user opens an instrument session through a specific driver, the default value is `VI_TRUE`.

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

IVI_ATTR_VISA_RM_SESSION

Data Type: `ViSession`
Restrictions: Read-only

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 through the **Create IVI Instrument Driver** command in the **Tools**. 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 through the **Edit Instrument Attributes** command the **Tools** menu.
- 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 in this chapter and 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 which defines common functions and attributes for these types of instruments. The wizard generates a function panel (`.fp`) file, source (`.c`), include (`.h`), and `.sub` file for you, automating much of the tedious effort required when writing a driver.

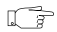
To write the driver for your specific instrument, we recommend the following procedure:

1. Name the instrument driver.
2. Define the instrument functions and function classes (automated by the wizard when you use a template).
3. Define the attributes (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 without loading the source code.
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 one of the following:

- An IVI instrument class template
- An existing driver for a similar instrument
- The core IVI driver template

Before using the wizard, complete the worksheet below 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 **Create IVI Instrument Driver** from the **Tools** menu 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 selection you make.

Selecting an Instrument Driver Template

After you click on the **Next** button on the initial wizard panel, you see the following panel.

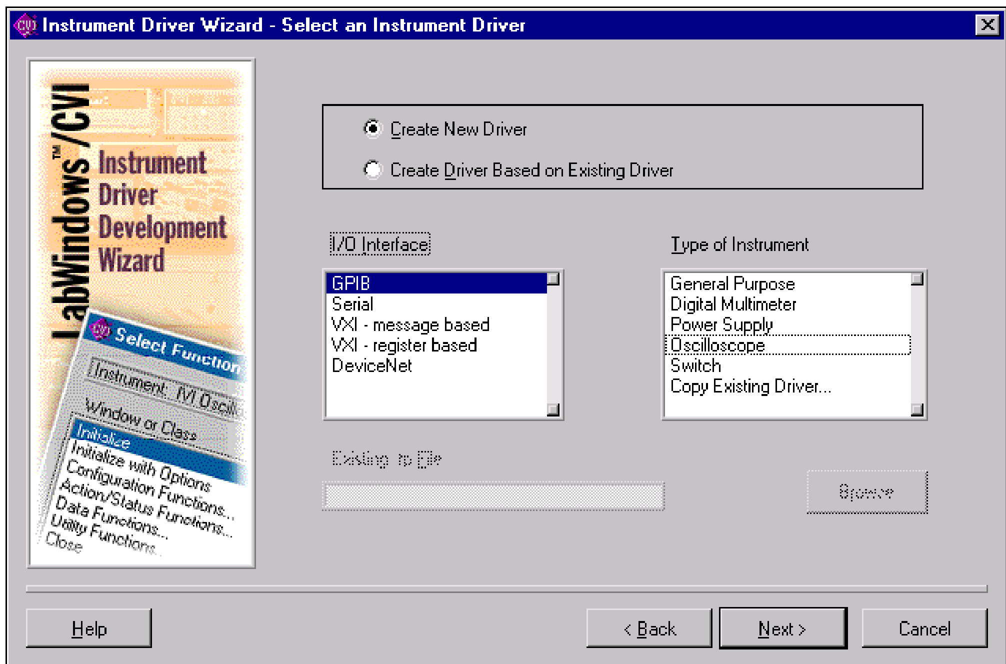


Figure 3-1. Instrument Driver Wizard Selection Panel

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 file name.

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 earlier in the wizard. After the delays that you specify for the self-test and reset operations, 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.

The Generated Function Panels

Your instrument driver function panel file displays all the instruments 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.

The .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 the one of the `SetAttribute` functions from the Configuration Functions in your driver. Click on the Attribute control to display the Select Attribute Constant dialog box. You can select each attribute and view the possible values for that attribute in the lower half of the dialog. The following figure illustrates the Select Attribute Constant dialog box.

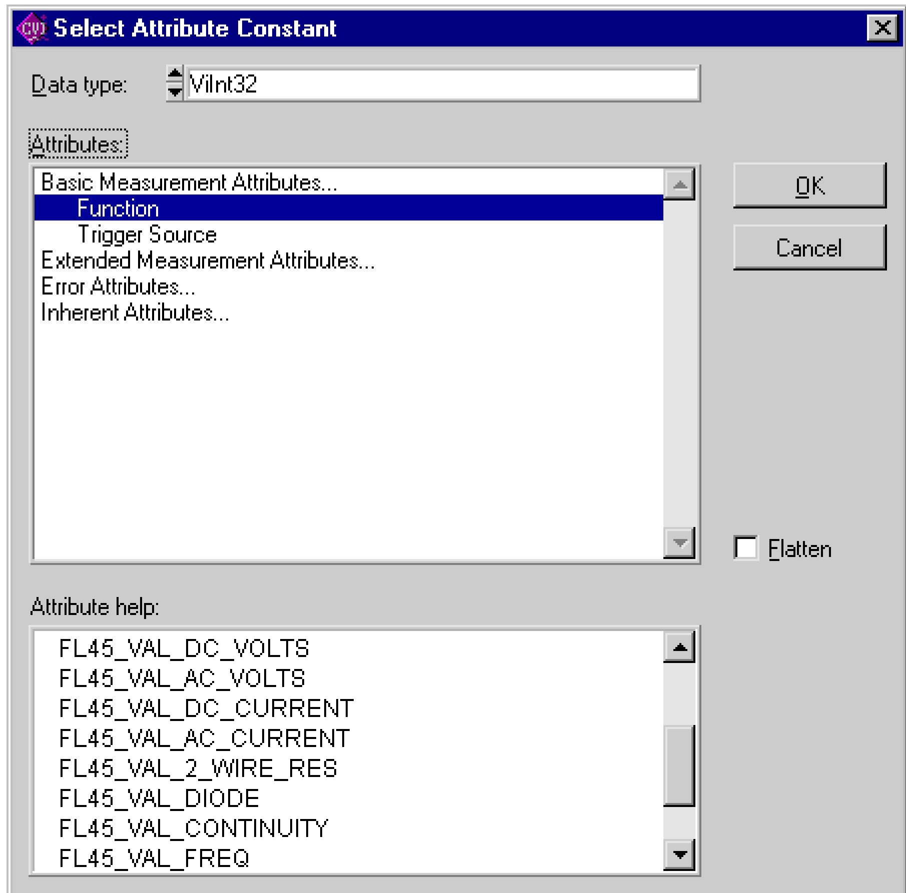


Figure 3-2. Select Attribute Constant Dialog Box

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 the **Edit Instrument Attributes** command from the **Tools** menu. 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*.

The 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 and 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 the acquisition mode of 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.

The 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 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 the 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 `Configure`, for setting up multiple parameters.

Defining the Hierarchy of Functions

It is very 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 broken 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 listed below.

```
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 runtime 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 function prototype shown below.

```
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 shown below.

```
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 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 function prototype shown below.

```
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

Create a user-defined data type for use in a function panel as follows:

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 **Data Types** command in the **Options** menu in the Function Panel Editor.

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;
```

Next, make the `waveform_var` data type available in the function panel file. Select **Data Types** from the **Options** menu of 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`.

See 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.)

to 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. These special data types for instrument drivers are as follows.

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
`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 **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.

See 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 easily use.
    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 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 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 *Prefix*_IviInit. The *Prefix*_close function calls the *Prefix*_IviClose function. If a user opens the IVI session from a class instrument driver, the *Prefix*_IviInit function in the class driver calls the *Prefix*_IviInit function in the specific driver, and the *Prefix*_IviClose function in the class driver calls the *Prefix*_IviClose function in the specific driver.

Chapter 9, *Required Instrument Driver Functions*, describes the implementation guidelines for the required instrument driver operations.

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 can also 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`, `.lib`, or `.dll`), `.h`, and `.fp` files, you can operate the instrument driver. Load the driver using the **Load** command in the **Instrument** menu 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*, tells more 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 of 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 of 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, and contains a function reference list explaining 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 executing the **Edit Instrument Attributes** command from the **Tools** menu of 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 violates 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 can also 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 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 *Error Macros* section in Chapter 11, *IVI Library*, for more information on `checkErr`.

Edit Driver Attributes Dialog Box

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

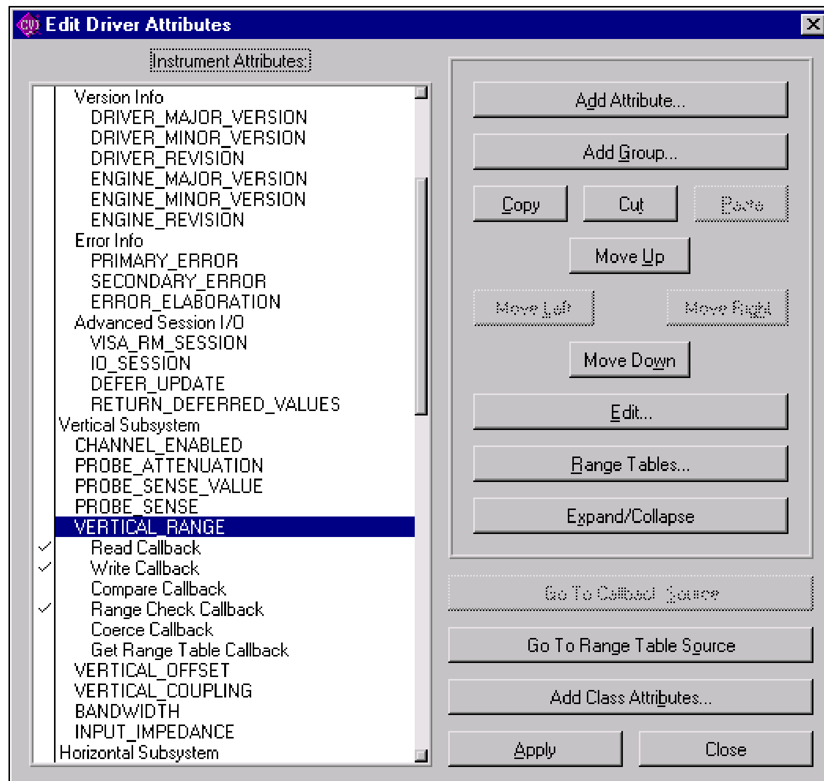


Figure 4-1. Edit Driver Attribute Dialog Box

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 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 <Spacebar> 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 check mark 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 check mark. If you add a check mark 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 check mark by pressing <Spacebar> or by clicking on the check mark.

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 can also 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** buttons to move attributes up or down in the list box or 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 **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. Under Windows 3.1, it appends a tilde (~) to each extension instead. 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.

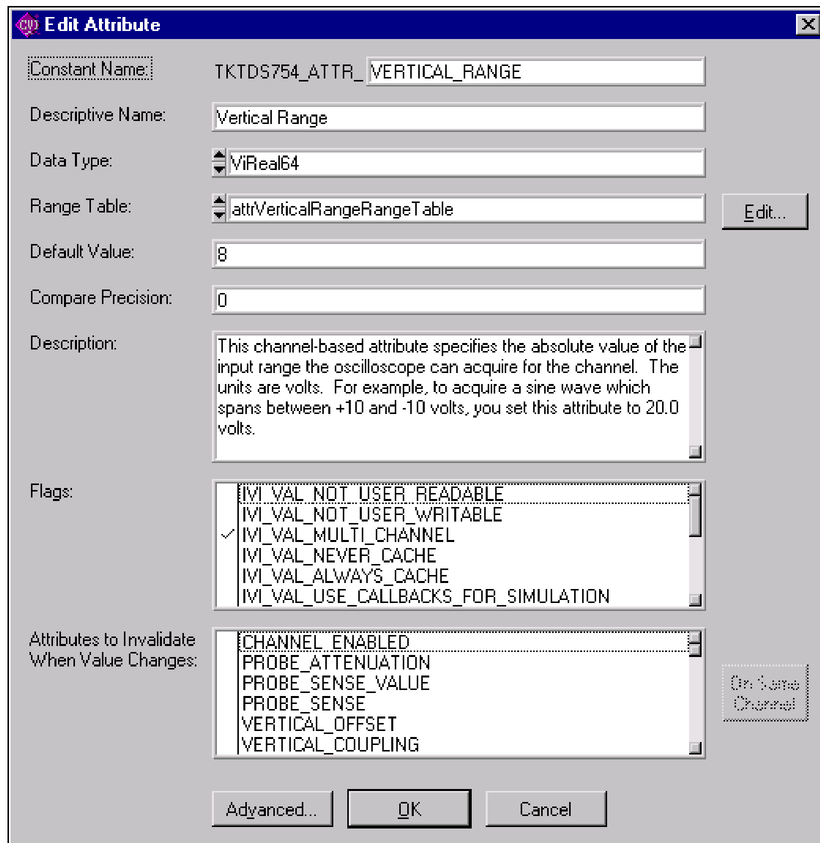


Figure 4-2. Edit Attribute Dialog Box

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

- **Attribute Constant**—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**—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 one of `ViInt32`, `ViReal64`, `ViString`, `ViBoolean`, `ViSession`, and `ViAddr`. You can use `ViAddr` only for attributes that you hide from the instrument driver user.
- **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.
- **Range Table**—A ring control that lets you select a static range table for the attribute. You can select an existing range table in the ring. If you do not want a range table, you can select the `none` entry in the ring. The Range Table ring control is enabled only for `ViInt32` and `ViReal64` attributes. The data type of the attribute must match the data type that appears in the Edit Range Table dialog box for the range table that you select.
- **Edit button**—Lets you invoke the Edit Range Table dialog box for the range table that currently appears in the Range Table ring. When the `none` entry appears in range table ring, 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 to 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.
- **Attribute 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.
- **Attribute Flags**—A list of the flags that you can set for the attribute. You set a flag by adding a check mark next to its name. You can hide an attribute from the instrument driver user by adding check marks 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**—A list of all the other attributes in the instrument driver. Add a check mark 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 check mark. Refer to the *IVI State-Caching Mechanism* section in Chapter 2, *IVI Architecture Overview*, for information about invalidation of attribute cache values.
- **On All Channels**—A toggle button that 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 check mark, the (`All`) or (`Same`) tag appears at the end of the item label. If the item you select does not have a check mark, 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 button**—invokes the Edit Attribute Advanced dialog box, which appears in Figure 4-3.

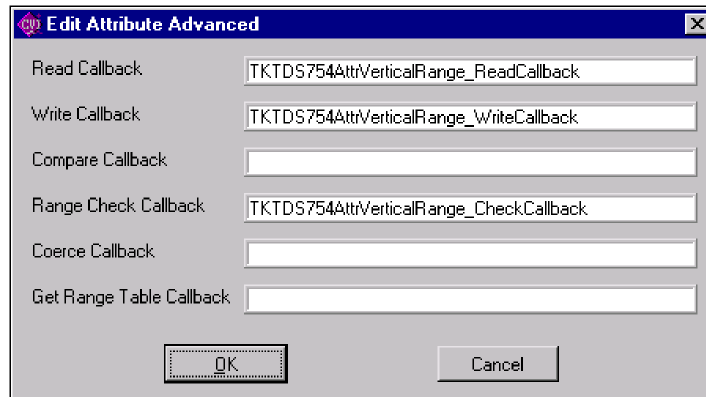


Figure 4-3. Edit Attribute Advanced Dialog Box

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 in Figure 4-4.

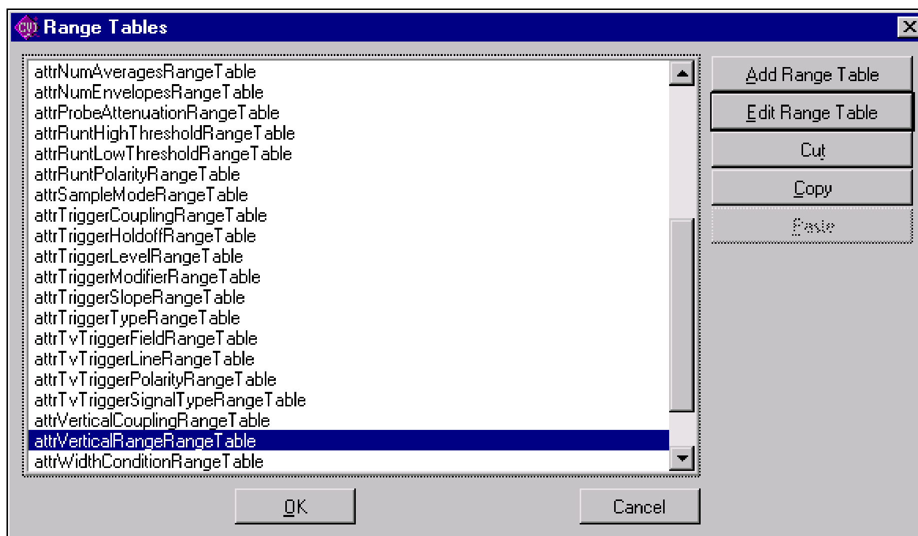


Figure 4-4. Range Tables Dialog Box

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 in Figure 4-5.

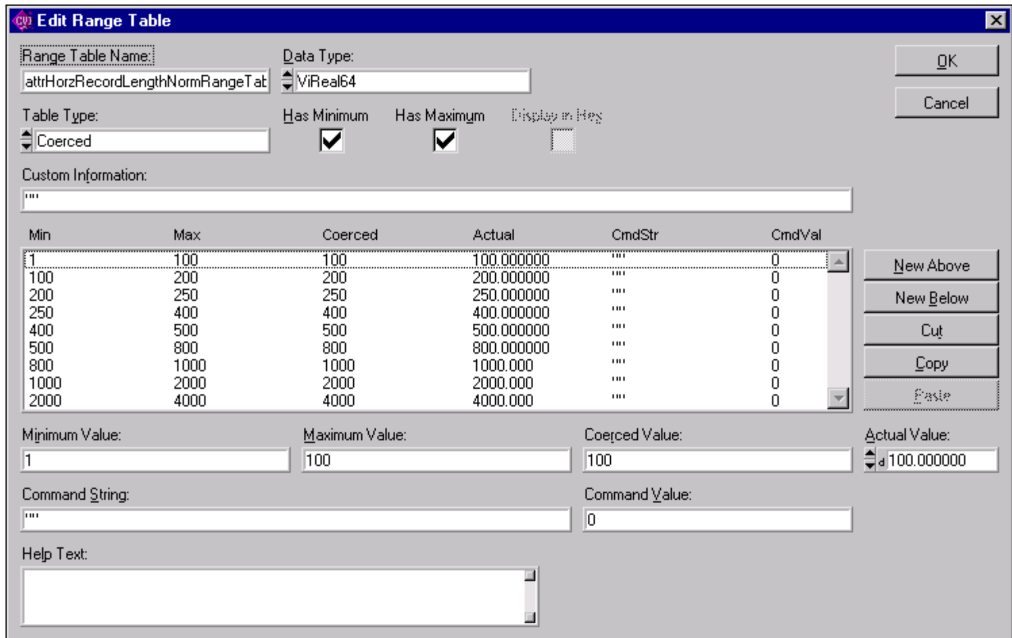


Figure 4-5. Edit Range Table Dialog Box

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**—The name you use 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**—You can 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**—A list box that contains the contents of each range table entry. 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, and 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 which 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 the **Function Tree (*.fp)** option from either the **New** or **Open** commands in the **File** menu.

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 **Edit** from the **Instrument** menu. Then highlight the name of the

instrument in the selection list of the Edit Instrument dialog box and press the **Edit Function Tree** button. A function tree appears in Figure 5-1.

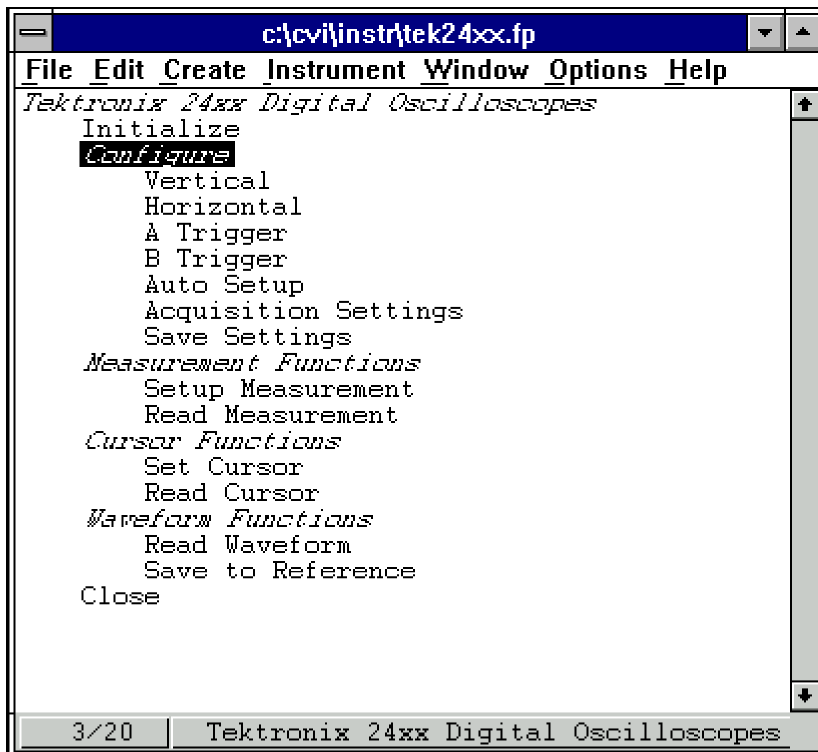


Figure 5-1. Function Tree

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.

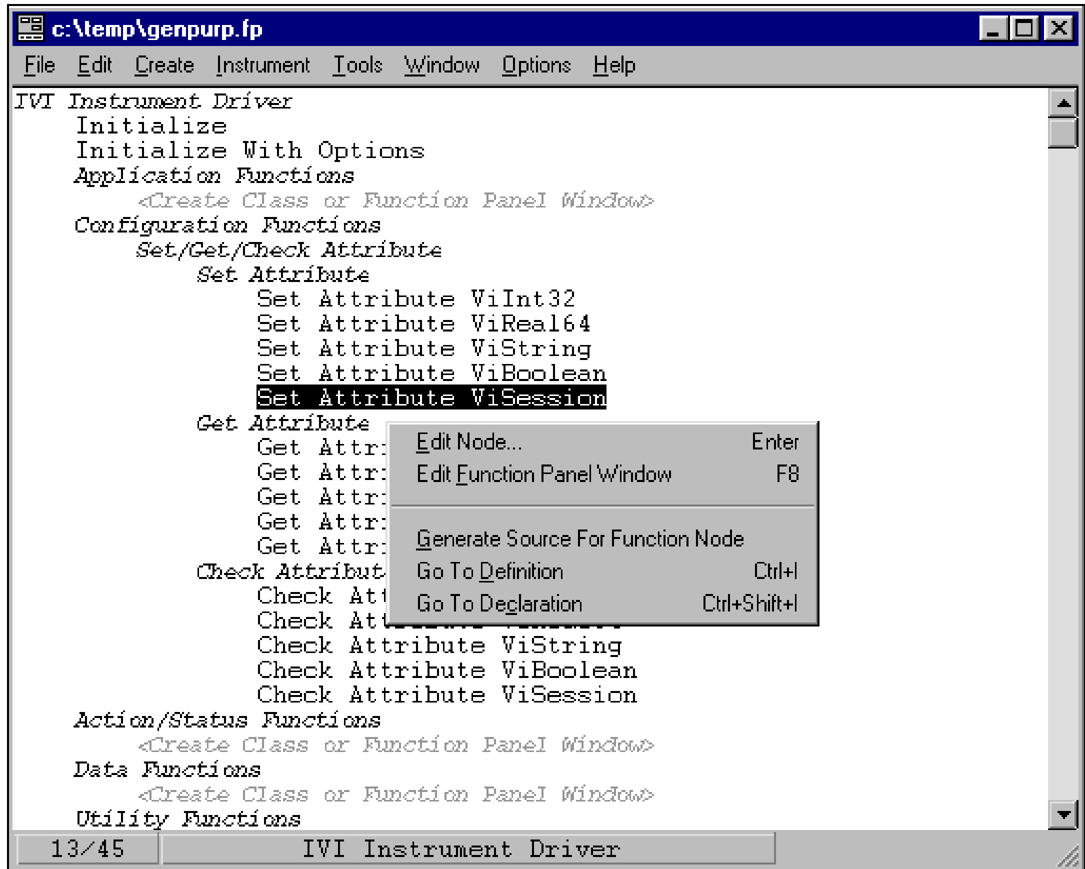


Figure 5-2. Function Tree Context Menu

The menu options are the same or similar to options available through the menu bar menus. The options are:

- **Edit Node** is the same as the **Edit Node** command available from the **Edit** menu.
- **Edit Function Panel Window** is the same as the **Edit Function Panel Window** command available from the **Edit** menu.
- **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. The *LabWindows/CVI User Manual*, Chapter 3, *Project Window*, tells more 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.
- When you cut or copy a class to the Function Tree Editor Clipboard, all of its subclasses and functions are cut or copied as well. Similarly, when you paste the class, all of its subclasses and functions are also pasted.
- **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. See 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. Chapter 6, *Function Panel Editor*, gives you 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* discussion for more information.

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 does continue 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 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 through the **Instrument Directories** command in the **Options** menu of 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. If the user finds the .fp file, the user is prompted to add the directory to the instrument directories list. The user is also given the option to add the file to the project.

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

When the user selects the **Unload** command from the **Instrument** menu, 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.

You have the following options in the **Create** menu.

- **Instrument** lets you create a new function tree.
- **Class** lets you add a new class to the function tree.
- **Function Panel Window** lets you add a new function to the function tree.

Instrument

The **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

Add a class to an empty tree as follows.

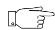
1. Select the line containing `Create Class or Function Panel Window`.
2. Execute the **Class** command in the **Create** menu. The Create Class Node dialog box appears.
3. Complete the Create Class Node dialog box. The class appears in the function tree window.

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. To insert a class into a function tree, follow these steps.

1. Select an existing function or class at the level you want to place the new class.
2. Execute the **Class** command in the **Create** menu. 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

The **Function Panel Window** command of the **Create** menu lets you 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 in the*** **Instrument** ***command in the*** **Create** ***menu.***

Adding a Function to an Empty Tree or Class

Add a function to an empty tree or class as follows:

1. Select the line containing Create Class or Function Panel Window.
2. Execute the **Function Panel Window** command in the **Create** menu. 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

Insert a function at any level in an existing function tree as follows:

1. Select an existing function or class at the level you want to place the new function.
2. Execute the **Function Panel Window** command in the **Create** menu. 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. The **Instrument** menu presents the following standard options.

- **Load** lets you add an instrument driver to the **Instrument** menu.
- **Unload** lets you remove one or all instrument drivers from the **Instrument** menu.
- **Edit** lets you invoke the Function Panel Editor or modify the relationship between the function panel file and its associated program file.

Load

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

Unload

- The **Unload** command removes 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 pressing the **Check All** button.
- Deselect all instrument drivers by pressing the **Check None** button.

- Press the **OK** button to unload the selected instrument drivers.
- Press 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 **Edit** from the **Instrument** menu, the dialog box shown in Figure 5-3 appears.

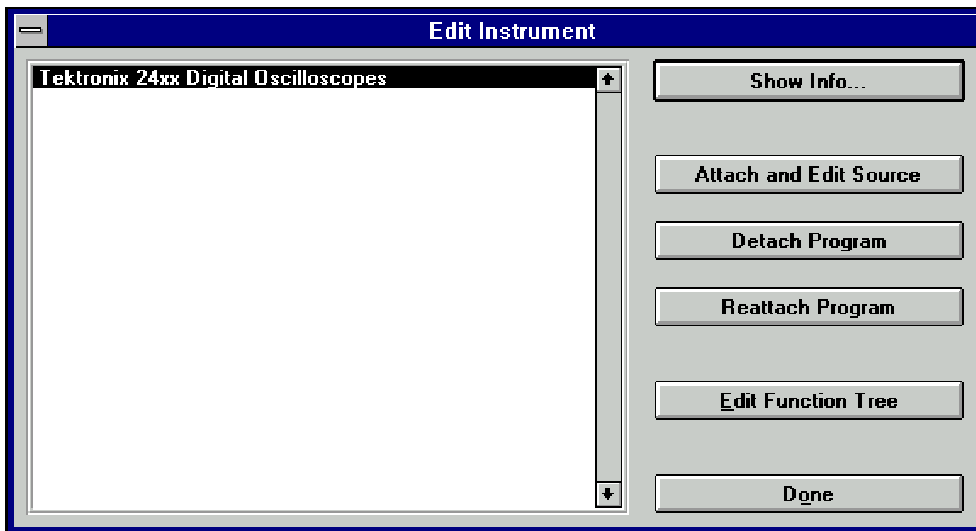


Figure 5-3. Edit Instrument Dialog Box

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 .lib, .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 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 an .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 only available for IVI drivers.
- **Enable Auto replace** enables the updating of an instrument driver's source files to reflect changes to function names in the function tree. This option is global to 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 Source For Function Tree** generates function definitions and declarations into your driver's source and include files. For each function panel in the function tree, the source (.c) file is searched for the function's definition. The header (.h) file is searched for the function's 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.

Window

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

- **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 containing prototypes for the functions in the function tree.
- **Generate Documentation** creates a window containing 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 DLL Make Files** (Windows 3.1 only) creates a `.mak` and a `.def` file to compile your instrument driver C source code into a 16-bit DLL. You are prompted to specify the target compiler, Microsoft Visual C++ or Borland C++.

- **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* which describes the function entry points in the DLL. For information on using type libraries see the *OLE 2 Programmers Reference, Volume 2*, from Microsoft Press.
- **Generate DEF File** (Windows 95/NT) 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** (Windows 95/NT) 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. For more information on creating DLLs, see the *Preparing Source Code for Use in a DLL* section in Chapter 3, Windows 95/NT Compiler/Linker Issues, in the *LabWindows/CVI Programmer Reference Manual*.
- **VXIplug&play Style** (Windows 95/NT) 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 of these settings using commands 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 containing all of 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 are the following.
        - A Visual Basic include file, which you can create using the **Generate Visual Basic Include** command in the **Options** menu of the Source window
        - A documentation file, which you can create using the **Generate Documentation** command in this menu
        - A help file, which you can create using the **Generate Windows Help** command in this menu and the Windows help compiler
        - A knowledge base file as defined in 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 the following.

- Creating a function tree with multiple classes
- Cutting and pasting functions and classes in a function tree
- Cutting and pasting functions and classes between the function trees of different drivers

In this example, you create function trees and panels without writing any code.

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 **New, Function Tree (*.fp)** from the **File** menu.

Create a new instrument and function tree as follows.

1. Execute **Instrument** from the **Create** menu.
2. Enter the name `Function Tree Examples` as the Name and `tree` as the Prefix. Click on **OK**.
3. Execute **Function Panel Window** from the **Create** menu.
4. Enter the name `Function 1` as the Name and `fun1` as the Function Name. Click on **OK**.
5. Execute **Class** from the **Create** menu.
6. Enter the name `Class 1` as the Name. Click on **OK**.
7. Select the line beneath the name `Class 1`.
8. Execute **Function Panel Window** from the **Create** menu.
9. Enter the name `Function 2` as the Name and `fun2` as the Function Name. Click on **OK**.
10. Execute **Save .FP File As** from the **File** menu and save the file as `mltcls`.

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

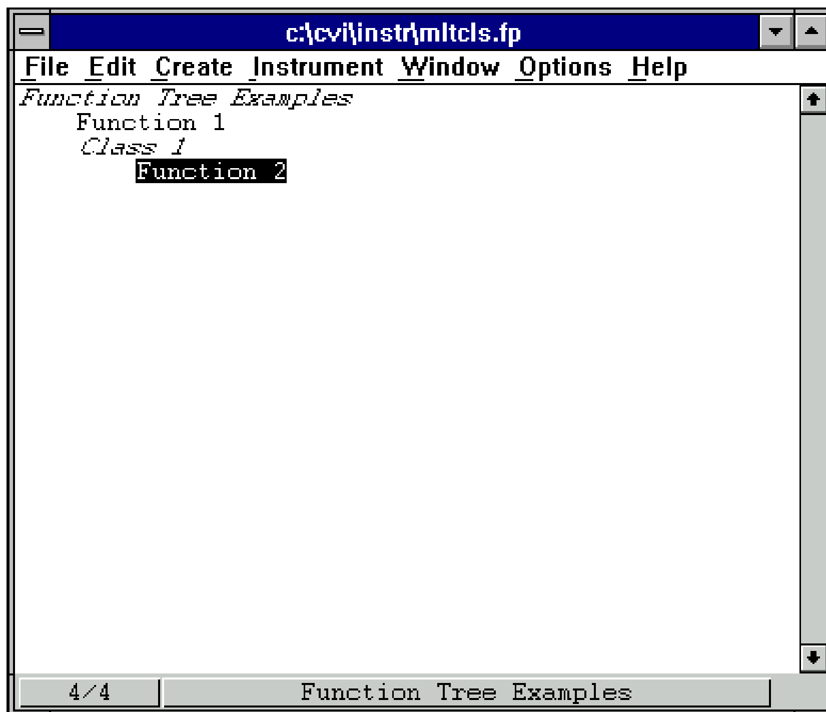


Figure 5-4. Sample Function Tree

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.

Cut and paste a function within a function tree as follows.

1. Position the selection on the name `Function 1`.
2. Execute **Cut** from the **Edit** menu. The function disappears from the tree and is stored on the Clipboard.
3. Position the selection on the name `Function 2`.
4. Execute **Paste Above** from the **Edit** menu. 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 **Paste Above** from the **Edit** menu. 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

Suppose now you want to copy some of the function panels from this driver to a new driver. Perform the following steps:

1. Execute **New, Function Tree (*.fp)** from the **File** menu. A new blank function tree window appears on the screen.
2. Execute **Instrument** from the **Create** menu.
3. Name the instrument `New Instrument` and type `new` in the prefix box. Click on **OK**.
4. Execute **Function Tree** from the **Window** menu and select the file called `mltcls`.
5. Position the selection on the item `Class 1`.
6. Execute **Copy** from the **Edit** menu.
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. Execute **Paste Below** from the **Edit** menu. `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 using the command **Edit Node** found in the **Edit** menu.

Change the name of the instrument driver and its prefix as follows:

1. Select `New Instrument`.
2. Execute **Edit Node** from the **Edit** menu. 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 appear at the top of the Function Tree in the Function Tree Editor as well as 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 in two ways.

- From the Function Tree Editor
- From a function panel

The following paragraphs describe the two ways to invoke the Function Panel Editor.

Invoking from the Function Tree Editor

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

1. Highlight the function corresponding to the function panel you want to edit.
2. Select **Edit Function Panel Window** from the **Edit** menu on the Function Tree Editor menu bar.

You can also 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 **Edit Function Panel Window** from the **Options** menu in 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.

The Function Panel Editor Menu Bar

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

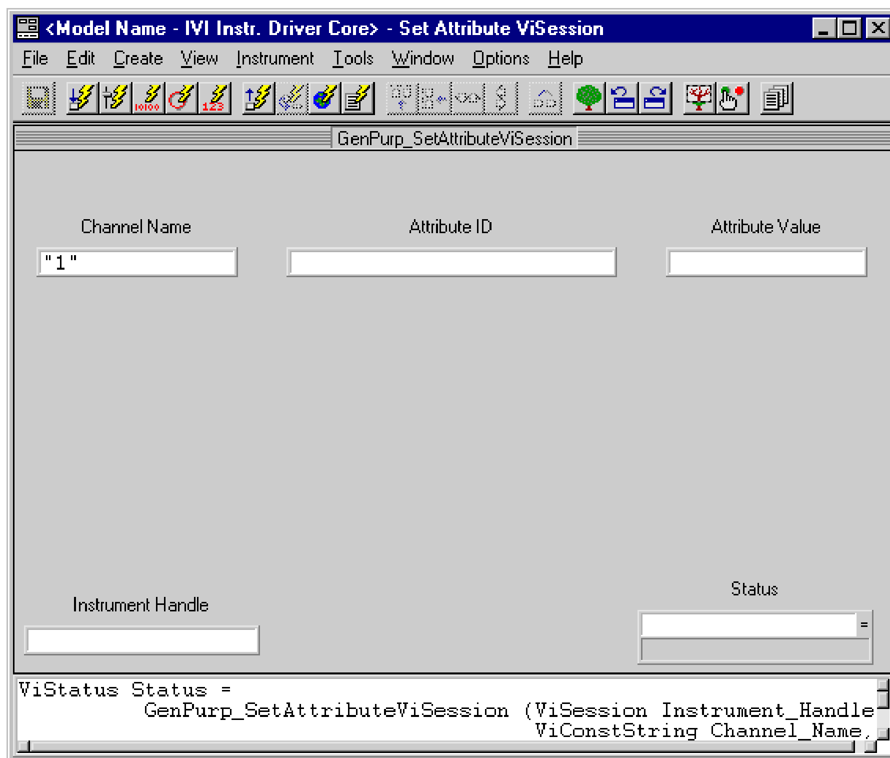


Figure 6-1. Function Panel Editor

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. Chapter 3, *Project Window*, of the *LabWindows/CVI User Manual*, gives 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 an `.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.

Note ***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.

Note ***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.

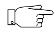
Cut Panel

The **Cut Panel** command removes the selected panel from the function panel window and places the panel, its controls, and all of the associated help information on the Clipboard.

Note ***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 of the associated help information to the Clipboard.

 **Note** ***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 listing 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. Chapter 2, *User Interface Editor Reference*, of the *LabWindows/CVI User Interface Reference Manual*, gives 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. Chapter 2, *User Interface Editor Reference*, of the *LabWindows/CVI User Interface Reference Manual*, gives 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 2, *User Interface Editor Reference*, of the *LabWindows/CVI User Interface Reference 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. Chapter 2, *User Interface Editor Reference*, of the *LabWindows/CVI User Interface Reference Manual*, gives 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**. Chapter 7, *Adding Help Information*, gives 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. See Chapter 5, *Function Tree Editor*, for more information on how to set the help style of the instrument driver. See Chapter 7, *Adding Help Information*, for more information about adding help to a function panel.

Create

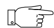
The **Create** menu lets you add controls to a function panel. There are nine control types in the **Create** menu: input, slide, binary, ring, numeric, output, return value, global variable, and message.

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 only have 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 which 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.***

Control Types

Use the **Create** menu to create the following control types for your function panels, as shown in Figure 6-2.

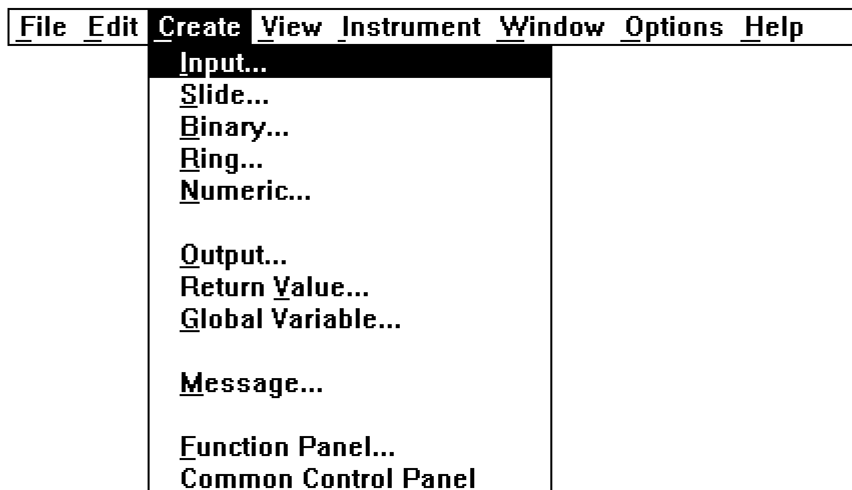


Figure 6-2. Control Types

Input

An *input control* accepts a variable name or value entered from the keyboard. When you select **Input** from the **Create** menu, the dialog box shown in Figure 6-3 appears.

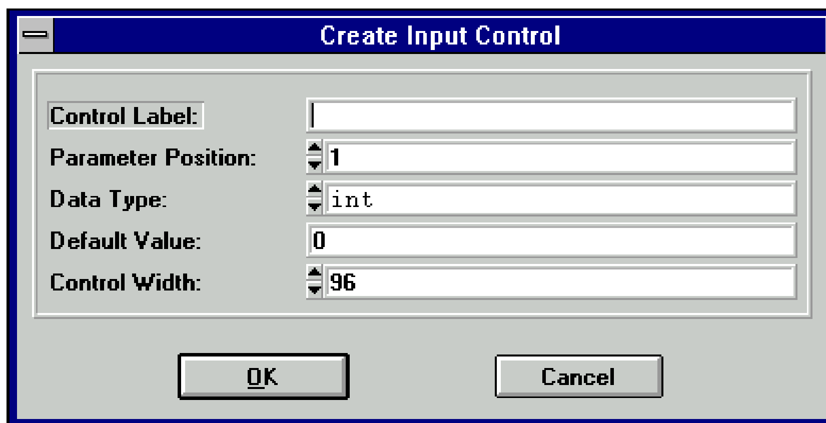


Figure 6-3. Create Input Control Dialog Box

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 the 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 **Slide** from the **Create** menu, the dialog box shown in Figure 6-4 appears.

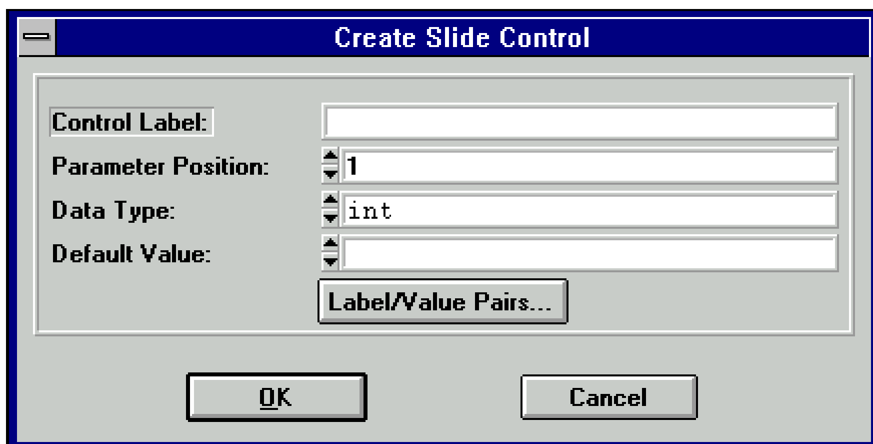


Figure 6-4. Create Slide Control Dialog Box

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 press the **Label/Value Pairs** button, the Edit Label/Value Pairs dialog box shown in Figure 6-5 appears.

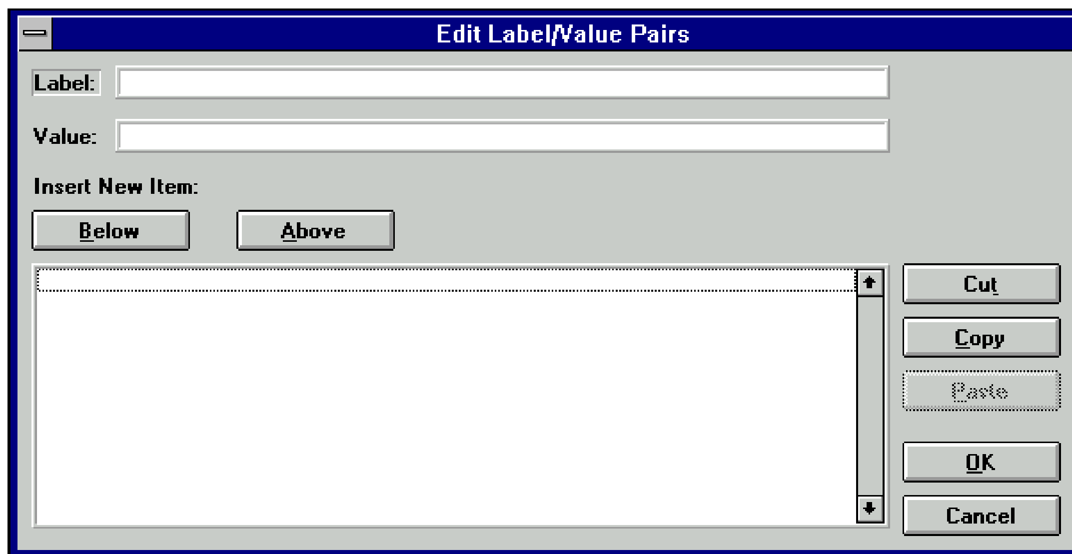


Figure 6-5. Edit Label/Value Pairs Dialog Box

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

Add a label to the slide control list as follows:

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.

The program 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** command 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 upon whether the control is in the up or down position. When you select **Binary** from the **Create** menu, the dialog box shown in Figure 6-6 appears.

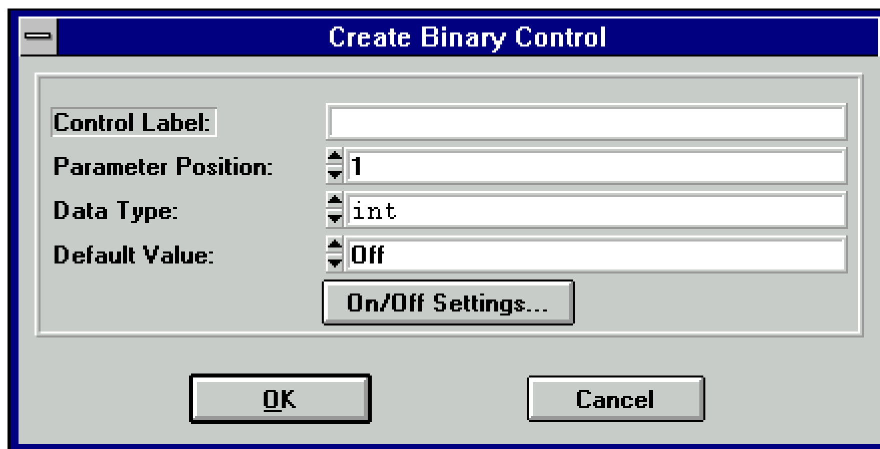


Figure 6-6. Create Binary Control Dialog Box

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 6-7 appears.

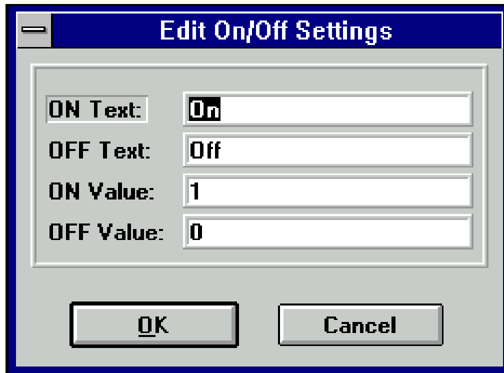


Figure 6-7. Edit On/Off Settings Dialog Box

- 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 6-8 appears.

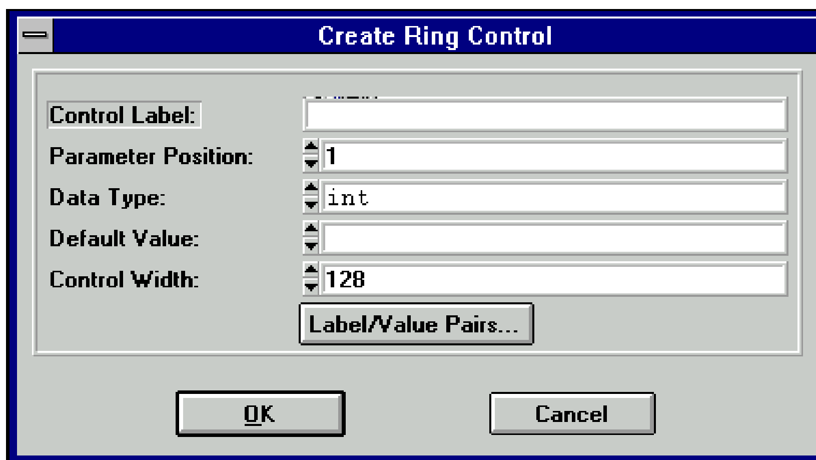


Figure 6-8. Create Ring Control Dialog Box

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 press the **Label/Value Pairs** button the Edit Label/Value Pairs dialog box shown in Figure 6-9 appears.

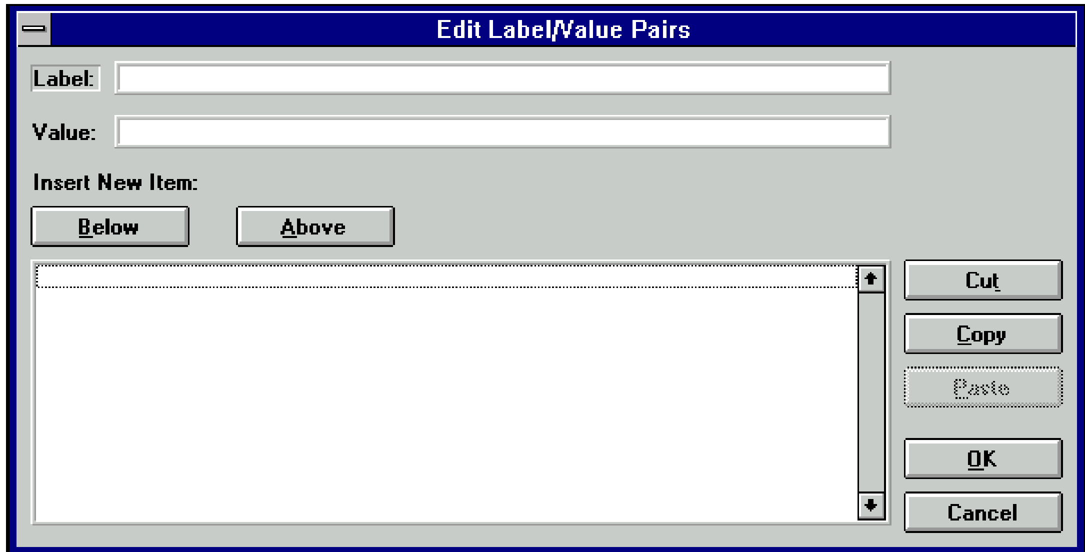


Figure 6-9. Ring Control Edit Label/Value Pairs Dialog Box

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

Add a label to the ring control list as follows:

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.

The program 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** command 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 **Numeric** from the **Create** menu, the dialog box shown in Figure 6-10 appears.

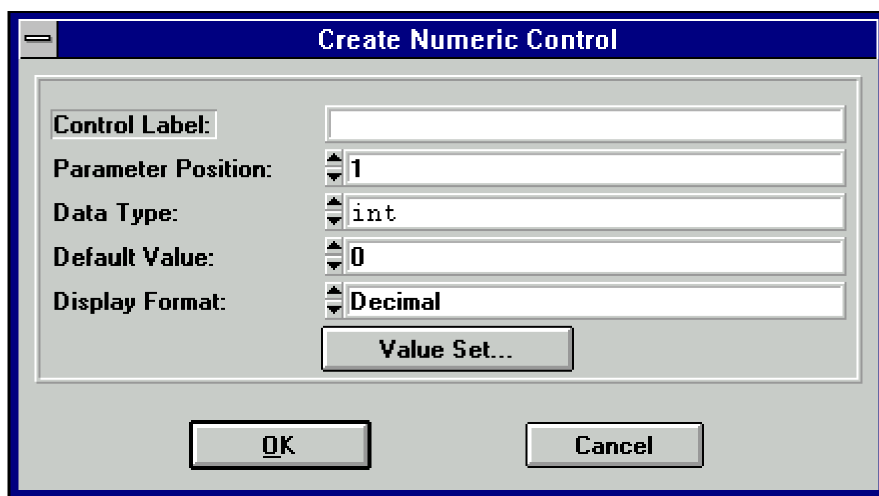


Figure 6-10. Create Numeric Control Dialog Box

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. For integer types, the options are Decimal, Hexadecimal, Octal, or ASCII. For double types, the options are Scientific and Floating Point.
- When you press the **Value Set** button the Edit Value Set dialog box shown in Figure 6-11 appears.

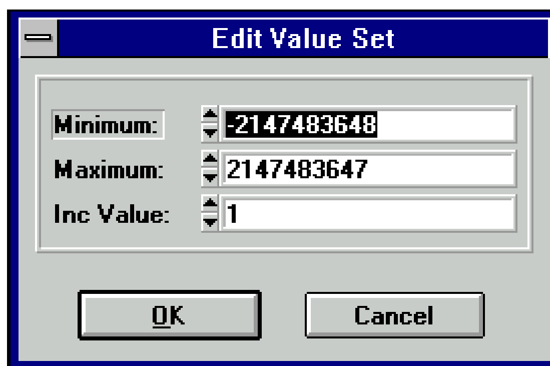


Figure 6-11. Edit Value Set Dialog Box

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 **Output** from the **Create** menu, the dialog box shown in Figure 6-12 appears.

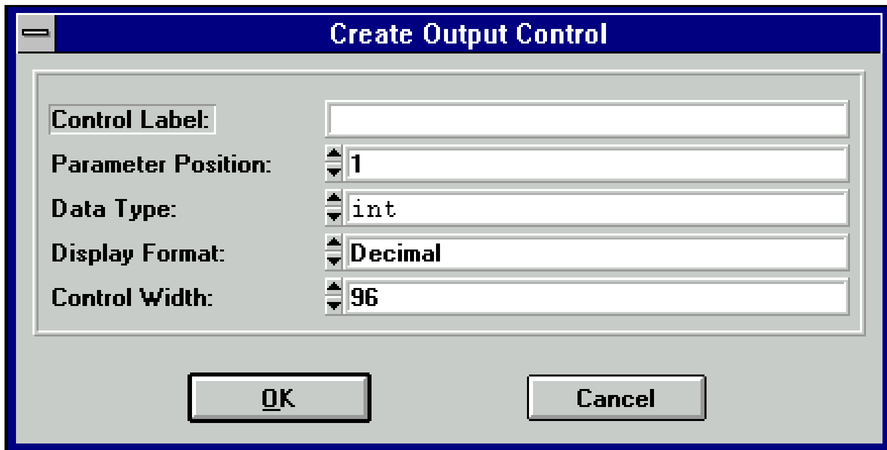


Figure 6-12. Create Output Control Dialog Box

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*.
- 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 **Return Value** from the **Create** menu, the dialog box shown in Figure 6-13 appears.

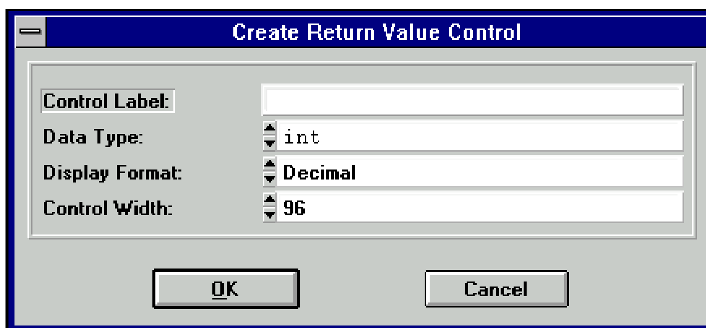


Figure 6-13. Create Return Value Control Dialog Box

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 **Global Variable** from the **Create** menu, the dialog box shown in Figure 6-14 appears.

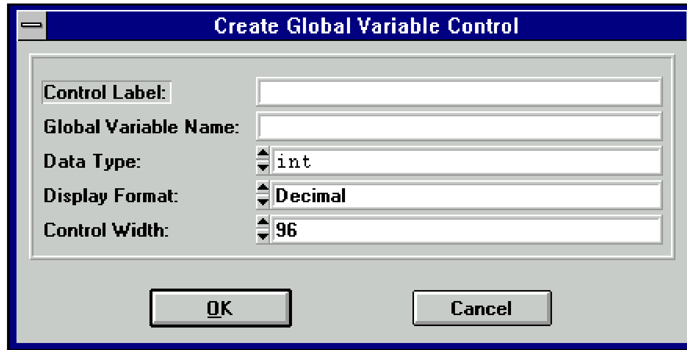


Figure 6-14. Create Global Variable Control Dialog Box

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 above 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 **Message** from the **Create** menu, a dialog box appears. Enter the desired text into the message text control and select the **OK** command 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. Chapter 5, *Using Function Panels*, in the *LabWindows/CVI User Manual*, gives 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. Chapter 5, *Function Tree Editor*, gives more information about the **Instrument** menu.

Tools

The Tools menu presents the following options:

- **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 an instrument driver's source files to reflect changes to function names in the function tree. This option is global to 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 Source For Function Panel** generates function definitions and declarations in your driver's source and header files. If a function definition already exists, 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.
- **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. Chapter 3, *Project Window*, in the *LabWindows/CVI User Manual*, gives more information about the **Window** menu.

Options

The **Options** menu lets you invoke the Function Tree Editor or operate the current function panel. You see the following items on the **Options** menu.

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 **Data Types** from the **Options** menu, the dialog box in Figure 6-15 appears.

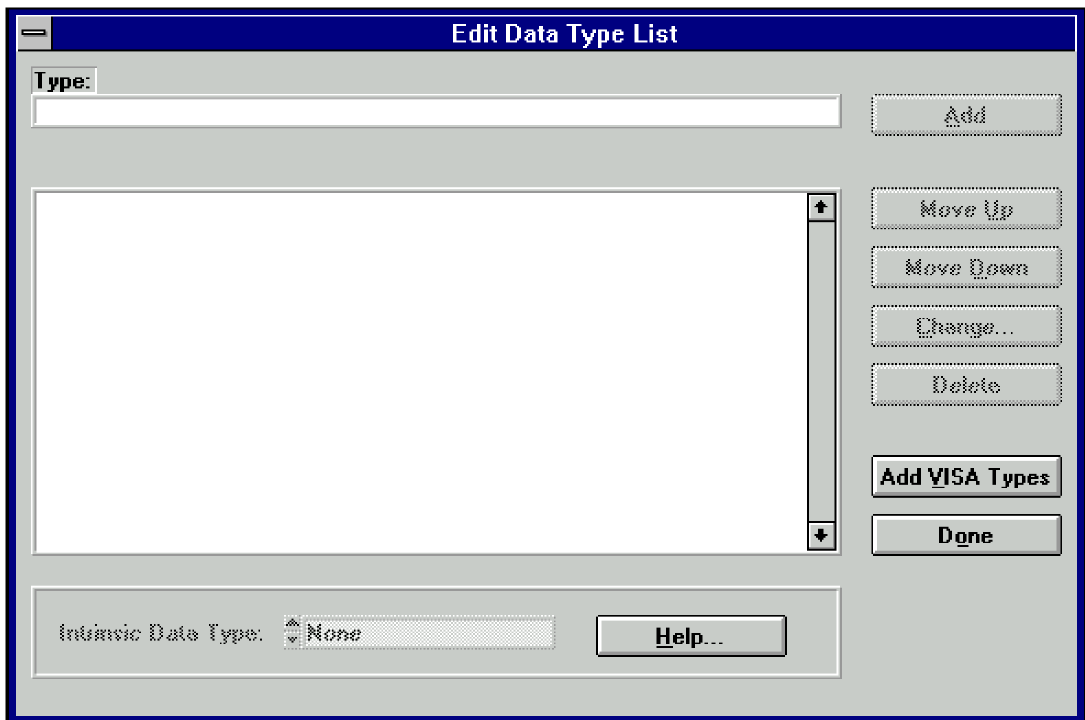


Figure 6-15. Edit Data Type List Dialog Box

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 toolbar.

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 moveable within a function panel editor window. Panels are never moveable 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.

Move a control using the keyboard as follows:

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 desired location. 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.

Move a control from one page to another as follows:

1. Select the desired control.
2. Execute **Cut Controls** from the **Edit** menu.
3. Move to the new function panel.
4. Execute **Paste** from the **Edit** menu.

Selecting Multiple Controls

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

Function Panel Editor Examples

The following examples teach you about creating and editing function panel windows, specifically the following:

- Creating a function panel window with one function panel
- Creating controls on a function panel
- Changing the type of a control
- Cutting and pasting controls on a panel and between panels

You create only function panels in this example without writing any code.

Example—Creating a Function Window

In this example you create a function panel. The example panel controls an oscilloscope with two channels, and configures the vertical sensitivity, coupling, and invert setting of the oscilloscope.

Follow these steps to create a new instrument and panel:

1. Execute the **Function Tree (*.fp)** option of the **New** command from the **File** menu.
2. Execute **Instrument** from the **Create** menu.
3. Enter `Function Panel Examples` as the Name and `panel` as the Prefix. Click on **OK**.
4. Execute **Function Panel Window** from the **Create** menu.
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 execute **Edit Function Panel Window** from the **Edit** menu. A new function panel window containing 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 **Binary** from the **Create** menu.
8. Complete the Create Binary Control dialog box as shown in Figure 6-16.

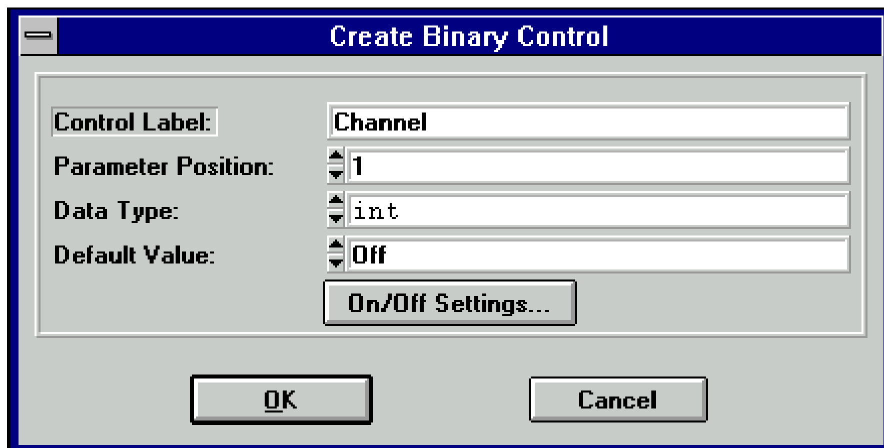


Figure 6-16. Channel Create Binary Control Dialog Box

9. Press the **On/Off Setting** button and complete the Edit On/Off Settings dialog box as shown in Figure 6-17. Position the control on the panel.

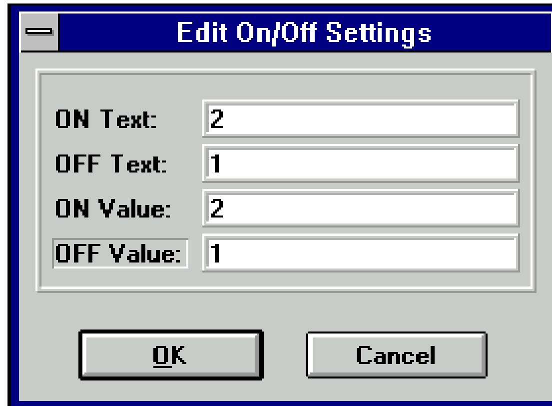


Figure 6-17. Channel Edit On/Off Settings Dialog Box

10. Execute **Input** from the **Create** menu.
11. Complete the Create Input Control dialog box as shown in Figure 6-18, and position the control on the panel.

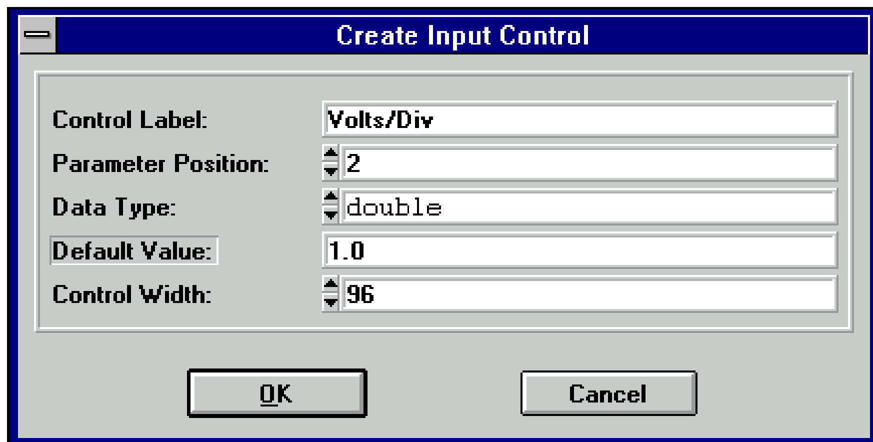


Figure 6-18. Volts/Div Create Input Control Dialog Box

12. Execute **Slide** from the **Create** menu.

13. Complete the Create Slide Control dialog box as shown in Figure 6-19.

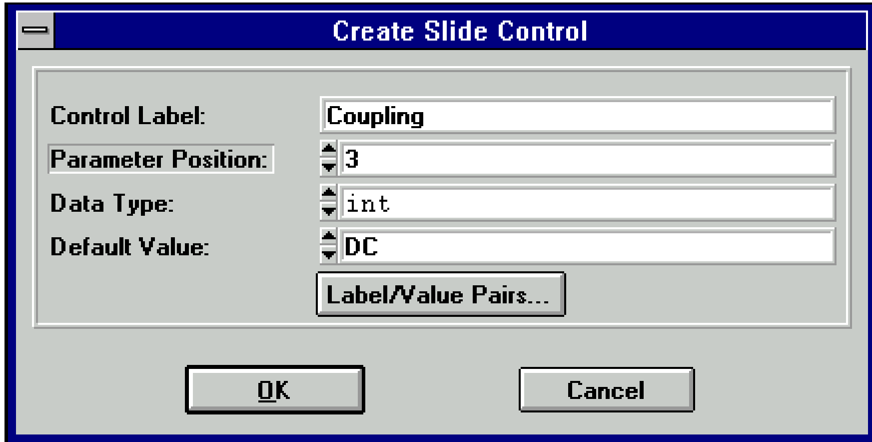


Figure 6-19. Coupling Create Slide Control Dialog Box

14. Press **Label/Value Pairs**, and complete the Edit Label/Value Pairs dialog box as shown in Figure 6-20, and position the control on the panel.

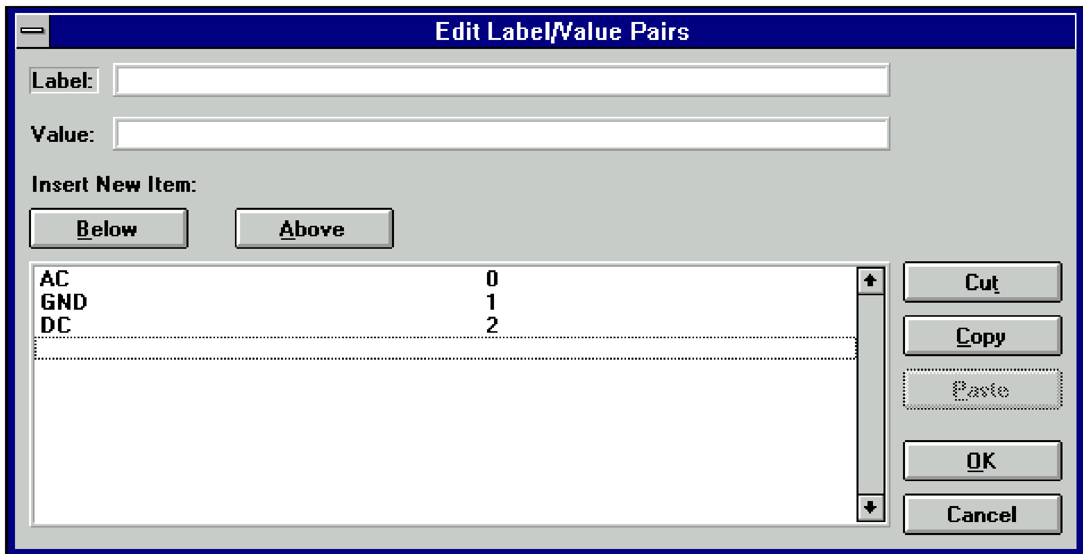


Figure 6-20. Coupling Edit Label/Value Pairs Dialog Box

15. Execute **Binary** from the **Create** menu.

16. Complete the Create Binary Control dialog box as shown in Figure 6-21.

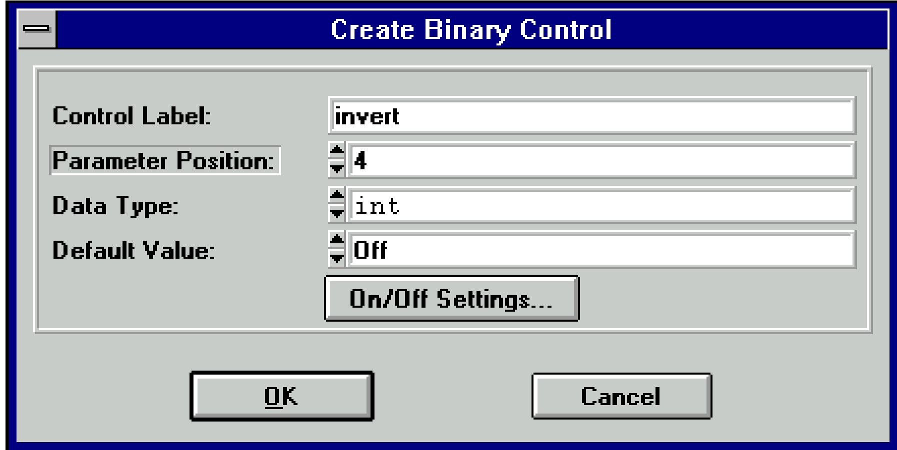


Figure 6-21. Invert Create Binary Control Dialog Box

17. Press the **On/Off Settings** button and complete the Edit On/Off Settings dialog box as shown in Figure 6-22. Position the control on the panel.

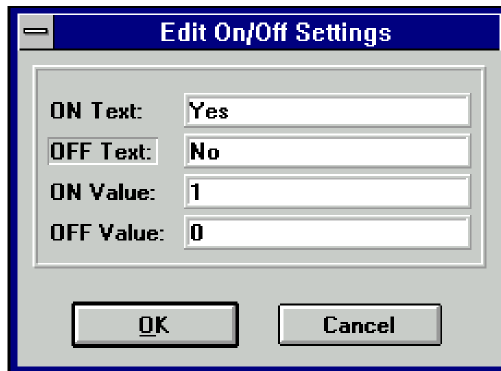


Figure 6-22. Invert Edit On/Off Settings Dialog Box

You now see the function panel shown in Figure 6-23.

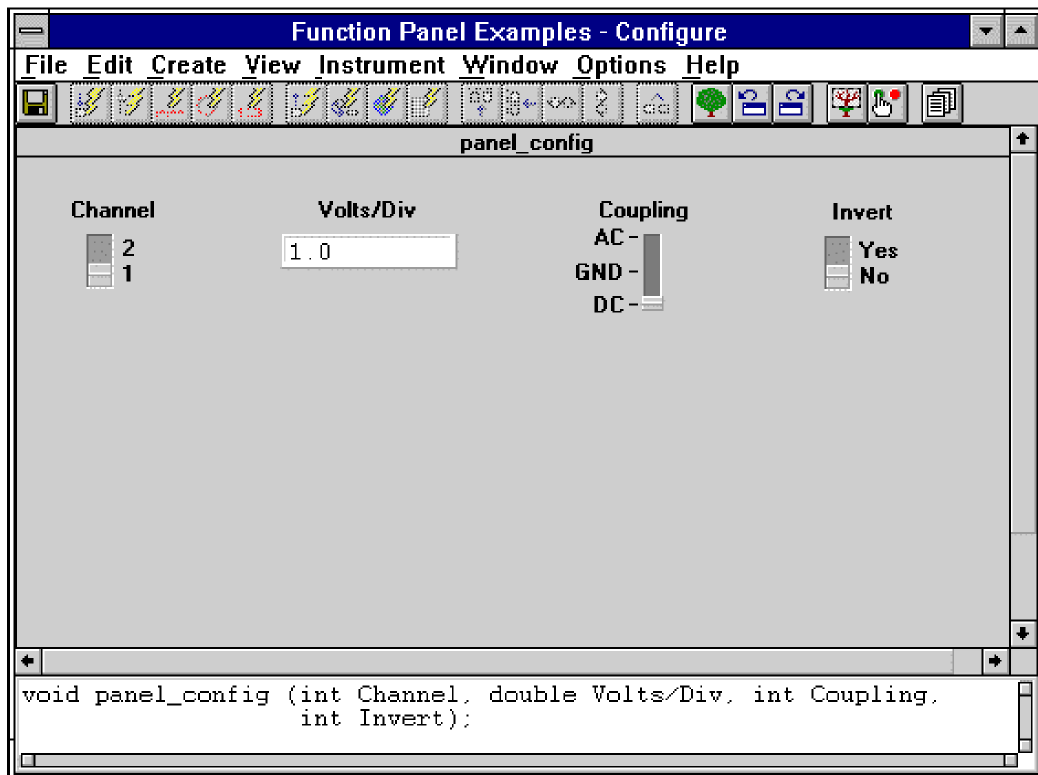


Figure 6-23. Function Panel Window

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. Execute **Change Control Type** from the **Edit** menu. The dialog box shown in Figure 6-24 appears.

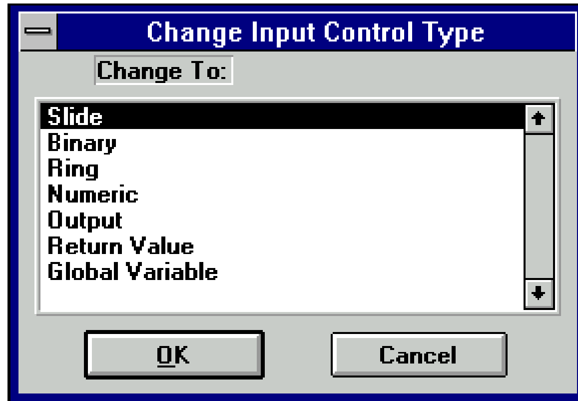


Figure 6-24. Change Input Control Type Dialog Box

3. Select Slide. The Edit Slide Control dialog box appears.
4. Select Label/Value Pairs. The Edit Label/Value Pairs dialog box appears.
5. Complete the dialog box as shown in Figure 6-25.

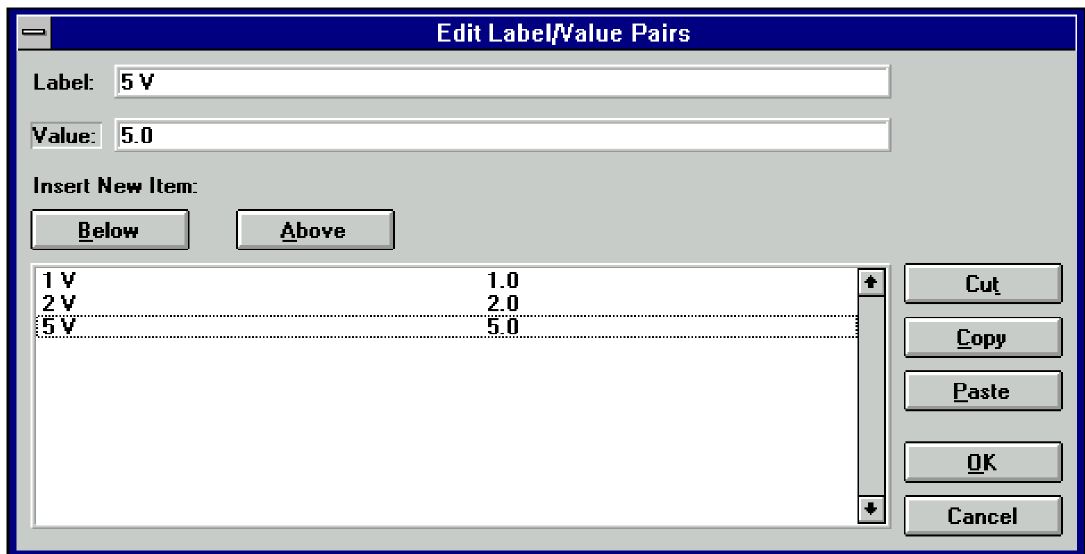


Figure 6-25. Volts/Div Edit Label/Value Pairs Dialog Box

After you complete the slide control dialog box, click on **OK** 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. Execute **Change Control Type** from the **Edit** menu.
3. Select Ring. The Edit Ring dialog box appears.
4. Press **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. Press **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. Perform 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. Execute **Control Help** from the **Edit** menu 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. Execute **Save .FP File** and then execute **Close** from the **File** menu of the Help Editor dialog box.
5. With the selection still on the Volts/Div control, execute **Copy Controls** from the **Edit** menu.
6. Execute **Paste** from the **Edit** menu.
7. With the selection on the new control, execute **Edit Control** from the **Edit** menu.
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`.

Create a new function panel and copy a control to the panel as follows:

1. Execute **Edit Function Tree** from the **Options** menu.
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. Execute **Edit Function Panel Window** from the **Edit** menu to return to the Configure panel.
5. Position the selection on the control Volts/Div 2.
6. Execute **Cut Controls** from the **Edit** menu.
7. Press <Ctrl-Page Down> to move to the New Panel function panel.
8. Execute **Paste** from the **Edit** menu.

The control appears on the panel. View the help information by selecting **Control Help** from the **Edit** menu. 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 vs. 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 that 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 of 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.

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. **Edit Help** in the **Edit** menu of the Function Tree Editor lets you add instrument and class help. **Function Help** and **Control Help** in the **Edit** menu of the Function Panel Editor let you add function panel and control help.

Add help information as follows:

1. From either the Function Tree Editor or the Function Panel Editor, select the item for which you want to add help information.
2. Choose **Edit Help**, **Function Help**, or **Control Help** from the **Edit** menu in the menu bar. The Help Editor window shown in Figure 7-1 appears.

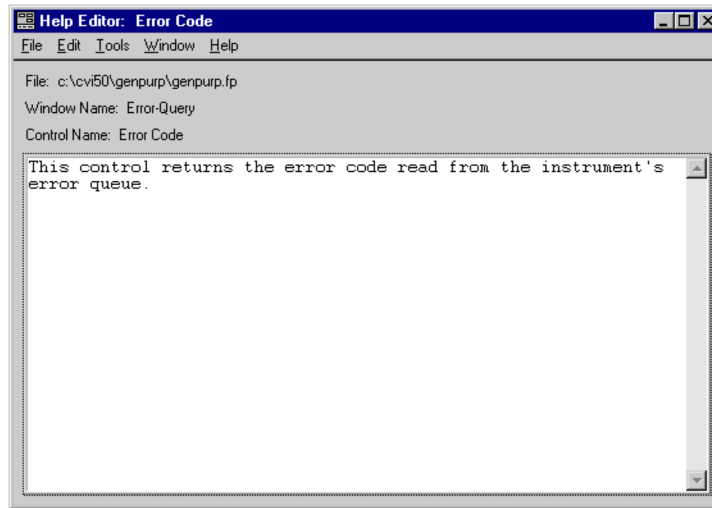


Figure 7-1. Help Editor Dialog Box

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. Chapter 3, *The Project Window*, in the *LabWindows/CVI User Manual*, gives 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 an `.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. Chapter 3, *Project Window*, in the *LabWindows/CVI User Manual*, gives 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. Follow these 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. Choose **Edit Help** from the **Edit** menu. 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. Follow these steps to add help information:

1. Select the class node in the function tree.
2. Choose **Edit Help** from the **Edit** menu 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 pertaining to a specific function panel by choosing **Function** from the **Help** menu 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. Follow these steps to add function panel help:

1. Activate the function panel.
2. Choose **Function Help** from the **Edit** menu 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. Type appropriate help text into the Help Editor window.

Chapter 5, *Function Tree Editor*, has 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 pertaining to a function panel window by selecting **Window** from the **Help** menu 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. Follow these steps to add function panel window help:

1. Choose **Window Help** from the **Edit** menu 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. Type appropriate help text into the Help Editor window.

Chapter 5, *Function Tree Editor*, gives 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 choosing **Control** from the **Help** menu 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.

Add help information for a function panel control as follows:

1. Select the control.
2. Choose **Control Help** from the **Edit** menu 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. Type appropriate help text into the Help Editor window.

Help Information Examples

The following examples teach you about creating and editing help information, specifically the following:

- Adding instrument and panel help information from the Function Tree Editor
- Adding panel and control help information from the Function Panel Editor
- Cutting and pasting help information between controls

You create only function trees and panels in this example, without writing any code.

Example—Adding Help Information in the Function Tree Editor

In this example, you add instrument and function class help information to a function tree. Follow these steps to create a new instrument and function tree:

1. Choose the **Function Tree (*.fp)** option of the **New** command from the **File** menu.
2. Choose **Instrument** from the **Create** menu.
3. Type `Help Information Examples` as the Name and `help` as the Prefix. Click on **OK**.
4. Choose **Class** from the **Create** menu.
5. Enter `Class 1` as the Name. Click on **OK**.
6. Select the line beneath the name `Class 1`.
7. Choose **Function Panel Window** from the **Create** menu.
8. Enter `Function 1` as the Name and `fun1` as the Function Name. Click on **OK**.

The new function tree appears in Figure 7-2.

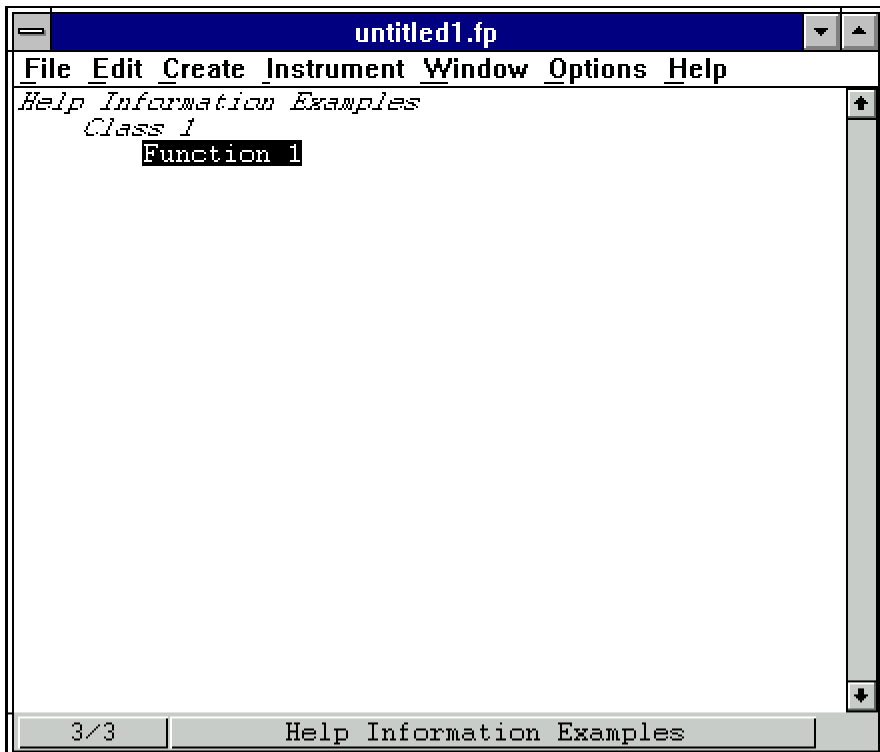


Figure 7-2. Sample Function Tree

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

Add help information to the top level of the tree as follows:

1. Select the name `Help Information Examples`.
2. Choose **Edit Help** from the **Edit** menu, 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. Choose **Save .FP File** and then choose **Close** from the **File** menu to save the text and remove the Help Editor window.

Add help information to `Class 1` as follows:

1. Select the name `Class 1`.
2. Choose **Edit Help** from the **Edit** menu 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. Choose **Save .FP File** and then choose **Close** from the **File** menu to save the text and remove the Help Editor window.

View the help information as follows;

1. Select `Help Information Examples` from the **Instrument** menu. The Select Function Panel dialog box appears.
2. Select `Class 1` and press Help to display the Class Help window.
3. Click **Instrument Help** to display the Instrument Help window.
4. Click **Done** to exit the Instrument Help window.
5. Click **Done** to exit the Class Help window.
6. Click **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.

To modify the help information for the function panel, perform the following steps from the Function Panel Editor:

1. Choose **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. Choose **Save .FP File** and then choose **Close** from the **File** menu to save the text and remove the Help Editor window.

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

Add a control to the current panel as follows:

1. Choose **Input** from the **Create** menu.
2. Enter `Input Control` for the Control Label.
3. Press **OK**.

Now add help information to the control:

1. Select the control and choose **Control Help** from the **Edit** menu. Alternatively, click the right mouse button on the control. The Help Editor window appears.
2. Enter the following text in the Help Editor window:

   `This control is an input control on the Function 1 function panel.`
3. Choose **Save .FP File** and then choose **Close** from the **File** menu to save the text and remove the Help Editor window.

You have now added help information to all possible locations. Select **Operate Function Panel** from the **Options** menu 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.

Copy the help information between controls on different panels as follows:

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 **Global Variable** from the **Create** menu.
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. Choose **Save .FP file** and then choose **Close** from the **File** menu to save the text.
7. Select the Status control. Choose **Copy Controls** from the **Edit** menu.
8. Press <Ctrl-Page Down> to display the `Function 2` function panel.
9. Choose **Paste** from the **Edit** menu. The Status control appears on the function panel.
10. Choose **Operate Function Panel** from the **Options** menu and view the help information. Notice that the help information stays with a control when you copy that control.

Copy the help text *without copying the control* as follows:

1. Choose **Edit Function Panel** Window from the **Options** menu.
2. Choose **Global Variable** from the **Create** menu.
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. Choose **Control Help** from the **Edit** menu or click the right mouse button on the control.
6. Select all of the text in the dialog box.
7. Choose **Copy** from the **Edit** menu.
8. Choose **Close** from the **File** menu.
9. Select the Error control.
10. Choose **Control Help** from the **Edit** menu or click the right mouse button on the control.

11. Choose **Paste** from the **Edit** menu. 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 any location, for example, for 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 may 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 Chapter 10, *Instrument Driver Examples*, 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

The instrument driver development wizard has the following templates. 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 of 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 multi-point 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.
Power Supply Template	Controls basic operations such as outputting DC and AC power and configuring the over-voltage and over-current protection. It also includes advanced features such as creating user-defined AC waveforms, 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 the above list.***

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, do the following.

- Modify the existing attributes and functions.
- Delete the attributes and functions that the instrument does not use.
- 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.

To modify an attribute do the following:

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, be sure to 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.

To modify an instrument driver function do the following:

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. Follow the procedures below to delete attributes and functions from your instrument driver.

To delete an attribute do the following:

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.***

To delete an instrument driver function do the following:

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. Follow the procedures below to add attributes and functions to your instrument driver.

To add a new attribute, do the following:

1. Create the attribute with the attribute editor.
2. Create a range table for the attribute from the Edit Attribute dialog box. Be sure to 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.

To add a new instrument driver function, do the following:

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 example below 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 when simulating. 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. The following table 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 illustrate 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 example below 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 the next two sections 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 can also 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. The 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's 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, the 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. The following table 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 IVI. To make developing user-callable functions easier, IVI 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;
   }
```

The important features of this standard function structure are:

- 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.
- 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 clean-up 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 Chapter 11, *IVI Library*, for detailed information on how to use the error macros.

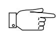
You fill in the standard function structure with the code that performs the operation for your instrument. You can divide the function code you write into the following steps.

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. If the function has output parameters and you are simulating, the function creates and returns simulated data.
6. Check the instrument status.
7. Unlock the session.
8. Return the value of the `error` variable.

The following example illustrates all of the steps from the above list.

```
/*****************************************************************
 * 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 if (Ivi_UseSpecificSimulation(vi))
      {
      /* 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` and the `Ivi_UseSpecificSimulation` 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 `F145_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 Chapter 11, *IVI Library*, 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. There are four types of parameters you must consider 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 parameter 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 when the following conditions are true:

- The `Ivi_UseSpecificSimulation` function returns `VI_TRUE`.
- 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 create and return simulated data for output parameters.

Checking the Instrument Status

In general, you call the *`Prefix`*`_CheckStatus` utility function prior to 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

Special considerations regarding the initialization functions are listed below.

- 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, then you must use "1" as the only valid channel string.

Close Functions

Special considerations regarding the close function are listed below.

- 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.

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
`ViChar[]`	A 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. The following table contains the macro definition for each platform.

Table 8-5. VISA I/O Library Macros

Macro	CVI Environment (Windows 3.1)	Outside the 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.

Listed below are important rules 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` which 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 returned from the VISA I/O libraries.
Negative values	Errors that are detected in an instrument driver such as the range-checking of function parameters or I/O errors reported by the VISA I/O libraries.

Refer to Chapter 11, *IVI Library*, for a complete list of IVI error codes that you can use in your driver.

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`

Table 8-7. Instrument Driver Completion and Warning Codes (Continued)

Completion Code	Description	Error Number
VI_WARN_NSUP_ERROR_QUERY	Error query not supported	0x3FFC0104L
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

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.

`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 Chapter 11, *IVI Library*, 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 (Virtual Instrument Software Architecture) 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 of 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, the Fluke 45 instrument driver function tree is shown in Figure 8-1.

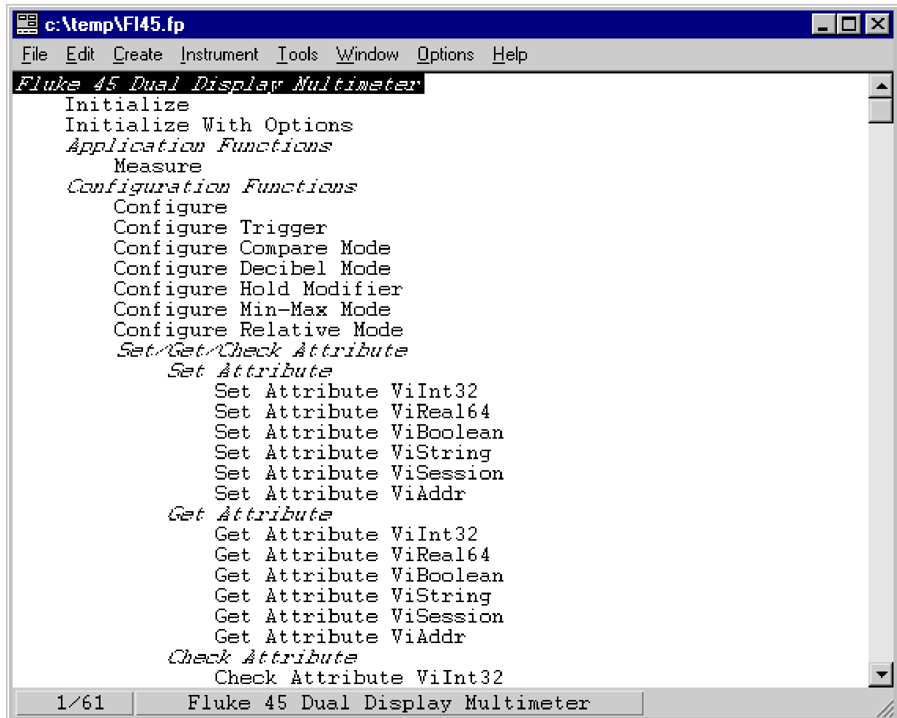


Figure 8-1. Fluke 45 Digital Multimeter Function Tree

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

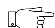
Writing useful documentation is an essential step in developing instrument drivers. Proper documentation helps the user understand the instrument driver and its functions. Instrument driver documentation consists of the following.

- 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.

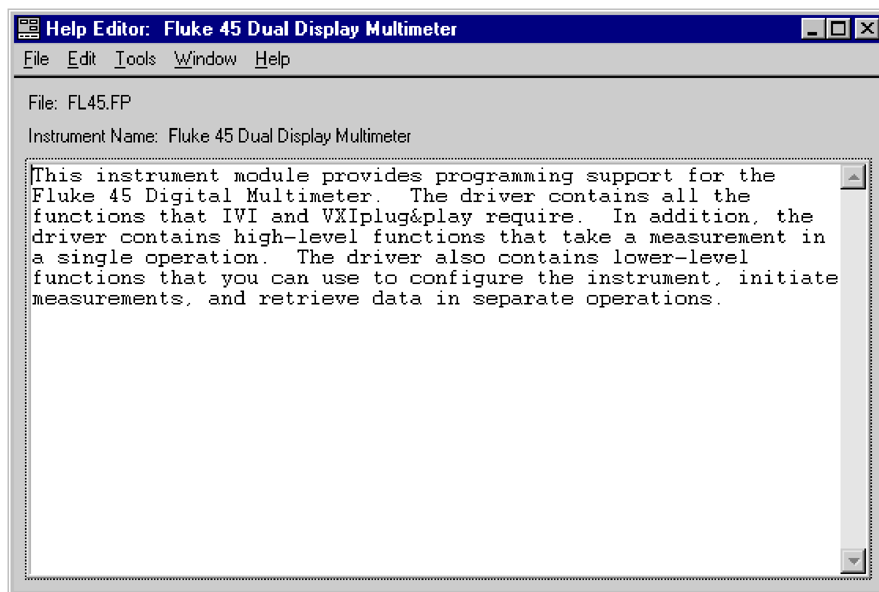


Figure 8-2. Fluke 45 Instrument Help

- *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.

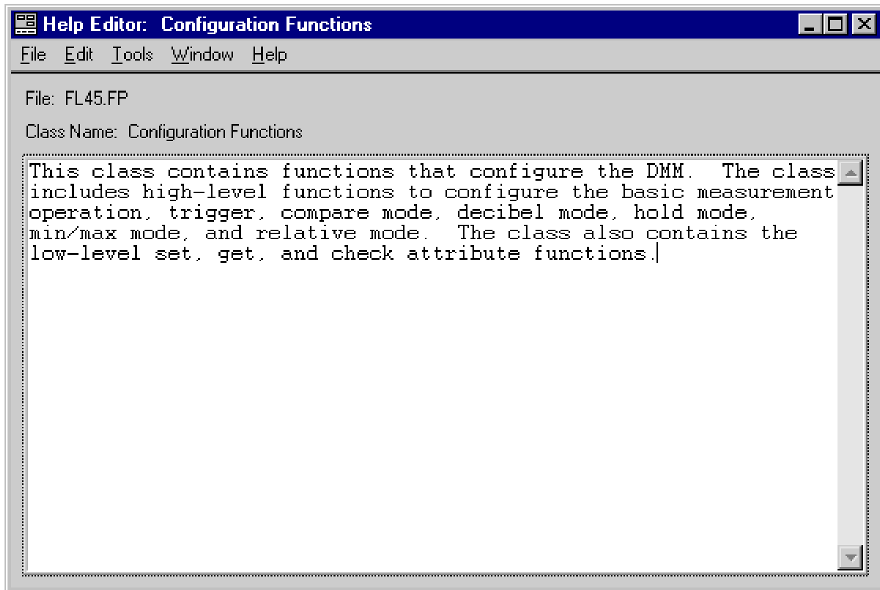


Figure 8-3. Fluke 45 Function Class Help

- *Function panel help* describes the function call. Figure 8-4 shows the function panel help from the Fluke 45 instrument driver.

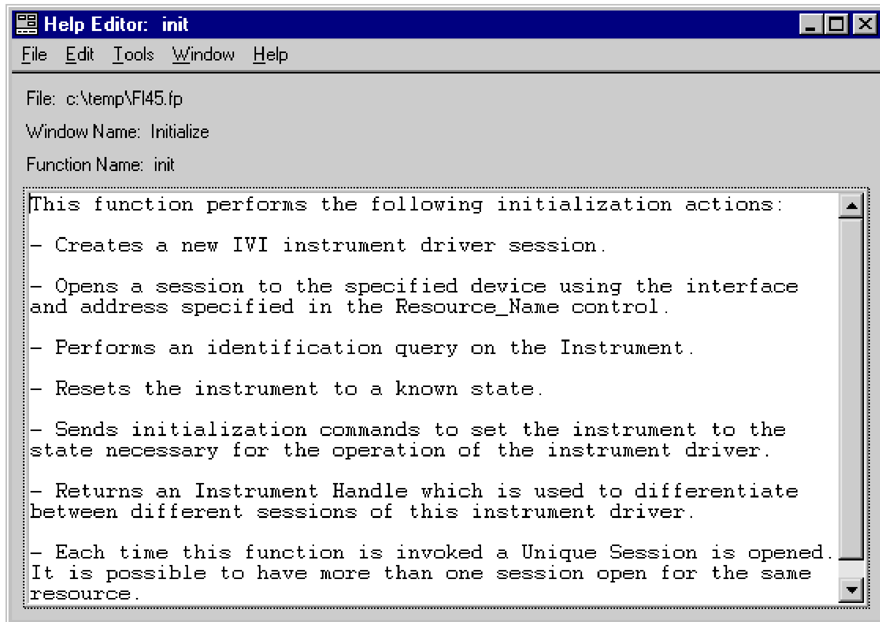


Figure 8-4. Fluke 45 Function Panel Help

- *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.

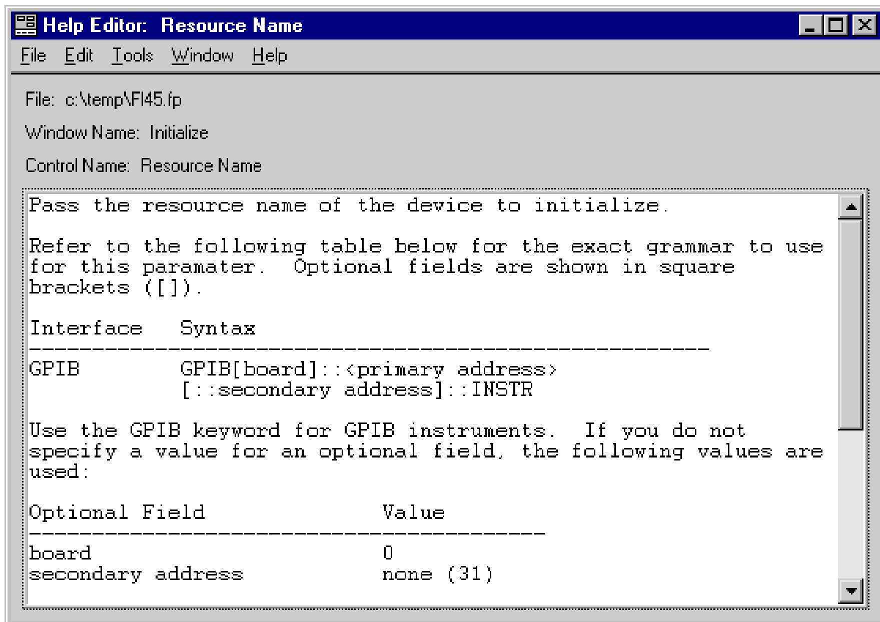


Figure 8-5. Fluke 45 Function Panel Control Help

- *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.

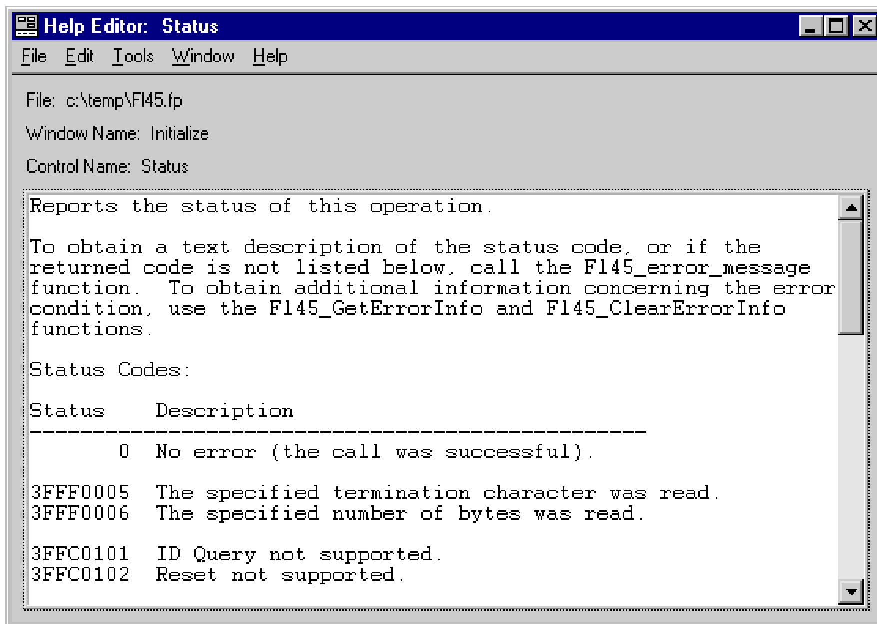


Figure 8-6. Fluke 45 Function Panel Status Control Help

The .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 using the **Generate Documentation** command in the **Options** menu of 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:

- ____ 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. You 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. Correctly 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. Correctly 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. Uses only VISA data types for function parameters.

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

 ____ 4. Declares 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

Required Instrument Driver Functions

This chapter contains information and function descriptions for required instrument driver functions.

Required Instrument Driver Function Overview

Each IVI instrument driver must implement a common set of sixteen functions. These functions are called the *required* functions.

The required functions are grouped into three categories:

Initialize/Close functions.

- *Prefix*_init
- *Prefix*_InitWithOptions
- *Prefix*_IviInit (for class driver)
- *Prefix*_close
- *Prefix*_IviClose (for class driver)

Utility functions.

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

Wrappers for IVI Library functions.

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

Except for *Prefix*_IviInit and *Prefix*_IviClose, all the required functions are user-callable. The instrument driver must have function panels for the fourteen user-callable functions and must have function prototypes for them in the include file. No function panels or prototypes are necessary for the *Prefix*_IviInit and *Prefix*_IviClose functions.

IVI instrument drivers must implement the *Prefix*_IviInit and *Prefix*_IviClose functions so that users can open an instrument driver session through either a class instrument driver or a specific instrument driver. In both class and specific instrument drivers, the *Prefix*_init function calls the *Prefix*_InitWithOptions function, which in turn calls *Prefix*_IviInit. Also, the *Prefix*_close function calls *Prefix*_IviClose. In a specific instrument driver, the *Prefix*_IviInit and *Prefix*_IviClose functions contain the bulk of the code that performs the initialization and closing operations for the particular instrument. In a class instrument driver, however, the *Prefix*_IviInit and *Prefix*_IviClose functions merely call the specific driver's *Prefix*_IviInit and *Prefix*_IviClose functions.

This chapter contains descriptions for the Initialize/Close functions and the Utility functions, including the implementation requirements. For the Wrapper functions, refer to the corresponding functions in the Chapter 11, *IVI Library*.

Note ***If you use the instrument driver developer wizard to create your instrument driver, the source file that the wizard generates contains the required functions with as much of the implementation source code as possible and with comments explaining how to complete the functions.***

Required Instrument Driver Function Reference

Prefix_init

```
ViStatus status = _VI_FUNC Prefix_init (ViRsrc resourceName,
                           ViBoolean idQuery, ViBoolean reset,
                           ViSession *vi);
```

Purpose

When you use an instrument driver, you must call the `Prefix_init` or `Prefix_InitWithOptions` function first. `Prefix_init` performs the following initialization actions:

- Creates a new IVI instrument driver session.
- Opens a session to a device using the interface and address you specify in the **resourceName** parameter.
- If you pass `VI_TRUE` for the **idQuery** parameter, the function queries the instrument for its ID and verifies that the instrument driver supports the particular instrument.
- If you pass `VI_TRUE` for the **reset** parameter, the function resets the instrument to a known state.
- Configures the I/O interface to the instrument.
- Sends default setup commands to the instrument to configure settings for the proper operation of the instrument driver. The default setup commands might include turning headers on or off or using the long or short form for queries.
- Returns a `ViSession` handle that you use to identify the session in all subsequent calls to the instrument driver.

The **resourceName** parameter is an resource descriptor string that identifies the I/O interface and configuration information specific to the interface. The following table shows the grammar of the resource descriptor strings for GPIB, VXI, and serial instruments. The grammar indicates optional parameters in square brackets ([ ]).

Interface	Grammar
GPIB	GPIB[*board*]::*primary address*[::*secondary address*][::INSTR]
VXI	VXI[*board*]::*VXI logical address*[::INSTR]
GPIB-VXI	GPIB-VXI[*board*][::*GPIB-VXI primary address*]::*VXI logical address*[::INSTR]
Serial	ASRL<*port*>::INSTR

Use the GPIB keyword for GPIB instruments. Use the VXI keyword for either embedded or MXIbus controllers. Use the GPIB-VXI keyword for a GPIB-VXI controller. Use the ASRL keyword for serial instruments. If you do not specify a board, the function uses 0. If you do not specify a secondary address, the function uses 31, which indicates no secondary address.

The following table contains example resource descriptors and their meanings.

Resource Descriptor	Meaning
GPIB::22::INSTR	GPIB board 0, primary address 22 no secondary address
GPIB::22::5::INSTR	GPIB board 0, primary address 22 secondary address 5
GPIB1::22::5::INSTR	GPIB board 1, primary address 22 secondary address 5
VXI::64::INSTR	VXI board 0, logical address 64
VXI1::64::INSTR	VXI board 1, logical address 64
GPIB-VXI::64::INSTR	GPIB-VXI board 0, logical address 64
GPIB-VXI1::64::INSTR	GPIB-VXI board 1, logical address 64
ASRL2::INSTR	COM port 2

Parameters

Input

Name	Type	Description

resourceName	`ViRsrc`	Instrument Description Examples:
		`VXI::5`
		`GPIB-VXI::128::INSTR`

idQuery	`ViBoolean`	if `VI_TRUE`, perform in-system verification if `VI_FALSE`, do not perform in-system verification
reset	`ViBoolean`	if `VI_TRUE`, perform reset operation if `VI_FALSE`, do not perform reset operation

Output

Name	Type	Description
vi	`ViSession`	Unique identifier for an IVI session

Return Values

Name	Type	Description
status	ViStatus	This is the operational return status. It returns either a completion code or an error code as follows.

Completion Code	Description	Set By
VI_SUCCESS	Session opened successfully	
VI_WARN_NSUP_ID_QUERY	Identification query not supported	Driver
VI_WARN_NSUP_RESET	Reset operation not supported	Driver

Error Codes	Description	Set By
VI_ERROR_FAIL_ID_QUERY	Instrument identification query failed	Driver
VI_ERROR_PARAMETER2	**idQuery** parameter out of range	Driver
VI_ERROR_PARAMETER3	reset parameter out of range	Driver
VI_ERROR_INV_RSRC_NAME	Invalid resource specified; parsing error	VISA
VI_ERROR_INV_ACC_MODE	Invalid access mode	VISA
VI_ERROR_RSRC_NFOUND	Insufficient location information or resource not present in the system	VISA
VI_ERROR_ALLOC	Insufficient system resources to open a session	VISA

Implementation Requirements

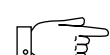
Report an error if the user passed VI_NULL for the address of the **vi** output parameter. Otherwise, call *Prefix*_InitWithOptions, passing an empty string for the **optionsString** parameter.

Prefix_InitWithOptions

```
ViStatus status = Prefix_InitWithOptions (ViRsrc resourceName,
                         ViBoolean idQuery, ViBoolean reset,
                         ViString optionsString, ViSession *vi);
```

Purpose

When you use an instrument driver, you must call the *Prefix*_init or *Prefix*_InitWithOptions function first. *Prefix*_InitWithOptions performs the same actions as *Prefix*_init, but it also allows you to set the initial the value of certain inherent IVI attributes.



Note ***Refer to*** *Prefix*_init ***for a description of the actions, parameters, and error codes that are common to*** *Prefix*_InitWithOptions ***and*** *Prefix*_init***. This function description documents only the features that are unique to*** *Prefix*_InitWithOptions**.**

You can use the **optionsString** parameter to set the initial value of certain IVI attributes for the session. The following table lists the attributes, their default initial values, and the name you use in this parameter to identify the attribute.

Name	Attribute	Default Value
RangeCheck	IVI_ATTR_RANGE_CHECK	VI_TRUE
QueryInstrStatus	IVI_ATTR_QUERY_INSTR_STATUS	VI_TRUE
Cache	IVI_ATTR_CACHE	VI_TRUE
Simulate	IVI_ATTR_SIMULATE	VI_FALSE
RecordCoercions	IVI_ATTR_RECORD_COERCIONS	VI_FALSE
DriverSetup	IVI_ATTR_DRIVER_SETUP	" "

If you pass VI_NULL or an empty string for this parameter, the session uses the default values. You can override the default values by assigning a value explicitly in a string you pass for this parameter.

The format of an assignment is, "*Name*=*Value*" where *Name* is the first column in the table above. *Prefix*_InitWithOptions interprets the *Name* and *Value* fields in a case-insensitive manner.

For the ViBoolean attributes, *Value* can be any of the following:

- To set the attribute to VI_TRUE, use VI_TRUE, True, or 1.
- To set the attribute to VI_FALSE, use VI_FALSE, False, or 0.

For DriverSetup, the valid values are entirely dependent on the particular instrument driver. Many drivers do not use the DriverSetup string.

To set multiple attributes, separate the assignments with commas.

You do not have to specify all of the attributes. If you do not specify one of the attributes, the session uses the default value.

Additional Error Codes

The following table lists the error codes that *Prefix*_InitWithOptions can return, excluding the error codes that appear in the function description for *Prefix*_init. All codes in the table pertain to errors in the **optionsString** parameter.

Error Code	Description
IVI_ERROR_MISSING_OPTION_NAME	**optionsString** contains an entry without a name. For example: "=True".
IVI_ERROR_MISSING_OPTION_VALUE	**optionsString** contains an entry without a value. For example: "Cache=".
IVI_ERROR_BAD_OPTION_NAME	**optionsString** contains an entry with an unknown option name.
IVI_ERROR_BAD_OPTION_VALUE	**optionsString** contains an entry with an unknown option value.

Implementation Requirements

Return an error if the user passes VI_NULL for the address of the **vi** output parameter.

Call Ivi_SpecificDriverNew to create the IVI session. Pass the **optionsString** parameter through to Ivi_SpecificDriverNew. Ivi_SpecificDriverNew validates the **optionsString** and sets the attributes to the values the user specifies in the string.

Then call *Prefix*_IviInit to perform the bulk of the initialization actions.

Upon success, return the IVI session you obtain from Ivi_SpecificDriverNew in the **vi** output parameter.

Upon failure, return VI_NULL in the **vi** output parameter, unless the user passed VI_NULL for the address of the parameter. Also, if Ivi_SpecificDriverNew succeeded, call Ivi_Dispose on the session handle that Ivi_SpecificDriverNew returned.

Prefix_IviInit

```
ViStatus status = Prefix_IviInit (ViRsrc resourceName, ViBoolean idQuery,
                                  ViBoolean reset, ViSession vi);
```

Purpose

Contains the bulk of the code to initialize an instrument driver session. *Prefix*_InitWithOptions calls *Prefix*_IviInit after it calls Ivi_SpecificDriverNew to create the IVI session. Also, the *Prefix*_IviInit functions in class drivers call the *Prefix*_IviInit functions in specific drivers. This allows a user to open an IVI session to an instrument from a class driver or a specific driver.

Note ***Refer to*** *Prefix*_init ***for a description of the initialization actions and parameters. This function description documents only the implementation requirements for*** *Prefix*_IviInit**.**

Implementation Requirements

Call Ivi_BuildChannelTable to specify the set of valid channel strings for the instrument. If the instrument does not have multiple channels, specify "1" for the channel strings.

Create all of the attributes you want to use, excluding the inherent IVI attributes. If you are developing your driver according to a class definition, create the class attributes that you want to use. Generally, you should create the attributes in an internal *Prefix*_InitAttributes function. If you use the instrument driver developer wizard, the wizard creates this function for you. You must have this function in your source file to be able to use the attribute editor.

If simulation is disabled, create the I/O session, and set the IVI_ATTR_IO_SESSION attribute. Configure the I/O interface. Set the IVI_ATTR_SUPPORTS_WR_BUF_OPER_MODE attribute to VI_TRUE if the instrument can accept commands and data that the driver sends to it using VISA buffered I/O.

If the **reset** parameter is VI_TRUE, call *Prefix*_reset. Otherwise, if not simulating, send the default setup commands to the instrument. It is best to have an internal *Prefix*_DefaultInstrSetup function for this purpose because *Prefix*_reset also sends the default setup commands.

If the **idQuery** parameter is VI_TRUE and simulation is disabled, verify the identity of the instrument if possible. For IEEE 488.2 compatible instruments, send use the *IDN query. For VXI register-based instruments, check the manufacturer ID and model number. Return an error if the instrument is not one that the driver supports.

Finally, check the status of the instrument. Typically, you do this through by calling an internal *Prefix*_CheckStatus function. The instrument driver developer wizard generates the internal *Prefix*_CheckStatus function for you. The internal *Prefix*_CheckStatus function calls the check status callback if status checking is enabled, simulation is disabled, and the driver has performed instrument I/O since the last time it queried the instrument status.

If a failure occurs after you open the I/O session, close the I/O session and set the IVI_ATTR_IO_SESSION parameter back to 0.

Prefix_close

```
ViStatus status = _VI_FUNC Prefix_close (ViSession vi);
```

Purpose

When you are finished using an instrument driver session, you must call the *Prefix*_close. *Prefix*_close performs the following actions.

- Closes the instrument I/O session.
- Destroys the IVI session and all of its attributes.
- Deallocates any memory resources used by the IVI session.

You might also want to put the instrument into an idle state before closing the I/O session. For example, a switch driver might disconnect all switches.

Parameter

Input

Name	Type	Description
vi	`ViSession`	Unique identifier for an IVI session.

Return Values

Name	Type	Description
status	`ViStatus`	This is the operational return status. It returns either a completion code or an error code as follows.

Completion Code	Description	Set By
`VI_SUCCESS`	Session closed successfully	

Error Code	Description	Set By
`VI_ERROR_INV_SESSION`	The given session is invalid	VISA
`VI_ERROR_CLOSING_FAILED`	Unable to deallocate the previously allocated data structures corresponding to this session	VISA

Implementation Requirements

First call `Ivi_LockSession` to lock the IVI session.

Call *`Prefix`*`_IviClose`.

Call `Ivi_UnlockSession` to unlock the IVI session.

Finally, call `Ivi_Dispose` on the IVI session. It is very important to unlock the IVI session *before* calling `Ivi_Dispose`.

`Ivi_Dispose` destroys the instrument driver session and all of its attributes. It also deallocates any memory blocks that you associated with the session when you called `Ivi_Alloc` or `Ivi_RangeTableNew`. *`Prefix`*`_IviClose` performs all the other clean-up operations.

Prefix_IviClose

```
ViStatus status = Prefix_IviClose (ViSession vi);
```

Purpose

Performs all the clean-up operations for closing an instrument driver session, except for destroying the IVI session. The `Prefix_close` functions in both specific and class instrument drivers call `Prefix_IviClose`. This allows a user to open and close an IVI session to an instrument from a class driver or a specific driver.

Implementation Requirements

Set the instrument to an idle state, if that is appropriate.

Close the I/O session and set `IVI_ATTR_IO_SESSION` to 0.

If you have any hidden `ViAddr` attributes that point to memory that you dynamically allocated, free the memory.

Prefix_reset

```
ViStatus status = _VI_FUNC Prefix_reset (ViSession vi);
```

Purpose

Places the instrument in a known state. In an IEEE 488.2 instrument, the *Prefix*_reset function sends the command string "*RST" to the instrument. *Prefix*_reset also sends the default setup commands to the instrument to configure settings for the proper operation of the instrument driver. You can either call the *Prefix*_reset function separately, or you can select it to be called from the *Prefix*_init function.

Parameter

Input

Name	Type	Description
vi	ViSession	Unique identifier for an IVI session.

Return Values

Name	Type	Description
status	ViStatus	This is the operational return status. It returns either a completion code or an error code as follows.

Completion Code	Description	Set By
VI_SUCCESS	Reset successful	
VI_WARN_NSUP_RESET	Reset operation not supported	Driver

Error Code	Description	Set By
VI_ERROR_INV_SESSION	The given session is invalid	VISA
VI_ERROR_TMO	Timeout expired before operation completed	VISA
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer	VISA
VI_ERROR_BERR	Bus error occurred during transfer	VISA

Error Code	Description	Set By
VI_ERROR_NCIC	The interface associated with the given **vi** is not currently the controller in charge	VISA
VI_ERROR_NLISTENERS	No Listeners condition is detected	VISA

Implementation Requirements

Call `Ivi_LockSession` to lock the IVI session.

If simulation is disabled, send the reset command to the instrument. On IEEE 488.2 instruments, you do this by sending the `*RST` command. Remember to call `Ivi_SetNeedToCheckStatus` with `VI_TRUE` before you perform the instrument I/O.

Send the default setup commands to the instrument. It is best to have an internal *Prefix*_DefaultInstrSetup function for this purpose. *Prefix*_IviInit must call *Prefix*_DefaultInstrSetup when it does not call *Prefix*_Reset.

Finally, call `Ivi_UnlockSession` to unlock the IVI session.

Be sure to document the state in which the *Prefix*_reset function places the instrument. Include the information in the function panel help for the *Prefix*_reset function.

Prefix_self_test

```
ViStatus status = _VI_FUNC Prefix_self_test(ViSession vi,
                        ViInt16 * testResult, ViChar testMessage[]);
```

Purpose

Causes the instrument to perform a self-test. `Prefix_self_test` waits for the instrument to complete the test. It then queries the instrument for the results of the self test and returns the results to the user.

Parameter

Input

Name	Type	Description
vi	`ViSession`	Unique identifier for an IVI session.

Output

Name	Type	Description
testResult	`ViInt16`	Numeric result from self-test operation 0 = no error (test passed)
testMessage	`ViChar` array	Self-test status message

Return Values

Name	Type	Description
status	`ViStatus`	This is the operational return status. It returns either a completion code or an error code as follows.

Completion Code	Description	Set By
`VI_SUCCESS`	Self test successful	
`VI_WARN_NSUP_SELF_TEST`	Self-test operation not supported	Driver

Error Code	Description	Set By
VI_ERROR_INV_RESPONSE	Error interpreting instrument response	Driver
VI_ERROR_INV_SESSION	The given session is invalid	VISA
VI_ERROR_TMO	Timeout expired before operation completed	VISA
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer	VISA
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer	VISA
VI_ERROR_BERR	Bus error occurred during transfer	VISA
VI_ERROR_NCIC	The interface associated with the given **vi** is not currently the controller in charge	VISA
VI_ERROR_NLISTENERS	No Listeners condition is detected	VISA

Implementation Requirements

Report an error if the user passes `VI_NULL` for either of the output parameters.

Call `Ivi_LockSession` to lock the IVI session.

If simulation is disabled, send the self-test command to the instrument. On IEEE 488.2 instruments, you do this by sending the *TST command. Then read the results from the instrument into the **testResult** and **testMessage** output parameters.

If simulation is enabled but `Ivi_UseSpecificSimulation` returns `VI_TRUE`, set the **testResult** output parameter to 0, and copy `"No error."` into the **testMessage** output parameter.

Call your internal *Prefix*_CheckStatus function.

Finally, call `Ivi_UnlockSession` to unlock the IVI session.

If the instrument cannot perform a self-test operation, you should still include the function in the driver and return the warning `VI_WARN_NSUP_SELF_TEST`.

Prefix_error_query

```
ViStatus status = _VI_FUNC Prefix_error_query (ViSession vi,
                           ViInt32 * errCode, ViChar errMessage[]);
```

Purpose

Queries the instrument and returns the instrument-specific error information.

Generally, you call this function after another function in the instrument driver returns the `IVI_ERROR_INSTR_SPECIFIC` error code. The driver returns `IVI_ERROR_INSTR_SPECIFIC` when the instrument's status register indicates that the instrument's error queue is not empty. `Prefix_error_query` extracts the first error out of the instrument's error queue. For instruments that have status registers but no error queue, the driver simulates an error queue in software.

Parameter

Input

Name	Type	Description
vi	`ViSession`	Unique identifier for an IVI session.

Output

Name	Type	Description
errCode	`ViInt32`	Instrument error code
errMessage	`ViChar` array	Instrument error message

Return Values

Name	Type	Description
status	`ViStatus`	This is the operational return status. It returns either a completion code or an error code as follows.

Completion Code	Description	Set By
`VI_SUCCESS`	Error query successful	
`VI_WARN_NSUP_SELF_TEST`	Self-test operation not supported	Driver

Error Code	Description	Set By
VI_ERROR_INV_RESPONSE	Error interpreting instrument response	Driver
VI_ERROR_INV_SESSION	The given session is invalid	VISA
VI_ERROR_TMO	Timeout expired before operation completed	VISA
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer	VISA
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer	VISA
VI_ERROR_BERR	Bus error occurred during transfer	VISA
VI_ERROR_NCIC	The interface associated with the given **vi** is not currently the controller in charge	VISA
VI_ERROR_NLISTENERS	No Listeners condition is detected	VISA

Implementation Requirements

Report an error if the user passes `VI_NULL` for either of the output parameters.

If the instrument has status registers and an error queue, do the following.

1. Call `Ivi_LockSession` to lock the IVI session.
2. If simulation is disabled, send the error query command to the instrument. On IEEE 488.2 instruments, you do this by sending the `:SYST:ERR?` command. Then read the results from the instrument into the **errCode** and **errMessage** output parameters.
3. If simulation is enabled but `Ivi_UseSpecificSimulation` returns `VI_TRUE`, set the **errCode** output parameter to 0, and copy `"No error."` into the **errCode** output parameter.
4. Call `Ivi_UnlockSession` to unlock the IVI session.

Some instruments have status registers but no error queue. The act of reading the status registers clears the error information. For such instruments, the check status callback must call `Ivi_QueueInstrSpecificError` to add the error information to a software error queue whenever the status registers indicate an error. The `*Prefix*_error_query` function must do the following:

1. Call `Ivi_LockSession` to lock the IVI session.
2. Call `Ivi_InstrSpecificErrorQueueSize` to determine if the software error queue is empty. If it is empty, call the check status callback and then `Ivi_InstrSpecificErrorQueueSize` again to determine if the software error queue is still empty.
3. If the software queue is not empty, call `Ivi_DequeueInstrSpecificError` to extract the error information into the **errCode** and **errMessage** parameters.
4. Otherwise, set the **errCode** output parameter to 0, and copy `"No error."` into the **errMessage** output parameter.
5. Call `Ivi_UnlockSession` to unlock the IVI session.

If the instrument cannot perform an error query, you should still include the function in the driver and return the warning `VI_WARN_NSUP_ERROR_QUERY`.

Prefix_error_message

```
ViStatus status = _VI_FUNC Prefix_error_message (ViSession vi,
                            ViStatus errCode, ViChar errMessage[]);
```

Purpose

Translates the error return value from an instrument driver function to a user-readable string.

Parameter

Input

Name	Type	Description
vi	`ViSession`	Unique identifier for an IVI session. Can be `VI_NULL`
errCode	`ViStatus`	Instrument driver error code

Output

Name	Type	Description
errMessage	`ViChar` array	Instrument driver error message

Return Values

Name	Type	Description
status	`ViStatus`	This is the operational return status. It returns either a completion code or an error code as follows.

Completion Code	Description	Set By
`VI_SUCCESS`	Error message successful	
`VI_WARN_UNKNOWN_STATUS`	The status code passed to the operation could not be interpreted	Driver

Implementation Requirements

Your *Prefix*_error_message function must accept a value of VI_NULL for the **vi** input parameter. This allows the user to call the function even when *Prefix*_init or *Prefix*_InitWithOptions fails. On the other hand, report an error if the user passes VI_NULL for the address of the **errMessage** output buffer.

If your driver defines its own error codes, define a static string/value table containing the error codes and message strings. Use the IviStringValueTable typedef in ivi.h. Terminate the table with an entry that has VI_NULL in both fields.

If the **vi** parameter is not VI_NULL, call Ivi_LockSession to lock the IVI session.

Call the Ivi_GetSpecificDriverStatusDesc function. Pass the address of your error string/value table as the last parameter. If your driver does not have its own error codes, pass VI_NULL for the last parameter.

If the **vi** parameter is not VI_NULL, call Ivi_UnlockSession to unlock the IVI session.

Prefix_revision_query

```
ViStatus status = _VI_FUNC Prefix_revision_query (ViSession vi,
                            ViChar driverRev[], ViChar instrRev[])
```

Purpose

Obtains the following information:

- The revision of the instrument driver
- The firmware revision of the instrument you are currently using

Parameter

Input

Name	Type	Description
vi	`ViSession`	Unique logical identifier to a session with an instrument

Output

Name	Type	Description
driverRev	`ViChar` array	Instrument driver revision
instrRev	`ViChar` array	Instrument firmware revision

Return Values

Name	Type	Description
status	`ViStatus`	This is the operational return status. It returns either a completion code or an error code as follows.

Completion Code	Description	Set By
`VI_SUCCESS`	Revision query successful	
`VI_WARN_NSUP_REV_QUERY`	Revision query not supported	Driver

Error Code	Description	Set By
VI_ERROR_INV_RESPONSE	Error interpreting instrument response	Driver
VI_ERROR_INV_SESSION	The given session is invalid	VISA
VI_ERROR_TMO	Timeout expired before operation completed	VISA
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer	VISA
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer	VISA
VI_ERROR_BERR	Bus error occurred during transfer	VISA
VI_ERROR_NCIC	The interface associated with the given vi is not currently the controller in charge	VISA
VI_ERROR_NLISTENERS	No Listeners condition is detected	VISA

Implementation Requirements

Report an error if the user passes `VI_NULL` for either of the output parameters.

Call `Ivi_LockSession` to lock the IVI session.

Call `Ivi_GetAttributeViString` on the `IVI_ATTR_DRIVER_REVISION` attribute to get the driver revision string into the **driverRev** output parameter.

If simulation is disabled, send the revision query command to the instrument. On IEEE 488.2 instruments, you do this by sending the *IDN command. Then read the results from the instrument into **instrRev** output parameter. Remember to call `Ivi_SetNeedToCheckStatus` with `VI_TRUE` before you perform the instrument I/O.

If simulation is enabled but `Ivi_UseSpecificSimulation` returns `VI_TRUE`, copy `"No revision information available while simulating."` into the **instrRev** output parameter.

Call your internal *Prefix*`_CheckStatus` function.

Call `Ivi_UnlockSession` to unlock the IVI session.

10

Instrument Driver Examples

This chapter shows you how to create an IVI instrument driver. The examples in this chapter can serve as models for your instrument driver development.

This chapter illustrates the following procedures:

- 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 upon 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 the **Create IVI Instrument Driver** command from the **Tools** menu. 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.

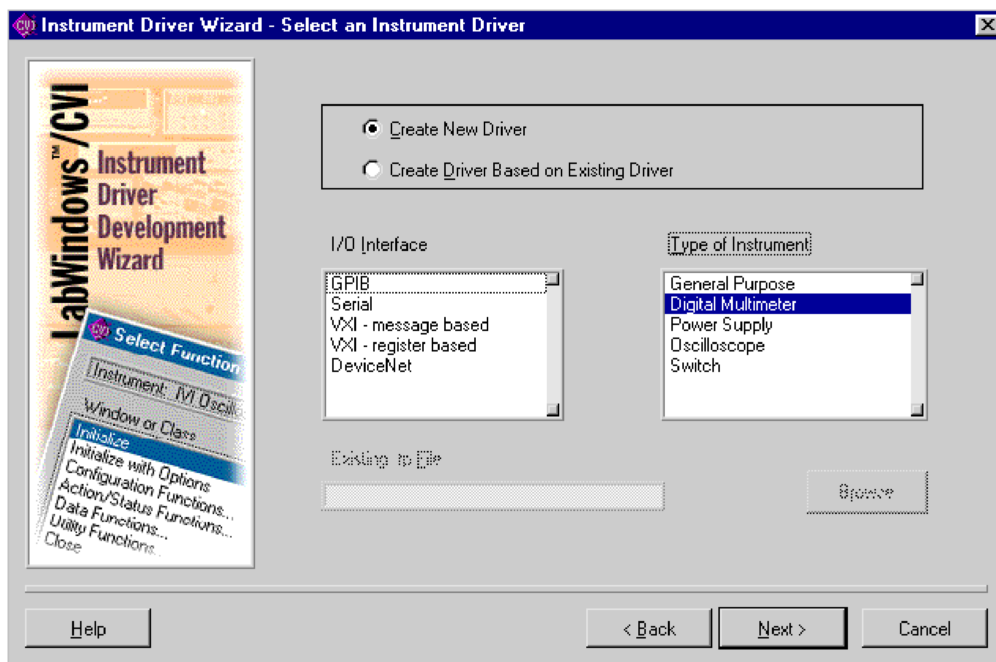


Figure 10-1. Select an Instrument Driver Panel

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

After you enter this information, the Select an Instrument Driver panel appears as shown in Figure 10-1. 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.

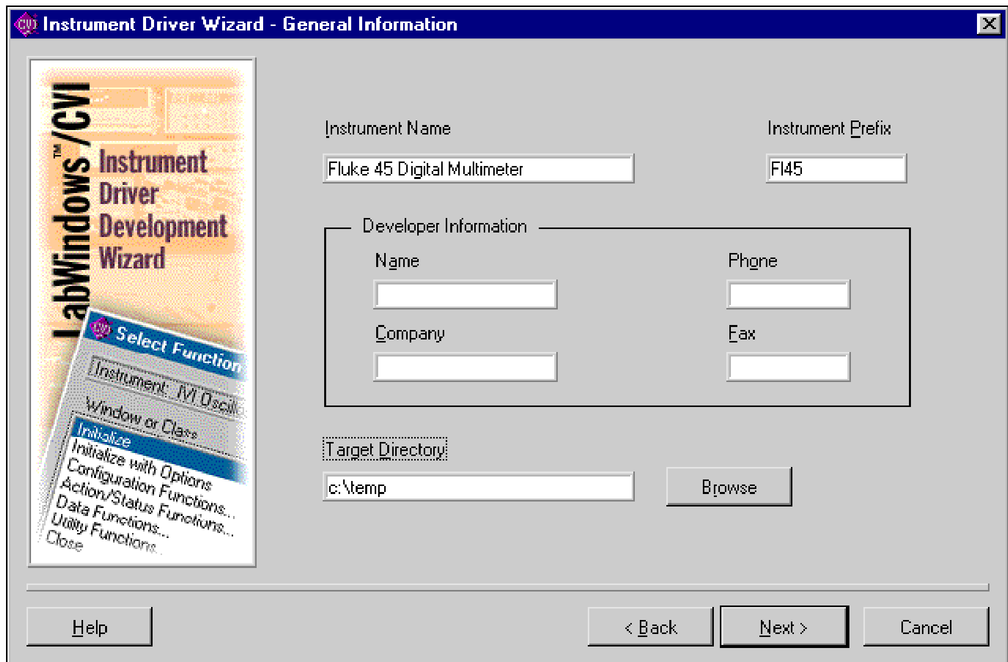


Figure 10-2. General Information Panel

Enter the general instrument driver information as follows.

- 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.

After you enter this information, the General Information panel appears as shown in Figure 10-2. Click on the **Next** button to continue.

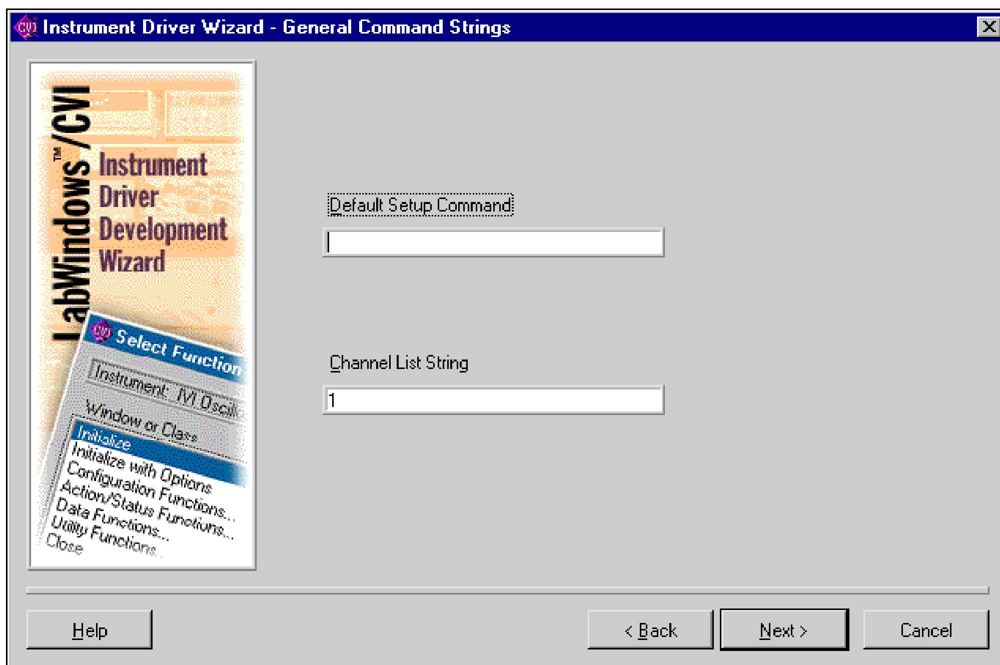


Figure 10-3. General Command Strings Panel

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.

After you enter this information, the General Command Strings panel appears as shown in Figure 10-3. Click on the **Next** button to continue.

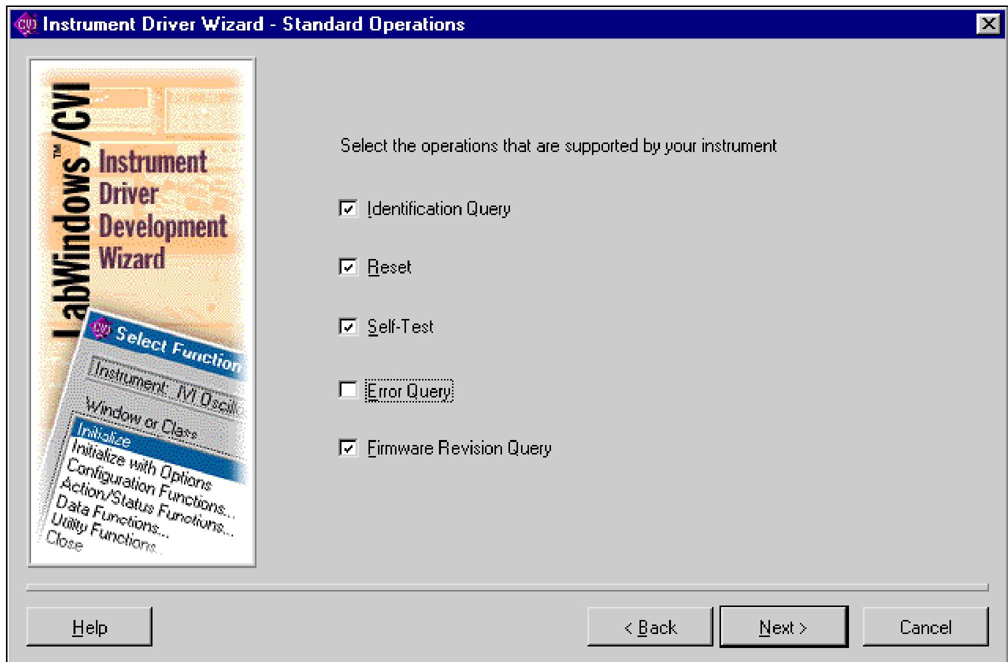


Figure 10-4. Standard Operations Panel

The Standard Operations panel, shown in Figure 10-4, allows you to select which standard operations your instrument supports. 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.

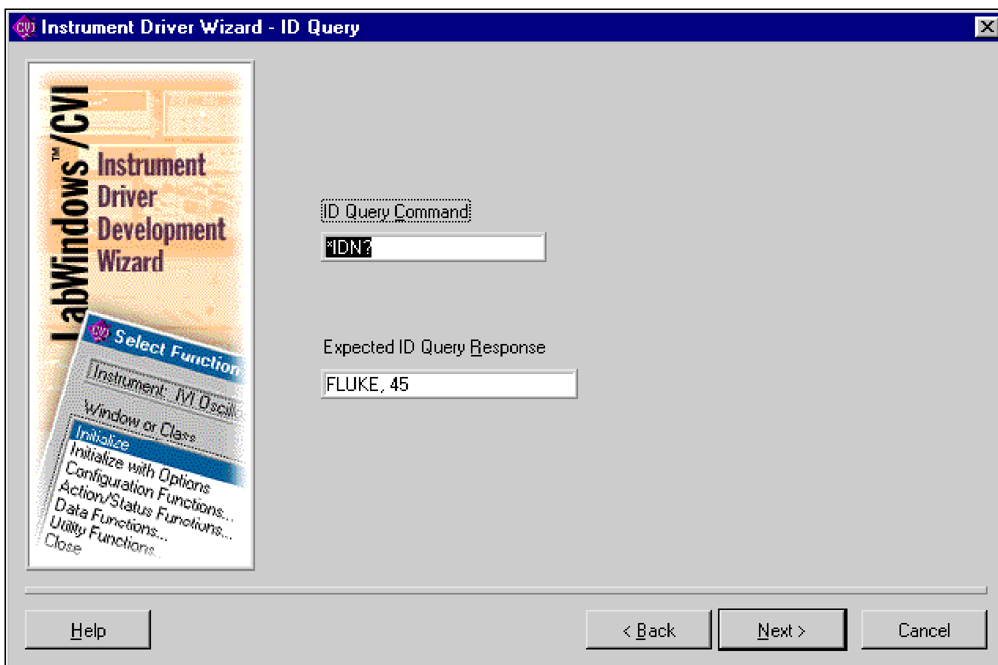


Figure 10-5. ID Query Panel

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.

After you enter this information, the ID Query panel appears as shown in Figure 10-5. Click on the **Next** button to continue.

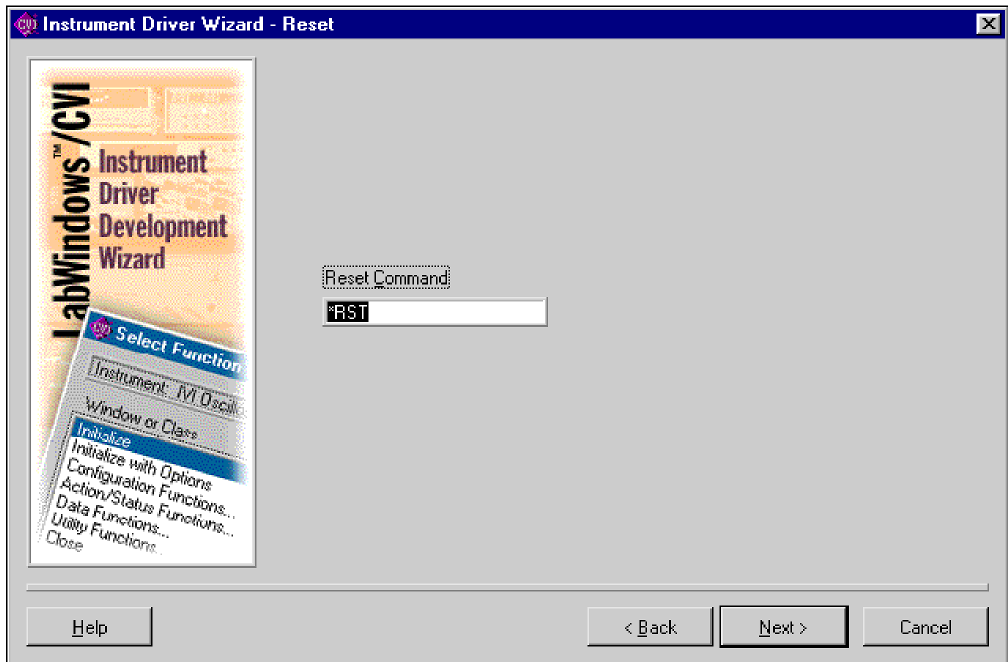


Figure 10-6. Reset Panel

The `*RST` command resets the Fluke 45. Enter `*RST` in the Reset Command control. This is the default.

After you enter this information, the Reset panel appears as shown in Figure 10-6. Click on the **Next** button to continue.

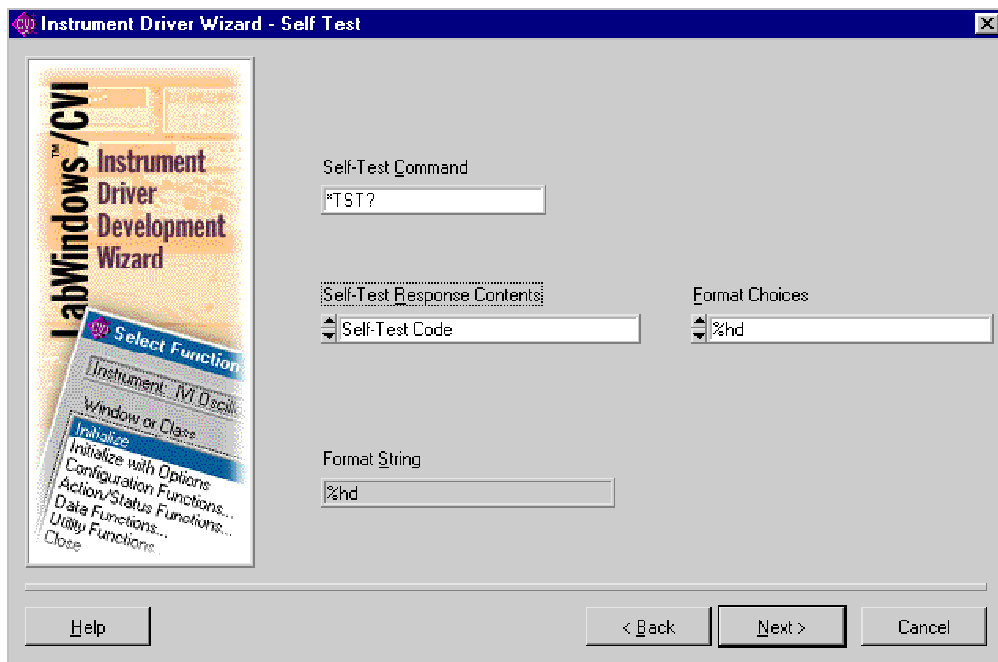


Figure 10-7. Self-Test Panel

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 list.
- Select `%hd` from the Format Choices list.

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.

After you enter this information, the Self Test panel appears as shown in Figure 10-7. Click on the **Next** button to continue.

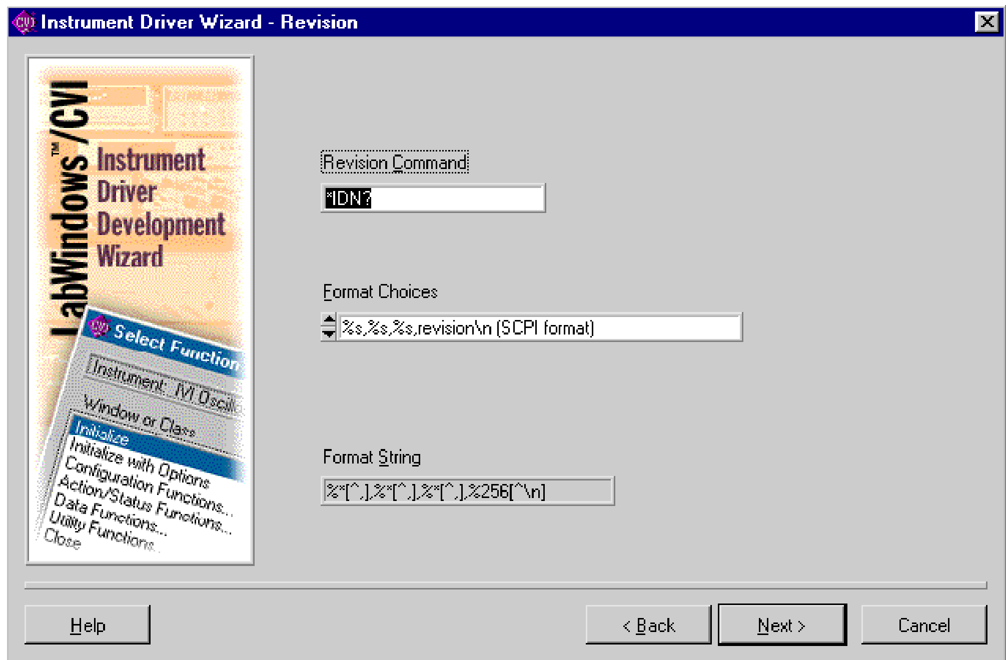


Figure 10-8. Revision Panel

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 list. 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.

After you enter this information, the Revision panel appears as shown in Figure 10-8. Click on the **Next** button to continue.

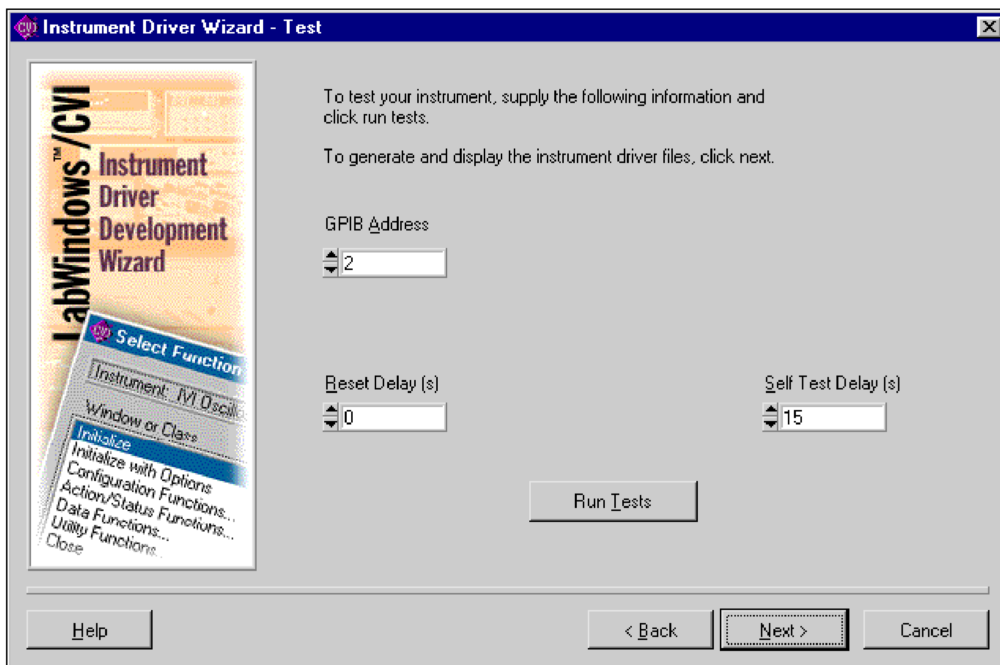


Figure 10-9. Test Panel

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.

After you enter this information, the Test panel appears as shown in Figure 10-9. Click on the **Run Tests** button.

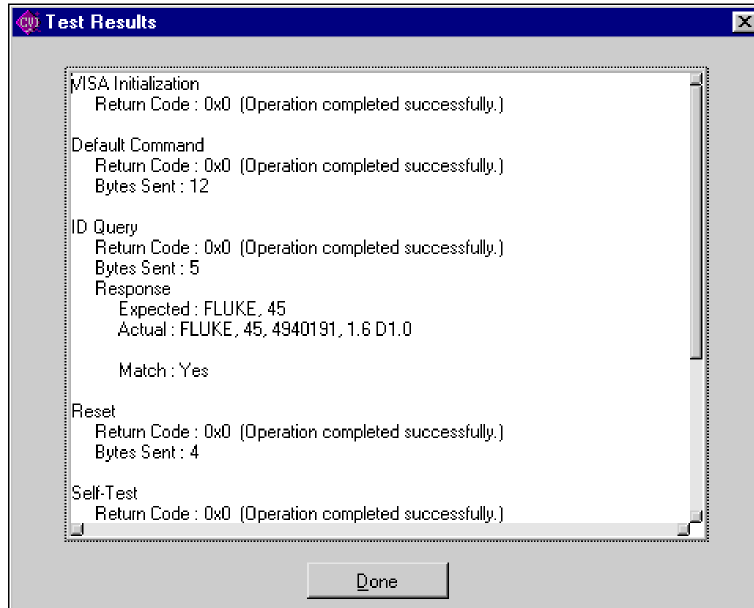


Figure 10-10. IVI Test Results Panel

Figure 10-10 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.

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 Chapter 9, *Required Instrument Driver Functions*, 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.

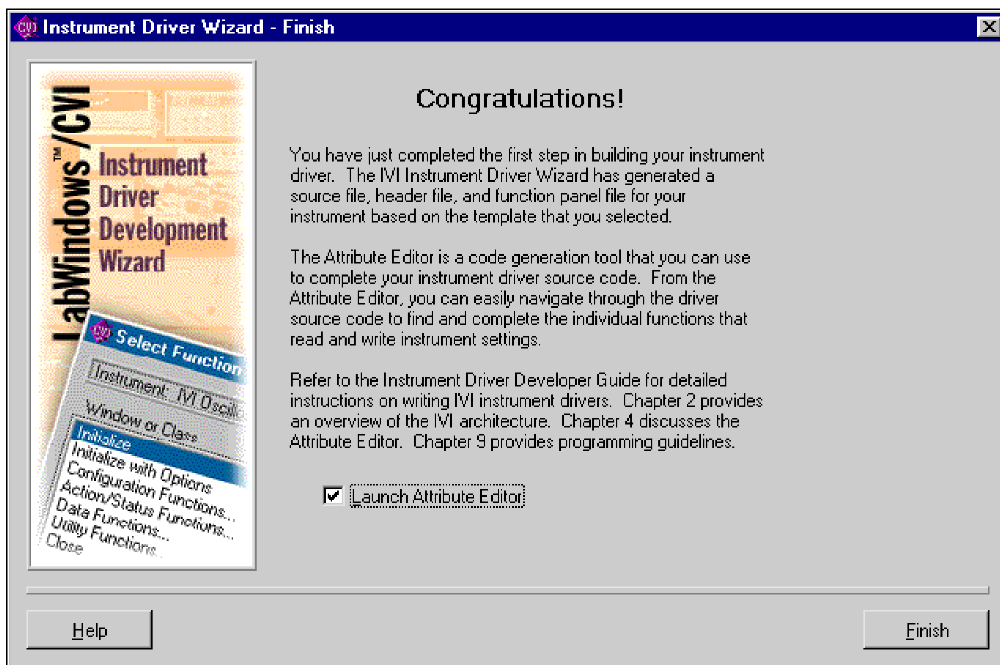


Figure 10-11. Finish Panel

Figure 10-11 shows the final panel of the wizard. Select the Launch Attribute Editor option and click on the **Finish** button to invoke the attribute editor.

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 the **Edit Instrument Attributes** command from the **Tools** menu. Figure 10-12 shows all the attributes that the Instrument Driver Development Wizard created for the Fluke 45 instrument driver.

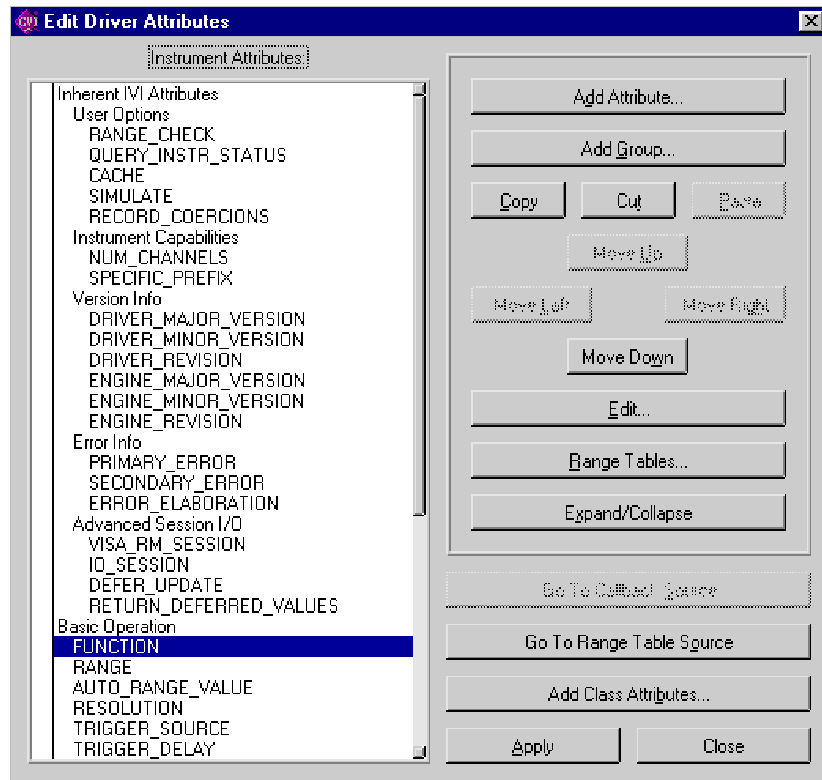


Figure 10-12. Edit Driver Attributes Dialog Box

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:

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 10-13.

Notice that the Instrument Driver Development Wizard has already filled in the attribute information.

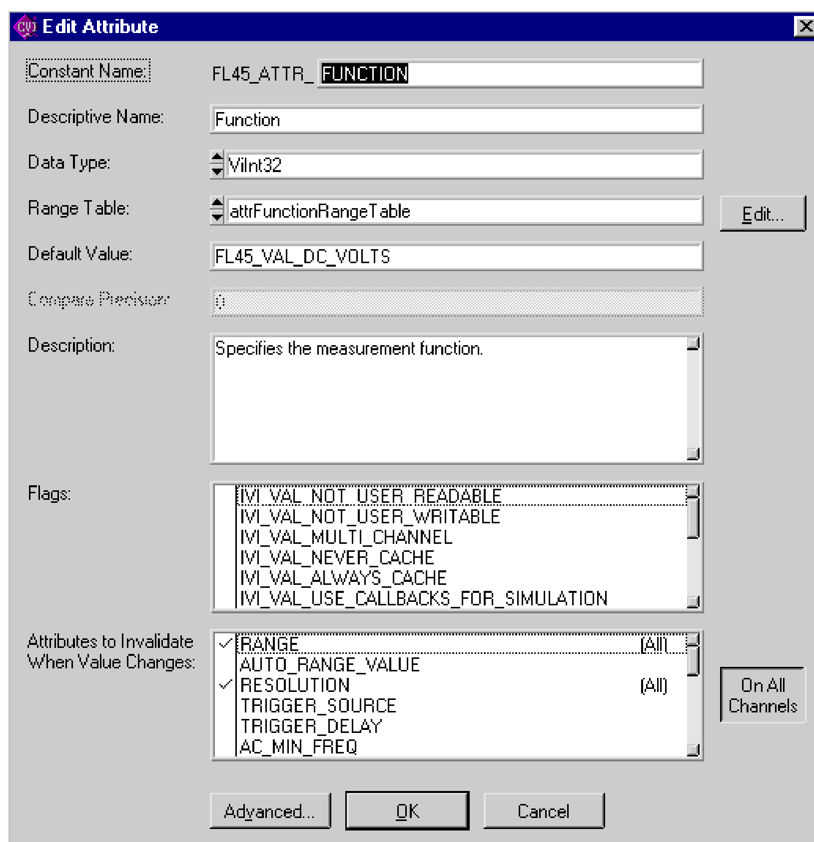


Figure 10-13. Edit Attribute Dialog Box

Click on the **Edit** button to edit the range table for the attribute. The Edit Range Table dialog box is shown in Figure 10-14.

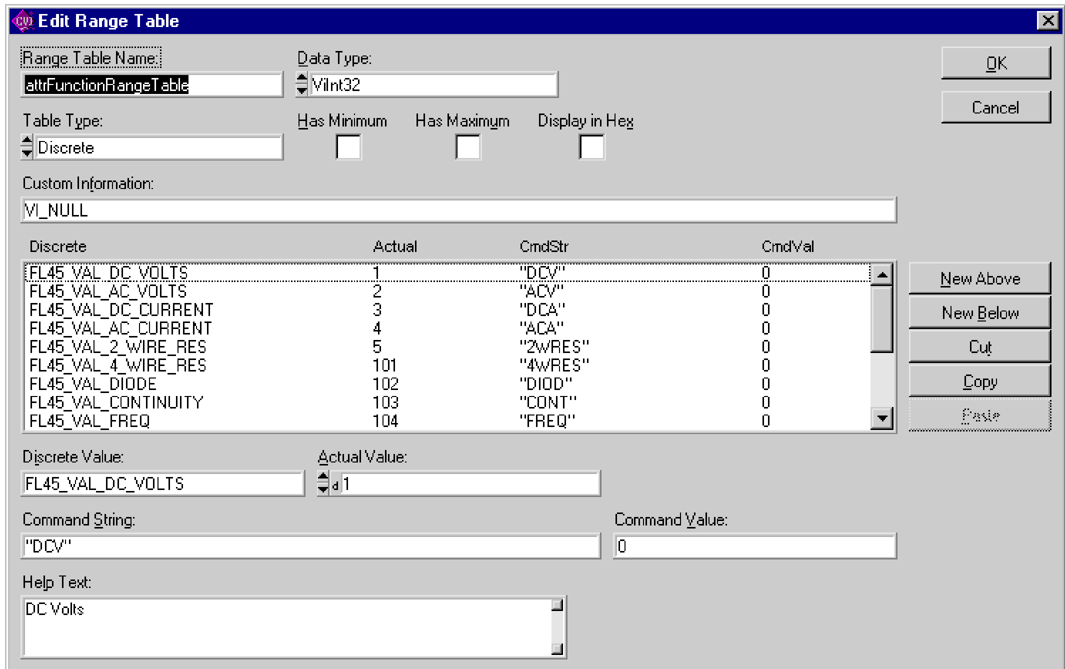


Figure 10-14. Edit Range Table Dialog Box

Modify the range table as follows:

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 10-15.

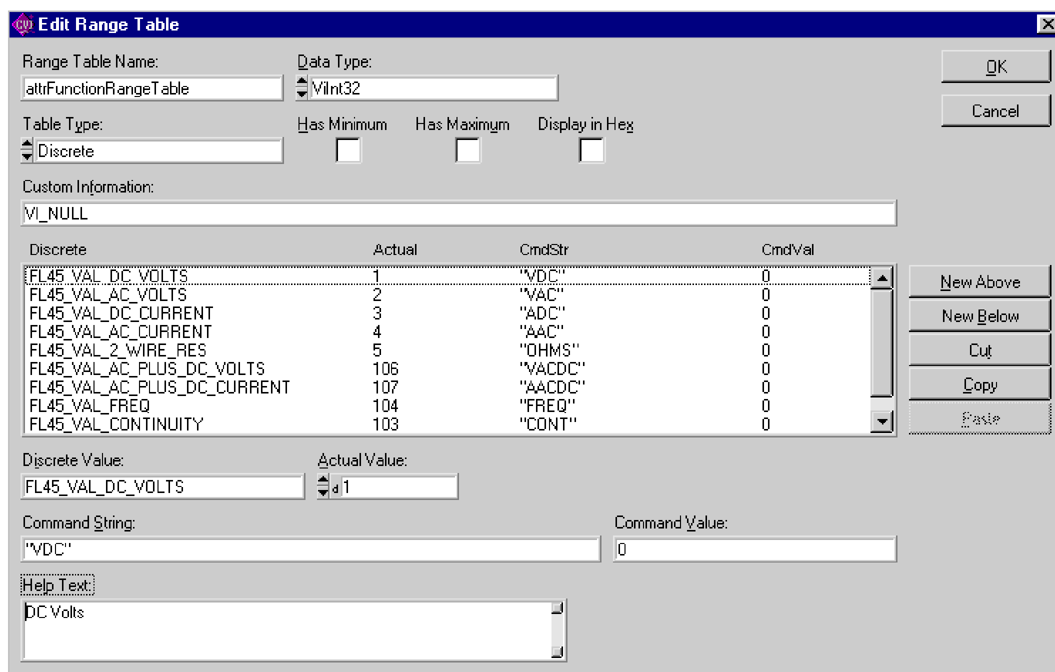


Figure 10-15. Edit Range Table Dialog Box With Modifications

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 must also delete the corresponding value that the `fl45.h` header file defines.

To delete the unused measurement function values that the `fl45.h` header file defines, perform the following steps:

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

Choose the **Edit Instrument Attributes** command from the **Tools** menu 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;
}
```

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 `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 the **Edit Instrument Attributes** command from the **Tools** menu to invoke the attribute editor.

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. Select the Miscellaneous group and click on the **Cut** button.
4. 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 10-16.

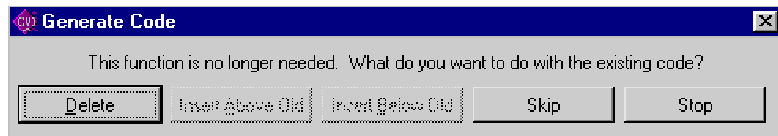


Figure 10-16. Generate Code Dialog Box

5. Click on the **Delete** button for each callback.
6. You must also 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 10-17.

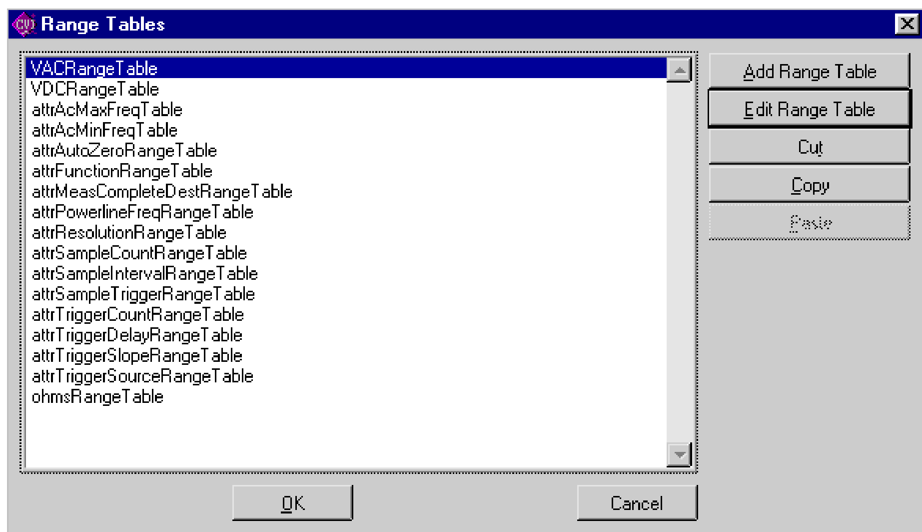


Figure 10-17. Range Tables Dialog Box

7. 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 10-18.

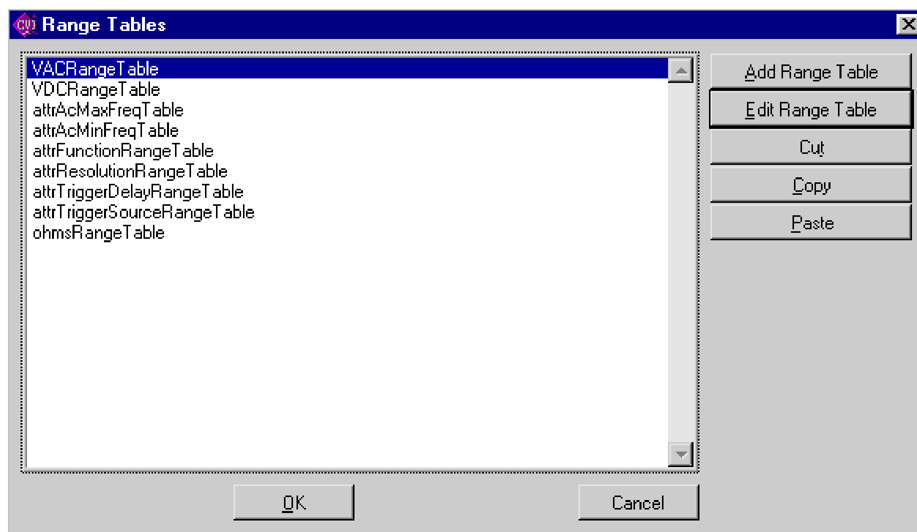


Figure 10-18. Range Tables Dialog Box with Modifications

8. Click on the **OK** button to return to the Edit Driver Attributes dialog box.
9. Click on the **Apply** button to commit the range tables changes in the instrument driver files.
10. Click on the **Close** button on the Edit Driver Attributes dialog box to close the attribute editor.

You must also delete the values that the `fl45.h` header file defines for the range tables.

To delete the defined constants for values that the Fluke 45 does not use, perform the following:

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_ZERO -*/

#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

Use the **Open** command in the **File** menu to open the `fl45.fp` function panel file. The function tree appears as shown in Figure 10-19.

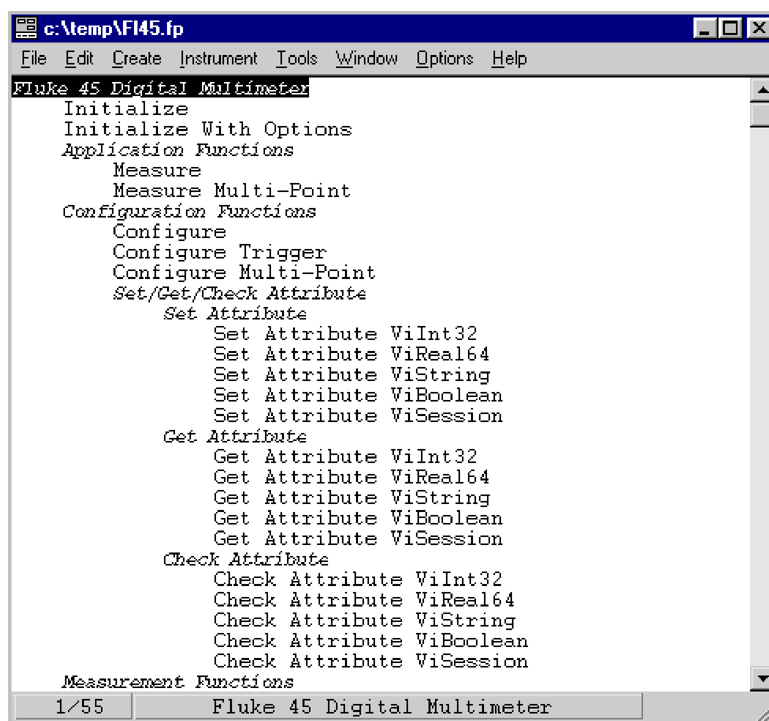


Figure 10-19. Fluke 45 Function Tree

Scroll down to the Fetch function. Right-click on the Fetch function to display the context menu. The context menu is shown in Figure 10-20.

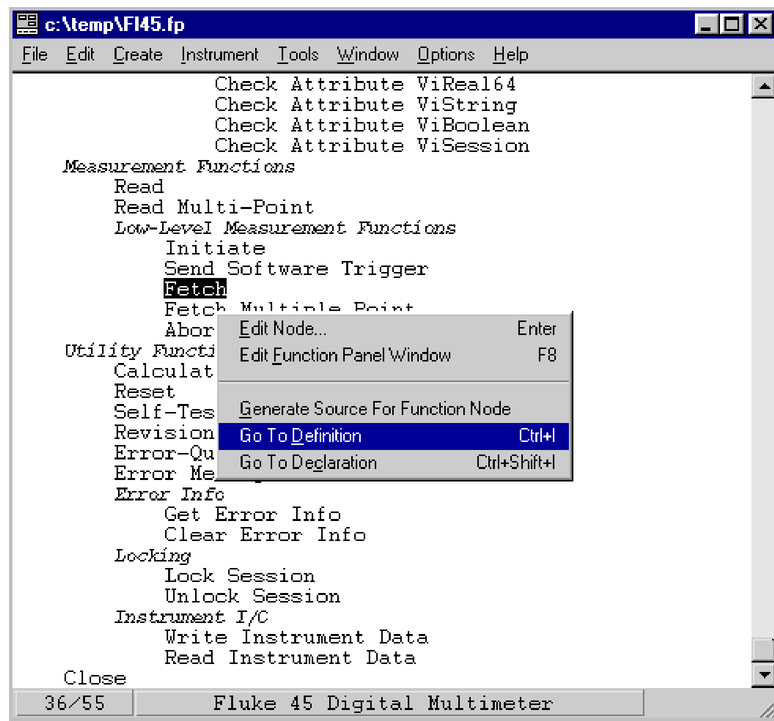


Figure 10-20. Function Tree Editor Context Menu

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.

Modify the code for the `Fl45_Fetch` function as follows:

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 if (Ivi_UseSpecificSimulation (vi))
        {
        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 operations. These operations are shown in the following table:

Table 10-1. 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.

To delete the functions, perform the following steps:

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. Choose the **Cut** command from the **Edit** menu to delete the function panel.
9. For each of the remaining functions, you can either repeat the steps above, or make all the edits in each file at once.

Example 4—Adding New Attributes and Functions

This example shows how to add new attributes and functions to your instrument driver. 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. The 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 the **Edit Instrument Attributes** command from the **Tools** menu to launch the attribute editor. Select the Advanced Triggering group. Select 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 10-21.

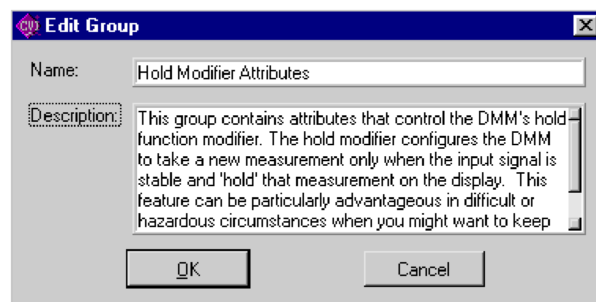


Figure 10-21. Edit Group Dialog Box

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 10-22.

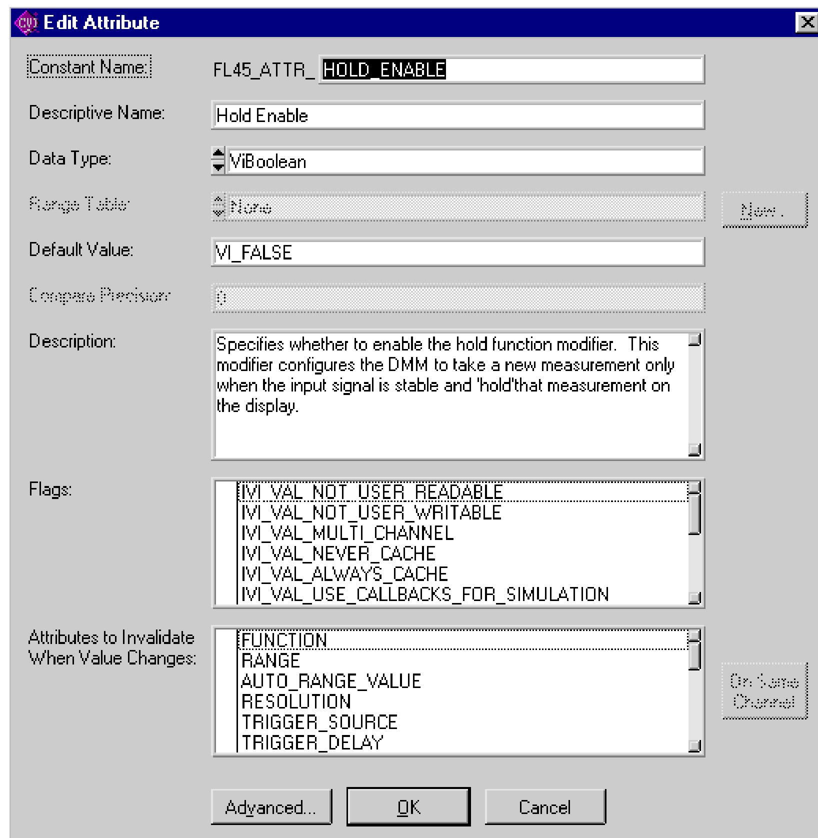


Figure 10-22. Hold Enable Edit Attribute Dialog Box

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 make 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;
}
```

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_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

Choose the **Edit Instrument Attributes** command from the **Tools** menu 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.

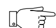
- Select `ViInt32` from the Data Type ring control.
- Click on the **New** button to create a range table for the attribute.

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 following information for the range table entries.

Table 10-2. 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 10-23.

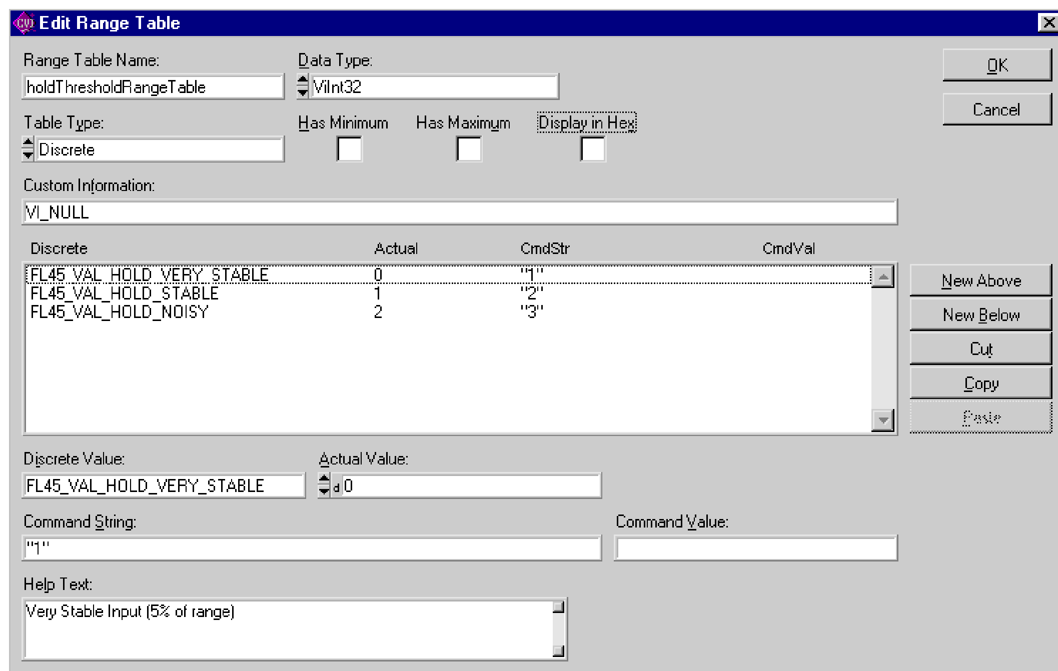


Figure 10-23. Hold Threshold Edit Range Table Dialog Box

Click on the **OK** button to return 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 10-24.

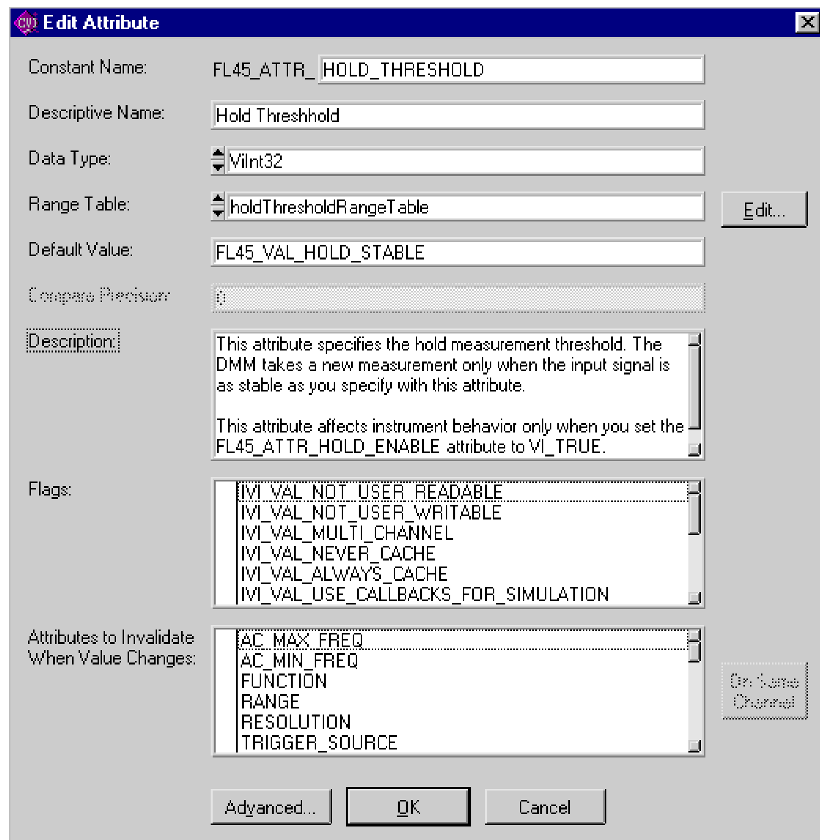


Figure 10-24. Hold Threshold Edit Attribute Dialog Box

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

Use the **Open** command in the **File** menu to open the `fl45.fp` function panel file. Select the Configure Trigger function. Choose the **Function Panel Window** from the **Create** menu. 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 10-25.

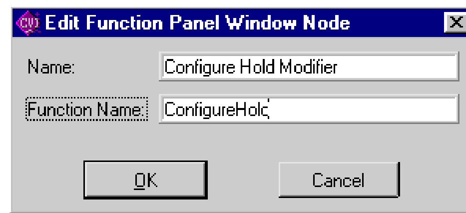


Figure 10-25. Edit Function Panel Window Node Dialog Box

Click on the **OK** button to return to the function tree editor.

To edit the function panel, perform the following steps:

1. Select the Configure Hold Modifier function and choose the **Edit Function Panel Window** command from the **Edit** menu.
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 **Close** from the **File** menu of the Help Editor.

To add an Instrument Handle control to the function panel, perform the following steps:

1. Press <Ctrl-Page Up> to display the Configure Trigger function panel.
2. Select the Instrument Handle control.
3. Choose the **Copy Controls** command from the **Edit** menu.
4. Press <Ctrl-Page Down> to display the Configure Hold Modifier function panel.
5. Choose the **Paste** command from the **Edit** menu 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.

Add a control to specify whether to enable the Hold Modifier as follows.

1. Choose the **Binary** command from the **Create** menu.
2. Complete the Edit Binary Control and Edit On/Off Settings dialog boxes as shown in Figures 10-26 and 10-27.

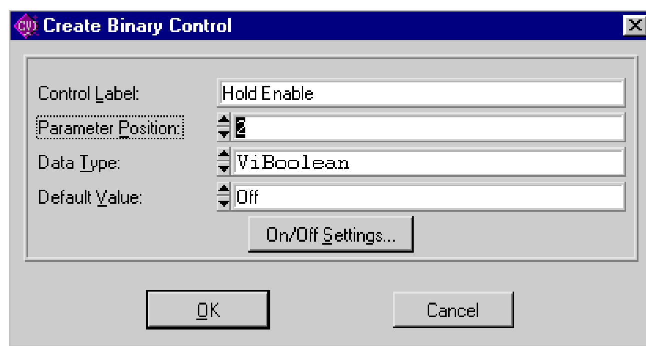


Figure 10-26. Create Binary Control Dialog Box

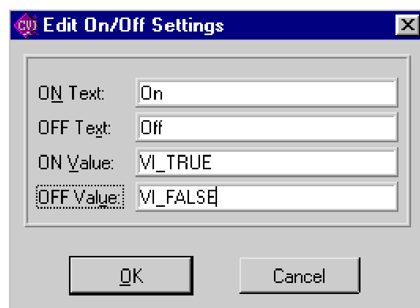


Figure 10-27. Edit On/Off Setting Dialog Box

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.

Add help to the Hold Enable control as follows:

1. Select the Hold Enable control and choose the **Control Help** command from the **Edit** menu.
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 the **Close** command in the **File** menu to return to the function panel window.

Add a control to specify the hold threshold as follows.

1. Choose the **Slide** command from the **Create** menu.
2. Complete the Edit Slide Control dialog box as shown in Figure 10-28.

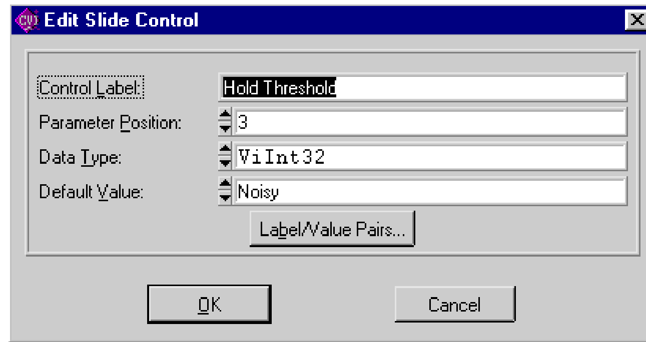


Figure 10-28. Edit Slide Control Dialog Box

3. Click on the **Label/Value Pairs** button and complete the Edit Label/Value Pairs dialog box as shown in Figure 10-29.

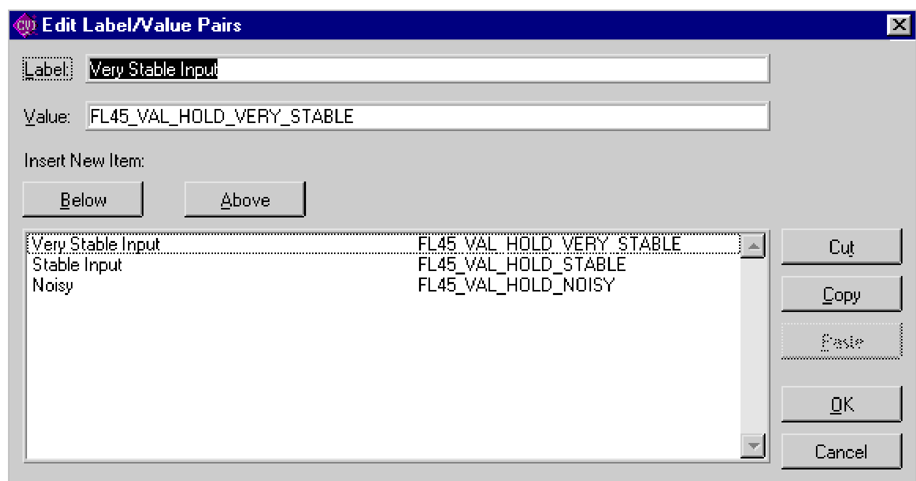


Figure 10-29. Edit Label/Value Pairs Dialog Box

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.

Add help to the Hold Threshold control as follows:

1. Select the Hold Threshold control and choose the **Control Help** command from the **Edit** menu.
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. Choose the **Close** command from the **File** menu of the help editor.

Add a return control to indicate the status of the function as follows.

1. Press <Ctrl-Page Up> to display the Configure Trigger function panel.
2. Select the Status control.
3. Choose the **Copy Controls** command from the **Edit** menu.
4. Press <Ctrl-Page Down> to display the Configure Hold Modifier function panel.
5. Choose the **Paste** command from the **Edit** menu 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.

Add help to the Status control as follows.

1. Select the Status return value and select the **Control Help** command from the **Edit** menu.
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 the **Close** command from the **File** menu of the help editor.

The Configure Hold Modifier function panel windows now appear as shown in Figure 10-30.

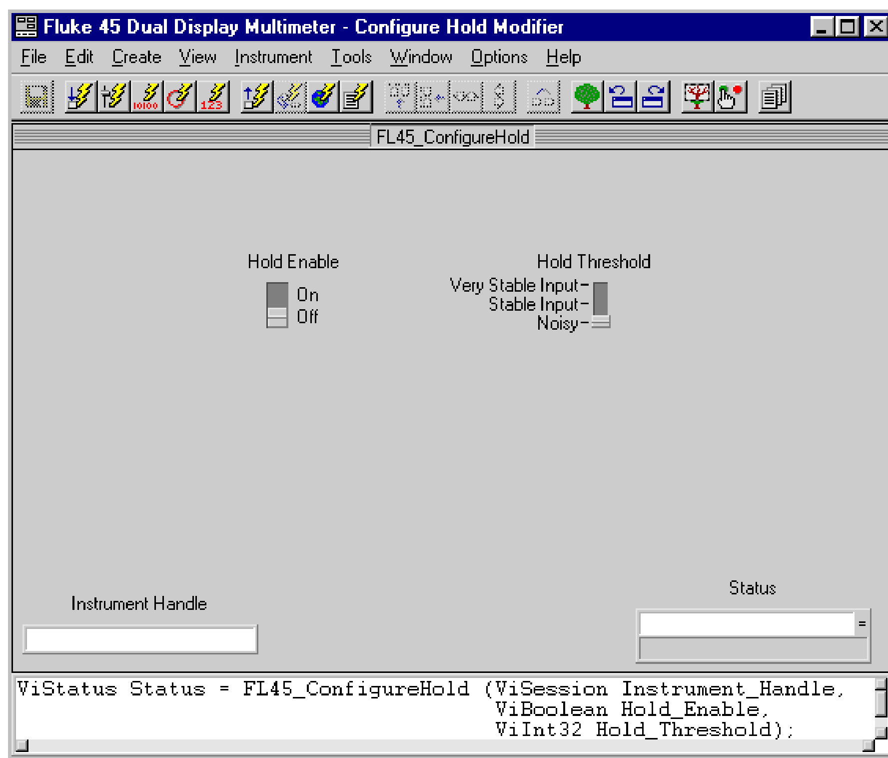


Figure 10-30. Configure Hold Function Panel Window

Creating the Configure Hold Function Body

Perform the following steps to create the Configure Hold Function Body.

1. Use the **Open** command in the **File** menu 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.

```
/*****************************************************************
 * 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 **Generate Documentation** command from the **Options** menu. The dialog box shown in Figure 10-31 lets you choose the programming language for which you wish to generate documentation.

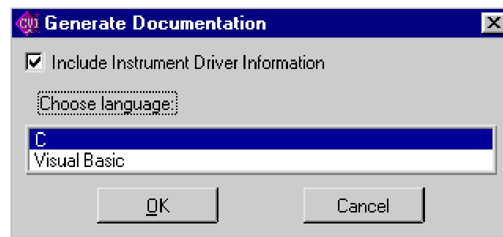


Figure 10-31. Generate Documentation Dialog

Select the language you want and click on the **OK** button. The instrument driver documentation appears as shown in Figure 10-32.

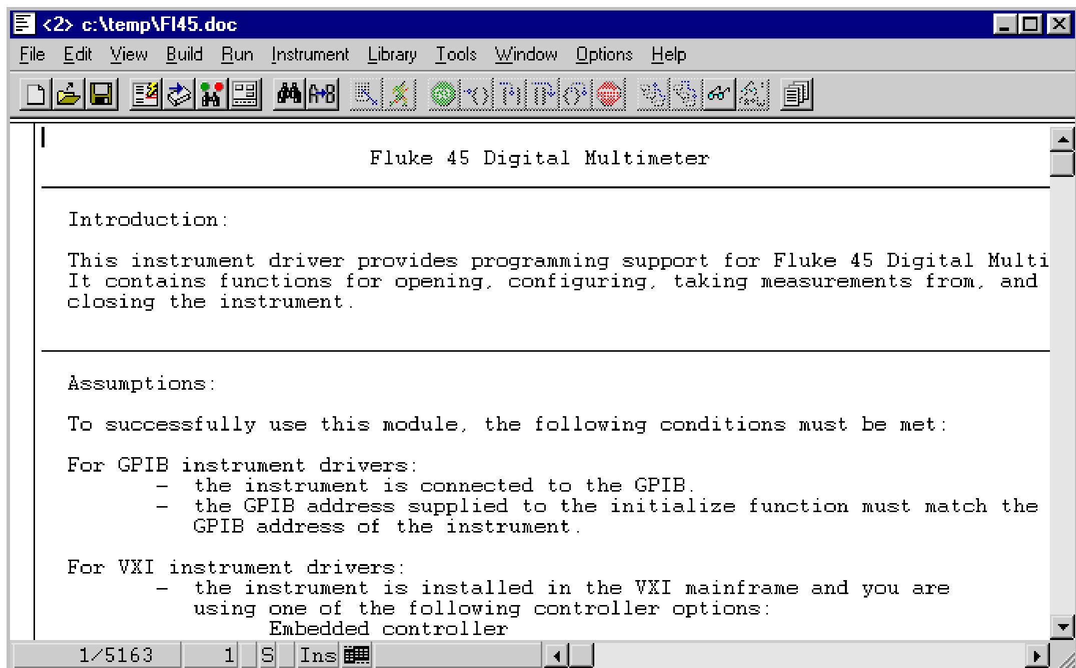


Figure 10-32. Fluke 45 Instrument Driver Documentation Window

Choose the **Save** command from the **File** menu and save the file as `fl45.doc`.

Creating Windows Help for the Driver

Open the file `fl45.fp` and edit the function tree. Choose the **Generate Windows Help** command from the **Options** menu. 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 10-33.

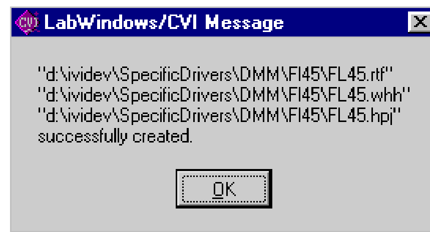


Figure 10-33. LabWindows/CVI Message Dialog Box

Under Windows 95/NT, run `cvi/sdk/bin/hcrtf -x fl45.hpj` to generate the help file. The `hcrtf.exe` program installs with LabWindows/CVI for Windows 95/NT.

Under Windows 3.1, run `hc fl45.hpj`. LabWindows/CVI does not install the `hc.exe` program for Windows 3.1. You must have a licensed copy of the help compiler.

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 the **Create IVI Instrument Driver** command from the **Tools** menu. Click on the **Next** button on the welcome panel to begin.

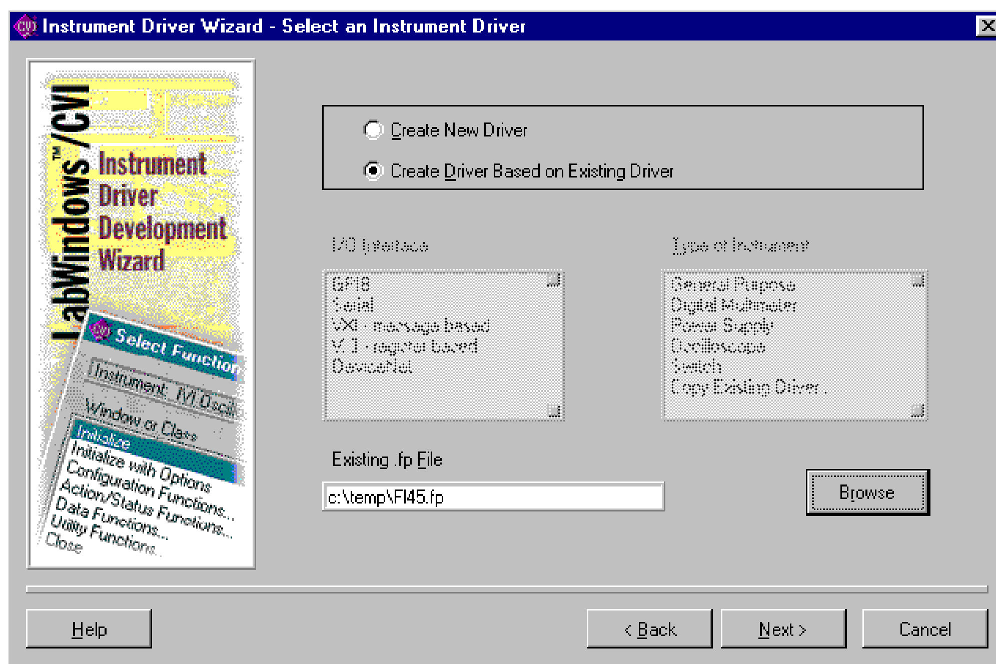


Figure 10-34. Select an Instrument Driver Panel

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 10-34. Click on the **Next** button to continue.

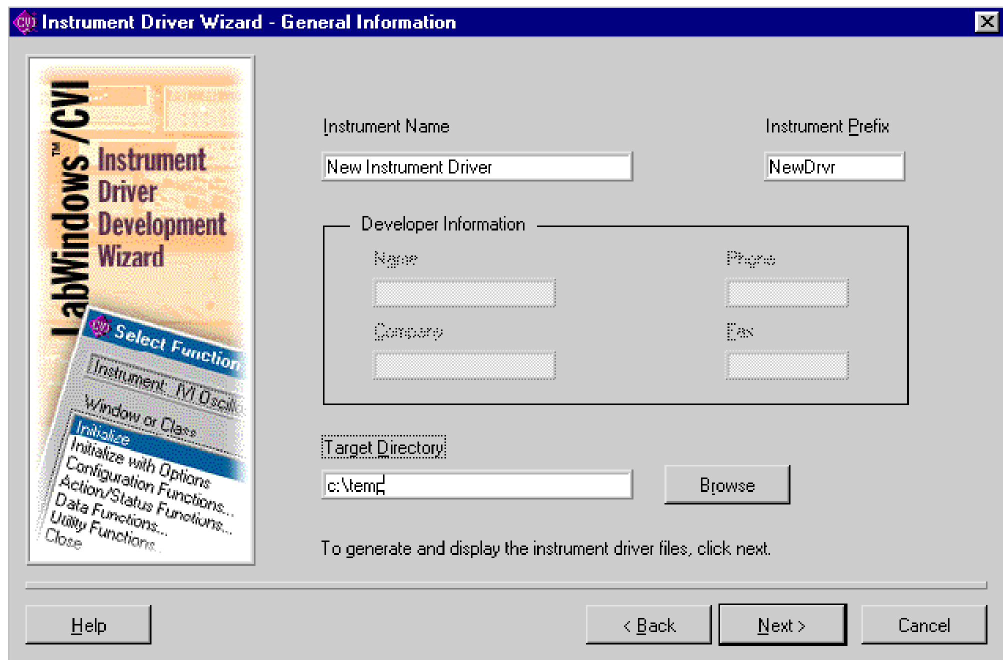


Figure 10-35. General Information Panel

Enter the name, prefix, and target directory for the new driver. After you enter this information, the panel appears as shown in Figure 10-35. 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 file name 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 be 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 are as follows:

- 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.

11

IVI Library

This chapter describes the functions in the LabWindows/CVI IVI (Intelligent Virtual Instruments) Library. The *IVI Library Function Overview* section contains general information about the IVI Library functions and panels. The *IVI Library Function Reference* section contains an alphabetical list of function descriptions.

IVI Library Function Overview

The IVI Library includes functions you can use to develop instrument drivers. It also includes functions you can use to develop application-level utilities for analyzing and displaying information from instrument drivers.

IVI Library Function Panels

The IVI Library function panels are grouped in the tree structure in Table 11-1 according to the types of operations they perform. The headings in the tree are the names of function classes and subclasses. Function classes and subclasses are groups of related function panels. Within the headings are the names of individual function panels. Each panel generates one IVI function call.

The IVI Library Function Tree

The following list shows the structure of the IVI Library function tree.

- Instrument Driver Session
  - Locking
- Channels
- Attribute Creation
  - Add Attribute
- Invalidation Lists
- Callbacks
  - Set Read Callback
  - Set Write Callback
  - Set Check Callback
  - Set Coerce Callback
  - Set Compare Callback
  - Comparison Precision
- Set/Get/Check Attribute
  - Set Attribute
  - Get Attribute
  - Check Attribute
  - Caching/Status-Checking Control
  - Range Tables
    - Range Table Entries
      - Get ViInt32 Entry
      - Get ViReal64 Entry
  - Range Table Ptr
  - Dynamic Range Tables
  - Error Information
    - Instrument Specific Error Queue
  - Memory Allocation
  - Helper Functions
    - Inherent Attribute Accessors
    - String Callbacks
    - Direct Instrument I/O
    - String/Value Tables
    - Value Manipulation
    - Default Callbacks
    - Attribute Information
  - Configuration
    - Configuration File
    - Logical Names
    - Run-Time Configuration

Table 11-1. IVI Library Function Tree

Class or Panel	Function
Instrument Driver Session	
Create New Session	`Ivi_SpecificDriverNew`
Validate Session	`Ivi_ValidateSession`
Locking	
Lock Session	`Ivi_LockSession`
Unlock Session	`Ivi_UnlockSession`
Dispose Session	`Ivi_Dispose`
Channels	
Build Channel Table	`Ivi_BuildChannelTable`
Add To Channel Table	`Ivi_AddToChannelTable`
Restrict Attr To Channels	`Ivi_RestrictAttrToChannels`
Validate Attr For Channel	`Ivi_ValidateAttrForChannel`
Coerce Channel Name	`Ivi_CoerceChannelName`
Get Nth Channel String	`Ivi_GetNthChannelString`
Get User Channel Name	`Ivi_GetUserChannelName`
Attribute Creation	
Add Attribute	
Add Attribute ViInt32	`Ivi_AddAttributeViInt32`
Add Attribute ViReal64	`Ivi_AddAttributeViReal64`
Add Attribute ViString	`Ivi_AddAttributeViString`
Add Attribute ViBoolean	`Ivi_AddAttributeViBoolean`
Add Attribute ViSession	`Ivi_AddAttributeViSession`
Add Attribute ViAddr	`Ivi_AddAttributeViAddr`
Invalidation Lists	
Add Attribute Invalidation	`Ivi_AddAttributeInvalidation`
Delete Attribute Invalidation	`Ivi_DeleteAttributeInvalidation`
Callbacks	
Set Read Callback	
Set Read Callback ViInt32	`Ivi_SetAttrReadCallbackViInt32`
Set Read Callback ViReal64	`Ivi_SetAttrReadCallbackViReal64`
Set read Callback ViString	`Ivi_SetAttrReadCallbackViString`
Set Read Callback ViBoolean	`Ivi_SetAttrReadCallbackViBoolean`
Set Read Callback ViSession	`Ivi_SetAttrReadCallbackViSession`
Set Read Callback ViAddr	`Ivi_SetAttrReadCallbackViAddr`
Set Write Callback	
Set Write Callback ViInt32	`Ivi_SetAttrWriteCallbackViInt32`
Set Write Callback ViReal64	`Ivi_SetAttrWriteCallbackViReal64`
Set Write Callback ViString	`Ivi_SetAttrWriteCallbackViString`
Set Write Callback ViBoolean	`Ivi_SetAttrWriteCallbackViBoolean`
Set Write Callback ViSession	`Ivi_SetAttrWriteCallbackViSession`
Set Write Callback ViAddr	`Ivi_SetAttrWriteCallbackViAddr`
Set Check Callback	
Set Check Callback ViInt32	`Ivi_SetAttrCheckCallbackViInt32`
Set Check Callback ViReal64	`Ivi_SetAttrCheckCallbackViReal64`

Table 11-1. IVI Library Function Tree (Continued)

Function	Name
Callbacks (continued)	
Set Check Callback (continued)	
Set Check Callback ViString	Ivi_SetAttrCheckCallbackViString
Set Check Callback ViBoolean	Ivi_SetAttrCheckCallbackViBoolean
Set Check Callback ViSession	Ivi_SetAttrCheckCallbackViSession
Set Check Callback ViAddr	Ivi_SetAttrCheckCallbackViAddr
Set Coerce Callback	
Set Coerce Callback ViInt32	Ivi_SetAttrCoerceCallbackViInt32
Set Coerce Callback ViReal64	Ivi_SetAttrCoerceCallbackViReal64
Set Coerce Callback ViString	Ivi_SetAttrCoerceCallbackViString
Set Coerce Callback ViBoolean	Ivi_SetAttrCoerceCallbackViBoolean
Set Coerce Callback ViSession	Ivi_SetAttrCoerceCallbackViSession
Set Coerce Callback ViAddr	Ivi_SetAttrCoerceCallbackViAddr
Set Compare Callback	
Set Compare Callback ViInt32	Ivi_SetAttrCompareCallbackViInt32
Set Compare Callback ViReal64	Ivi_SetAttrCompareCallbackViReal64
Set Compare Callback ViString	Ivi_SetAttrCompareCallbackViString
Set Compare Callback ViBoolean	Ivi_SetAttrCompareCallbackViBoolean
Set Compare Callback ViSession	Ivi_SetAttrCompareCallbackViSession
Set Compare Callback ViAddr	Ivi_SetAttrCompareCallbackViAddr
Set Range Table Callback	Ivi_SetAttrRangeTableCallback
Comparison Precision	
Set Comparison Precision	Ivi_SetAttrComparePrecision
Get Comparison Precision	Ivi_GetAttrComparePrecision
Delete Attribute	Ivi_DeleteAttribute
Set/Get/Check Attribute	
Set Attribute	
Set Attribute ViInt32	Ivi_SetAttributeViInt32
Set Attribute ViReal64	Ivi_SetAttributeViReal64
Set Attribute ViString	Ivi_SetAttributeViString
Set Attribute ViBoolean	Ivi_SetAttributeViBoolean
Set Attribute ViSession	Ivi_SetAttributeViSession
Set Attribute ViAddr	Ivi_SetAttributeViAddr
Update Deferred Settings	Ivi_UpdateGetAttribute
Get Attribute	
Get Attribute ViInt32	Ivi_GetAttributeViInt32
Get Attribute ViReal64	Ivi_GetAttributeViReal64
Get Attribute ViString	Ivi_GetAttributeViString
Get Attribute ViBoolean	Ivi_GetAttributeViBoolean
Get Attribute ViSession	Ivi_GetAttributeViSession
Get Attribute ViAddr	Ivi_GetAttributeViAddr
Check Attribute	
Check Attribute ViInt32	Ivi_CheckAttributeViInt32
Check Attribute ViReal64	Ivi_CheckAttributeViReal64
Check Attribute ViString	Ivi_CheckAttributeViString
Check Attribute ViBoolean	Ivi_CheckAttributeViBoolean

Table 11-1. IVI Library Function Tree (Continued)

Function Tree	Function Name
Set/Get/Check Attribute (continued)	
Check Attribute (continued)	
Check Attribute ViSession	`Ivi_CheckAttributeViSession`
Check Attribute ViAddr	`Ivi_CheckAttributeViAddr`
Caching/Status-Checking Control	
Invalidate Attribute	`Ivi_InvalidateAttribute`
Invalidate All Attributes	`Ivi_InvalidateAllAttributes`
Need To Check Status	`Ivi_NeedToCheckStatus`
Set Need To Check Status	`Ivi_SetNeedToCheckStatus`
Range Tables	
Get Attribute Range Table	`Ivi_GetAttrRangeTable`
Validate Range Table	`Ivi_ValidateRangeTable`
Range Table Entries	
Get Range Table Num Entries	`Ivi_GetRangeTableNumEntries`
Get ViInt32 Entry	
ViInt32 Entry From Value	`Ivi_GetViInt32EntryFromValue`
ViInt32 Entry From String	`Ivi_GetViInt32EntryFromString`
ViInt32 Entry From Index	`Ivi_GetViInt32EntryFromIndex`
ViInt32 Entry FromCmdValue	`Ivi_GetViInt32EntryFromCmdValue`
ViInt32 Entry From CoercedVal	`Ivi_GetViInt32EntryFromCoercedVal`
Get ViReal64 Entry	
ViReal64 Entry From Value	`Ivi_GetViReal64EntryFromValue`
ViReal64 Entry From String	`Ivi_GetViReal64EntryFromString`
ViReal64 Entry From Index	`Ivi_GetViReal64EntryFromIndex`
ViReal64 Entry From CmdValue	`Ivi_GetViReal64EntryFromCmdValue`
ViReal64 Entry From CoercedVal	`Ivi_GetViReal64EntryFromCoercedVal`
Range Table Ptr	
Get Stored Range Table Ptr	`Ivi_GetStoredRangeTablePtr`
Set Stored Range Table Ptr	`Ivi_SetStoredRangeTablePtr`
Dynamic Range Tables	
Range Table New	`Ivi_RangeTableNew`
Set Range Table Entry	`Ivi_SetRangeTableEntry`
Set Range Table End	`Ivi_SetRangeTableEnd`
Range Table Free	`Ivi_RangeTableFree`
Error Information	
Set Error Info	`Ivi_SetErrorInfo`
Clear Error Info	`Ivi_ClearErrorInfo`
Get Error Info	`Ivi_GetErrorInfo`
Get Error Message	`Ivi_GetErrorMessage`
Get Specific Driver Status Desc	`Ivi_GetSpecificDriverStatusDesc`
Instrument Specific Error Queue	
Queue Instr Specific Error	`Ivi_QueueInstrSpecificError`
Dequeue Instr Specific Error	`Ivi_DequeueInstrSpecificError`
Clear Instr Specific Err Queue	`Ivi_ClearInstrSpecificErrorQueue`
Instr Specific Error Queue Size	`Ivi_InstrSpecificErrorQueueSize`

Table 11-1. IVI Library Function Tree (Continued)

Function	Name
Set/Get/Check Attribute (continued)	
Memory Allocation	
Allocate Memory	`Ivi_Alloc`
Free Allocated Memory	`Ivi_Free`
Free All Allocated Memory	`Ivi_FreeAll`
Helper Functions	
Inherent Attribute Accessors	
I/O Session	`Ivi_IOSession`
Range Checking	`Ivi_RangeChecking`
Query Instr Status	`Ivi_QueryInstrStatus`
Simulating	`Ivi_Simulating`
Use Specific Simulation	`Ivi_UseSpecificSimulation`
Spying	`Ivi_Spying`
Interchange Checking	`Ivi_InterchangeCheck`
String Callbacks	
Set Value in String Callback	`Ivi_SetValInStringCallback`
Direct Instrument I/O	
Write Instr Data	`Ivi_WriteInstrData`
Read Instr Data	`Ivi_ReadInstrData`
Read To File	`Ivi_ReadToFile`
Write From File	`Ivi_WriteFromFile`
String/Value Tables	
Get Value From Table	`Ivi_GetStringFromTable`
Get String From Table	`Ivi_GetValueFromTable`
Value Manipulation	
Check Numeric Range	`Ivi_CheckNumericRange`
Check Boolean Range	`Ivi_CheckBooleanRange`
Coerce Boolean	`Ivi_CoerceBoolean`
Compare With Precision	`Ivi_CompareWithPrecision`
Default Callbacks	
Dflt Check Callback ViInt32	`Ivi_DefaultCheckCallbackViInt32`
Dflt Coerce Callback ViInt32	`Ivi_DefaultCoerceCallbackViInt32`
Dflt Check Callback ViReal64	`Ivi_DefaultCheckCallbackViReal64`
Dflt Coerce Callback ViReal64	`Ivi_DefaultCoerceCallbackViReal64`
Dflt Compare Callback ViReal64	`Ivi_DefaultCompareCallbackViReal64`
Dflt Coerce Callback ViBoolean	`Ivi_DefaultCoerceCallbackViBoolean`
Dflt Buffered I/O Callback	`Ivi_DefaultBufferedIOCallback`
Attribute Information	
Get Num Attributes	`Ivi_GetNumAttributes`
Get Nth Attribute	`Ivi_GetNthAttribute`
Get Attribute Flags	`Ivi_GetAttributeFlags`
Set Attribute Flags	`Ivi_SetAttributeFlags`
Get Attribute Type	`Ivi_GetAttributeType`
Get Attribute Name	`Ivi_GetAttributeName`
Get Invalidation List	`Ivi_GetInvalidationList`
Dispose Invalidation List	`Ivi_DisposeInvalidationList`

Table 11-1. IVI Library Function Tree (Continued)

Function Tree	Function Name
Set/Get/Check Attribute (continued)	
Helper Functions (continued)	
Attribute Information (continued)	
Attribute Is Cached	`Ivi_AttributeIsCached`
Attribute Update Is Pending	`Ivi_AttributeUpdateIsPending`
Get Next Coercion Info	`Ivi_GetNextCoercionInfo`
Get Attr Min Max ViInt32	`Ivi_GetAttrMinMaxViInt32`
Get Attr Min Max ViReal64	`Ivi_GetAttrMinMaxViReal64`
Configuration	
Configuration File	
Get ivi.ini Directory Path	`Ivi_GetIviIniDir`
Set ivi.ini Directory Path	`Ivi_SetIviIniDir`
Logical Names	
Get Logical Names List	`Ivi_GetLogicalNamesList`
Get Nth Logical Name	`Ivi_GetNthLogicalName`
Dispose Logical Names List	`Ivi_DisposeLogicalNamesList`
Run-Time Configuration	
Define Logical Name	`Ivi_DefineLogicalName`
Undefine Logical Name	`Ivi_UndefLogicalName`
Define VInstr	`Ivi_DefineVInstr`
Undefine VInstr	`Ivi_UndefVInstr`
Define Driver	`Ivi_DefineDriver`
Undefine Driver	`Ivi_UndefDriver`
Define Hardware	`Ivi_DefineHardware`
Undefine Hardware	`Ivi_UndefHardware`
Define Class	`Ivi_DefineClass`
Undefine Class	`Ivi_UndefClass`
Write Run-Time Defines to File	`Ivi_WriteRunTimeDefinesToFile`

The following describes the top-level classes in the function tree:

- **Instrument Driver Session**—Contains functions instrument drivers use to create, validate, lock, and unlock IVI sessions.
- **Channels**—Contains functions that define the valid channel strings for an instrument and convert virtual channel names to channel strings.
- **Attribute Creation**—Contains functions that add, configure, and delete attributes and the callback functions they use.
- **Set/Get/Check Attribute**—Contains the key functions that set, get, and check values of attributes.
- **Caching/State-Checking Control**—Contains functions that instrument drivers use to control or query certain IVI engine internal data involving caching and status checking.
- **Range Tables**—Contains functions that access attribute range tables and their entries, change attribute range tables, and dynamically create range tables.

- **Error Information**—Contains functions that report and retrieve error information.
- **Memory Allocation**—Contains functions that allocate and deallocate memory blocks and associate them with an IVI session.
- **Helper Functions**—Contains utility functions that facilitate instrument driver and application program development.
- **Configuration**—Contains functions that get and set the directory of the IVI configuration file, return the logical names the IVI engine currently recognizes, and create run-time configuration entries.

The online help for each panel contains specific information about the function and its parameters.

Error Reporting

The IVI Library has an extensive mechanism for reporting errors. Almost all functions in the IVI Library return a negative status code if an error occurs and return `VI_SUCCESS` (0) if the function succeeds. A few functions return positive values to indicate warnings. The `Ivi_GetAttributeViString` function returns a positive value if the buffer you pass is not large enough to hold the current attribute value. The positive value indicates size of the buffer you must pass to obtain the complete value.

The IVI Library functions return error and warning values from several sets of status codes. Some status codes are unique to the IVI Library. Other status codes are the same codes that VISA Library functions return. Still others are error or warning values that functions in specific instrument drivers return. Each set of status codes has its own numeric range. The *Status Codes* section at the end of this chapter lists the numeric ranges of the different sets of status codes. It also contains a listing of all the IVI error codes and the most commonly used VISA status codes.

Each IVI session has the following three attributes for reporting error information:

```
IVI_ATTR_PRIMARY_ERROR
IVI_ATTR_SECONDARY_ERROR
IVI_ATTR_ERROR_ELABORATION
```

Each instrument driver defines its own constant name for these attributes, with the instrument prefix replacing `IVI` in the name.

You can call `Ivi_SetErrorInfo` to set all three attributes at once. You can call `Ivi_ClearErrorInfo` to clear all three attributes at once. You can call `Ivi_GetErrorInfo` to obtain and then clear the values of all three attributes at once. Each instrument driver exports a *`Prefix_`* version of each of the `Get` and `Clear` functions.

You also can access the attribute values using the following functions:

- `Ivi_SetAttributeViInt32` to set primary or secondary error code
- `Ivi_SetAttributeViString` to set error elaboration string
- `Ivi_GetAttributeViInt32` to obtain primary or secondary error code
- `Ivi_GetAttributeViString` to obtain error elaboration string

The three attributes describe the first error or warning that occurred since the last call to `Ivi_GetErrorInfo` or `Ivi_ClearErrorInfo` on the session. The primary error code specifies the primary reason for the error or warning. If no error or warning occurred, the primary error code is `VI_SUCCESS (0)`. The secondary error code is optional and provides additional information about the error or warning condition. A value of 0 indicates no additional information. The error elaboration parameter is a string that can contain further descriptive information about the error or warning condition.

The IVI Library also maintains a primary error code, secondary error code, and error elaboration string for each execution thread. When you call `Ivi_SetErrorInfo` or `Ivi_ClearErrorInfo` on a session, the function sets or clears the error information for both the session and the thread. When you pass `VI_NULL` for the **vi** parameter to `Ivi_SetErrorInfo` or `Ivi_ClearErrorInfo`, the function sets or clears only the error information for the thread. This is useful when you do not have a session handle to pass, which occurs when a call to `Ivi_SpecificDriverNew` fails. To obtain the error information for the thread, you must call `Ivi_GetErrorInfo` with the **vi** parameter set to `VI_NULL`.

Normally, it is the responsibility of the user to decide when to clear the error information by calling *`Prefix`*`_GetErrorInfo` or *`Prefix`*`_ClearErrorInfo`. If an instrument driver calls `Ivi_GetErrorInfo`, it must restore the error information by calling `Ivi_SetErrorInfo`, possibly adding a secondary error code or an elaboration string.

`Ivi_SetErrorInfo` does not overwrite existing significant error information unless you request it to do so. This allows you to make multiple calls to `Ivi_SetErrorInfo` at different levels in your instrument driver source code without the risk of losing important error information. It also preserves the information about the first error for the user. Refer to the `Ivi_SetErrorInfo` function description for more information on this mechanism.

Error Macros

The `ivi.h` include file contains eight macros that you can use in your source code to facilitate error handling. All eight macros require that you have the following declaration at the top of the function in which the macro appears:

```
ViStatus error = VI_SUCCESS;
```

The macros also require that you have the following label near the end of the function:

```
Error:
```

Five of the macros require access to a ViSession variable named **vi**, which they pass to Ivi_SetErrorInfo in certain cases. The names of these macros all begin with viCheck.

Normally, you use the macros around function calls, but you also can use them around variables or expressions.

The following describes the behavior of each macro.

checkAlloc(pointer)

If **pointer** is VI_NULL, assign VI_ERROR_ALLOC to the error variable and jump to the Error label.

checkErr(status)

Assign **status** to the error variable. If **status** is positive, coerce the error variable to zero. If **status** is negative, jump to the Error label.

checkWarn(status)

Assign **status** to the error variable. If negative, jump to the Error label.

viCheckAlloc(pointer)

If **pointer** is VI_NULL, assign VI_ERROR_ALLOC to the error variable, call Ivi_SetErrorInfo with VI_ERROR_ALLOC as the primary error code, and jump to the Error label.

viCheckErr(status)

Assign status to the error variable. If **status** is positive, coerce the error variable to zero. If **status** is negative, pass it to Ivi_SetErrorInfo and jump to the Error label.

viCheckErrElab(status, elabString)

Assign **status** to the error variable. If **status** is positive, coerce the error variable to zero. If **status** is negative, pass it and **elabString** to Ivi_SetErrorInfo and jump to the Error label.

viCheckParm(status, parameterPosition, parameterName)

Assign **status** to the `error` variable. If **status** is positive, coerce the `error` variable to zero. If **status** is negative, do the following:

- Convert **parameterPosition** into one of the VXI*plug&play* error codes for invalid parameters, and pass it as the secondary error code to `Ivi_SetErrorInfo`. Pass **status** as the primary error code, and pass **parameterName** as the error elaboration.
- Jump to the `Error` label.

viCheckWarn(status)

Assign **status** to error. If **status** is nonzero, pass it to `Ivi_SetErrorInfo`. If **status** is negative, jump to the `Error` label.

Notice that the `checkWarn` and `viCheckWarn` macros preserve warnings whereas the other `viCheck` macros discard them. Also, `viCheckWarn` calls `Ivi_SetErrorInfo` on both warnings and errors, whereas the other macros call `Ivi_SetErrorInfo` only on errors.

When to Use the viCheck Macros

When returning an error or a warning, each user-callable instrument driver function must set the error information for the session and thread. You can do this by explicitly calling `Ivi_SetErrorInfo` at the end of the function, or you can use the `viCheck` macros in the function or in the lower-level routines that the function calls.

You can call the `viCheck` macros only when the following two conditions are true:

- The function in which it appears has a `ViSession` parameter named **vi** that is an IVI session handle or `VI_NULL`.
- The first argument you pass to the macro is either a pointer value, in the case of `viCheckAlloc`, or a status code that is negative if and only if an error occurs. IVI and VISA functions return such status codes.

It is best to use the `viCheck` macros at the lowest level in your code where these two conditions are true. You can then use the `check` versions of the macros at higher levels. All IVI Library functions that take the IVI session handle as a parameter call `Ivi_SetErrorInfo` when they return errors. Thus, you do not have to use the `viCheck` macros around calls to IVI functions. Nevertheless, it is harmless to make redundant use of the `viCheck` macros. The `viCheck` macros call `Ivi_SetErrorInfo` in such a way that it does not overwrite existing significant error information.

Examples

The following example shows how to handle errors that calls to IVI functions return.

```
checkErr( Ivi_SetAttributeViSession (vi, VI_NULL,
                          IVI_ATTR_IO_SESSION, 0, io));
```

The following example shows how to handle errors that VISA functions return. This method also works for other libraries that return errors as negative values.

```
viCheckErr( viSetAttribute (io, VI_ATTR_TMO_VALUE, 5000 ));
```

The following example shows how report an error with an elaboration string.

```
if (triggerCount > 1 || sampleCount > 1)
   viCheckErrElab( IVI_ERROR_INVALID_CONFIGURATION,
                "Cannot use single point measurement functions"
                " when DMM is configured for multi-point.");
```

The following example shows how report a parameter error in a user-callable instrument driver function.

```
viCheckParm( Ivi_SetAttributeViReal64(vi, VI_NULL,
           HP34401_ATTR_RESOLUTION, 0, resolution),
           4, "Resolution");
```

IVI Library Function Reference

This section describes each function in the LabWindows/CVI IVI Library in alphabetical order.

Ivi_AddAttributeInvalidation

```
ViStatus status = Ivi_AddAttributeInvalidation (ViSession vi,
                           ViAttr attributeID, ViAttr dependentAttributeID,
                           ViBoolean allChannels);
```

Purpose

This function creates an invalidation dependency relationship between two attributes. When you set the first attribute to a new value, the IVI Library marks the cache value for the second attribute value as invalid. When an attribute cache value is invalid, any attempt to obtain or change the current value of the attribute causes the IVI Library to invoke the read or write callback function for the attribute regardless of the cache value.

Create a dependency relationship if setting the value of one attribute can cause the value of another attribute to change or become out-of-range in the instrument. When this occurs, the cache value of the second attribute no longer reflects the true state of the instrument.

Although you can create a two-way invalidation dependency relationship between attributes it is rarely the correct thing to do. Refer to the *IVI State-Caching Mechanism* section in Chapter 2, *IVI Architecture Overview*, for more information.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
attributeID	`ViAttr`	The ID of the attribute for which a change in value can render the cache value of the dependent attribute unreliable.
dependentAttributeID	`ViAttr`	The ID of the attribute to invalidate when the value of the other attribute changes.
allChannels	`ViBoolean`	Specifies whether the invalidation occurs on all possible channels or only on the channel on which the value of the first attribute changes. This option is relevant only if both attributes are channel-based.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

The **allChannels** parameter is relevant only if both attributes are channel-based. Pass `VI_TRUE` for **allChannels** if you want the invalidation to occur on all possible channels. Pass `VI_FALSE` if you want the invalidation to occur only on the channel on which the value of the first attribute changes.

See Also

`Ivi_AddAttributeInvalidation`, `Ivi_InvalidateAttribute`, `Ivi_InvalidateAllAttributes`, `Ivi_GetInvalidationList`

Ivi_AddAttributeViAddr

```
ViStatus status = Ivi_AddAttributeViAddr (ViSession vi,
                        ViAttr newAttributeID,
                        ViConstString attributeName,
                        ViAddr defaultValue, IviAttrFlags flags,
                        ReadAttrViAddr_CallbackPtr readCallback,
                        WriteAttrViAddr_CallbackPtr writeCallback);
```

Purpose

Creates and configures a new `ViAddr` attribute for an instrument session.

You can use `ViAddr` attributes only for settings that are private to the instrument driver. Do not make `ViAddr` attributes accessible to the user.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
newAttributeID	`ViAttr`	The ID for the new attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.
attributeName	`ViConstString`	A string that contains the defined constant name of the attribute. Refer to the *Parameter Discussion* section.
defaultValue	`ViAddr`	The default initial value for the attribute. Refer to the *Parameter Discussion* section.
flags	`IviAttrFlags`	Refer to the *Attribute Flags* section in Chapter 2, *IVI Architecture Overview*.

Name	Type	Description
readCallback	`ReadAttrViAddr_CallbackPtr`	The read callback function you want the IVI engine to invoke when you request the current value of the attribute. Refer to the *Parameter Discussion* section.
writeCallback	`WriteAttrViAddr_CallbackPtr`	The write callback function you want the IVI engine to invoke when you set the value of the attribute. Refer to the *Parameter Discussion* section.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

The **attributeName** string must always be the same as the defined constant name for the attribute. For example, if the defined constant for an attribute is `FL45_ATTR_RANGE`, pass `"FL45_ATTR_RANGE"` for **attributeName**.

The IVI engine uses the value you pass for **defaultValue** as the current value of the attribute if all of the following conditions are true:

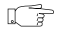
- The user sets the `IVI_ATTR_SIMULATION` attribute to `VI_TRUE`.
- You specify no read callback for the attribute, or the `IVI_VAL_USE_CALLBACKS_FOR_SIMULATION` attribute is `VI_TRUE`.
- You query the attribute value before you ever set it.

Read Callback

The IVI engine invokes the read callback function to obtain the current value of an attribute from the instrument. If you do not want to use a read callback function for the attribute, you can pass `VI_NULL` for **readCallback**. Otherwise, you must define the read callback function in the source code for the specific instrument driver and pass its address as the **readCallback** parameter. The read callback function must have the following prototype:

```
ViStatus _VI_FUNC ReadCallback(ViSession vi, ViSession io,
                         ViConstString channelName,
                         ViAttr attributeId, ViAddr *value);
```

Upon entry to the callback, ***value** contains the cache value. Upon exit from the callback, ***value** must contain the actual current value.



Note ***If you want to use the* Edit Instrument Attributes *command to develop your instrument driver source code, retain the parameter names as shown in the prototype for the callback.***

You can use `Ivi_SetAttrReadCallbackViAddr` to replace the read callback after you create the attribute.

Write Callback

The IVI engine invokes the write callback function to obtain the current value of an attribute from the instrument. If you do not want to use a write callback function for the attribute, you can pass `VI_NULL` for **writeCallback**. Otherwise, you must define the write callback function in the source code for the specific instrument driver and pass its address as the **writeCallback** parameter. The read callback function must have the following prototype:

You must define the write callback function in the source code for the specific instrument driver. The function must have the following prototype:

```
ViStatus _VI_FUNC WriteCallback(ViSession vi, ViSession io,
                                ViConstString channelName,
                                ViAttr attributeId, ViAddr value);
```

Note ***If you want to use the* Edit Instrument Attributes *command to develop your instrument driver source code, retain the parameter names as shown in the prototype for the callback.***

You can use `Ivi_SetAttrWriteCallbackViAddr` to replace the write callback after you create the attribute.

See Also

`Ivi_SetAttrReadCallbackViAddr`, `Ivi_SetAttrWriteCallbackViAddr`

Ivi_AddAttributeViBoolean

```
ViStatus status = Ivi_AddAttributeViBoolean (ViSession vi,
                         ViAttr newAttributeID,
                         ViConstString attributeName,
                         ViBoolean defaultValue, IviAttrFlags flags,
                         ReadAttrViBoolean_CallbackPtr readCallback,
                         WriteAttrViBoolean_CallbackPtr writeCallback);
```

Purpose

Creates and configures a new `ViBoolean` attribute for an instrument session.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
newAttributeID	`ViAttr`	The ID for the attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.
attributeName	`ViConstString`	A string that contains the defined constant name of the attribute. Refer to the *Parameter Discussion* section.
defaultValue	`ViBoolean`	The default initial value for the attribute. Refer to the *Parameter Discussion* section.
flags	`IviAttrFlags`	Refer to the *Attribute Flags* section in Chapter 2, *IVI Architecture Overview*.
readCallback	`ReadAttrViBoolean _CallbackPtr`	The read callback function you want the IVI engine to invoke when you request the current value of the attribute. Refer to the *Parameter Discussion* section.
writeCallback	`WriteAttr ViBoolean_ CallbackPtr`	The write callback function you want the IVI engine to invoke when you set the value of the attribute. Refer to the *Parameter Discussion* section.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

The **attributeName** string must always be the same as the defined constant name for the attribute. For example, if the defined constant for an attribute is `FL45_ATTR_RANGE`, pass `"FL45_ATTR_RANGE"` for **attributeName**.

The IVI engine uses the value you pass for **defaultValue** as the current value of the attribute if all of the following conditions are true:

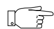
- The user sets the `IVI_ATTR_SIMULATION` attribute to `VI_TRUE`.
- You specify no read callback for the attribute, or the `IVI_VAL_USE_CALLBACKS_FOR_SIMULATION` attribute is `VI_TRUE`.
- You query the attribute value before you ever set it.

Read Callback

The IVI engine invokes the read callback function to obtain the current value of an attribute from the instrument. If you do not want to use a read callback function for the attribute, you can pass `VI_NULL` for **readCallback**. Otherwise, you must define the read callback function in the source code for the specific instrument driver and pass its address as the **readCallback** parameter. The read callback function must have the following prototype:

```
ViStatus _VI_FUNC ReadCallback(ViSession vi, ViSession io,
                           ViConstString channelName,
                           ViAttr attributeId, ViBoolean *value);
```

Upon entry to the callback, ***value** contains the cache value. Upon exit from the callback, ***value** must contain the actual current value.

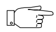
 **Note** *If you want to use the* **Edit Instrument Attributes** *command to develop your instrument driver source code, retain the parameter names as shown in the prototype for the callback.*

You can use `Ivi_SetAttrReadCallbackViBoolean` to replace the read callback function after you create the attribute.

Write Callback

The IVI engine invokes the write callback function to obtain the current value of an attribute from the instrument. If you do not want to use a write callback function for the attribute, you can pass `VI_NULL` for **writeCallback**. Otherwise, you must define the write callback function in the source code for the specific instrument driver and pass its address as the **writeCallback** parameter. The write callback function must have the following prototype:

```
ViStatus _VI_FUNC WriteCallback(ViSession vi, ViSession io,
                                ViConstString channelName,
                                ViAttr attributeId, ViBoolean value);
```

 **Note** ***If you want to use the* Edit Instrument Attributes *command to develop your instrument driver source code, retain the parameter names as shown in the prototype for the callback.***

You can use `Ivi_SetAttrWriteCallbackViBoolean` to replace the write callback function after you create the attribute.

See Also

`Ivi_SetAttrReadCallbackViBoolean`,
`Ivi_SetAttrWriteCallbackViBoolean`,
`Ivi_DefaultCoerceCallbackViBoolean`,
`Ivi_SetAttrCoerceCallbackViBoolean`

Ivi_AddAttributeViInt32

```
ViStatus status = Ivi_AddAttributeViInt32 (ViSession vi,
                        ViAttr newAttributeID,
                        ViConstString attributeName,
                        ViInt32 defaultValue, IviAttrFlags flags,
                        ReadAttrViInt32_CallbackPtr readCallback,
                        WriteAttrViInt32_CallbackPtr writeCallback,
                        IviRangeTablePtr rangeTable);
```

Purpose

Creates and configures a new `ViInt32` attribute for an instrument session.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
newAttributeID	`ViAttr`	The ID for the new attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.
attributeName	`ViConstString`	A string that contains the defined constant name of the attribute. Refer to the *Parameter Discussion* section.
defaultValue	`ViInt32`	The default initial value for the attribute. Refer to the *Parameter Discussion* section.
flags	`IviAttrFlags`	Refer to the *Attribute Flags* section in Chapter 2, *IVI Architecture Overview*.
readCallback	`ReadAttrViInt32_CallbackPtr`	The read callback function you want the IVI engine to invoke when you request the current value of the attribute. Refer to the *Parameter Discussion* section.

Name	Type	Description
writeCallback	`WriteAttrViInt32_CallbackPtr`	The write callback function you want the IVI engine to invoke when you set the value of the attribute. Refer to the *Parameter Discussion* section.
rangeTable	`IviRangeTablePtr`	The range table you want the IVI engine to use to validate and coerce values for this attribute. Refer to the Range Tables section in Chapter 2, *IVI Architecture Overview*.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

The **attributeName** string must always be the same as the defined constant name for the attribute. For example, if the defined constant for an attribute is `FL45_ATTR_RANGE`, pass `"FL45_ATTR_RANGE"` for **attributeName**.

The IVI engine uses the value you pass for **defaultValue** as the current value of the attribute if all of the following conditions are true:

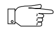
- The user sets the `IVI_ATTR_SIMULATION` attribute to `VI_TRUE`.
- You specify no read callback for the attribute, or the `IVI_VAL_USE_CALLBACKS_FOR_SIMULATION` attribute is `VI_TRUE`.
- You query the attribute value before you ever set it.

Read Callback

The IVI engine invokes the read callback function to obtain the current value of an attribute from the instrument. If you do not want to use a read callback function for the attribute, you can pass `VI_NULL` for **readCallback**. Otherwise, you must define the read callback function in the source code for the specific instrument driver and pass its address as the **readCallback** parameter. The read callback function must have the following prototype:

```
ViStatus _VI_FUNC ReadCallback(ViSession vi, ViSession io,
                               ViConstString channelName,
                               ViAttr attributeId, ViInt32 *value);
```

Upon entry to the callback, ***value** contains the cache value. Upon exit from the callback, ***value** must contain the actual current value.



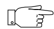
Note *If you want to use the* **Edit Instrument Attributes** *command to develop your instrument driver source code, retain the parameter names as shown in the prototype for the callback.*

You can use `Ivi_SetAttrReadCallbackViInt32` to replace the read callback function after you create the attribute.

Write Callback

The IVI engine invokes the write callback function to obtain the current value of an attribute from the instrument. If you do not want to use a write callback function for the attribute, you can pass `VI_NULL` for **writeCallback**. Otherwise, you must define the write callback function in the source code for the specific instrument driver and pass its address as the **writeCallback** parameter. The write callback function must have the following prototype:

```
ViStatus _VI_FUNC WriteCallback(ViSession vi, ViSession io,
                          ViConstString channelName,
                          ViAttr attributeId, ViInt32 value);
```



Note *If you want to use the* **Edit Instrument Attributes** *command to develop your instrument driver source code, retain the parameter names as shown in the prototype for the callback.*

You can use `Ivi_SetAttrWriteCallbackViInt32` to replace the write callback function after you create the attribute.

See Also

`Ivi_SetAttrWriteCallbackViInt32`, `Ivi_SetAttrWriteCallbackViInt32`, `Ivi_DefaultCheckCallbackViInt32`, `Ivi_SetAttrCheckCallbackViInt32`, `Ivi_DefaultCoerceCallbackViInt32`, `Ivi_SetAttrCheckCallbackViInt32`, `Ivi_SetAttrRangeTableCallback`, `Ivi_GetAttrRangeTable`, `Ivi_GetStoredRangeTablePtr`, `Ivi_SetStoredRangeTablePtr`

Ivi_AddAttributeViReal64

```
ViStatus status = Ivi_AddAttributeViReal64 (ViSession vi,
                           ViAttr newAttributeID,
                           ViConstString attributeName,
                           ViReal64 defaultValue, IviAttrFlags flags,
                           ReadAttrViReal64_CallbackPtr readCallback,
                           WriteAttrViReal64_CallbackPtr writeCallback,
                           IviRangeTablePtr rangeTable,
                           ViInt32 comparePrecision);
```

Purpose

Creates and configures a new `ViReal64` attribute for an instrument session.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
newAttributeID	`ViAttr`	The ID for the new attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.
attributeName	`ViConstString`	A string that contains the defined constant name of the attribute. Refer to the *Parameter Discussion* section.
defaultValue	`ViReal64`	The default initial value for the attribute. Refer to the *Parameter Discussion* section.
flags	`IviAttrFlags`	Refer to the *Attribute Flags* section in Chapter 2, *IVI Architecture Overview*.
readCallback	`ReadAttrViReal64_CallbackPtr`	The read callback function you want the IVI engine to invoke when you request the current value of the attribute. Refer to the *Parameter Discussion* section.

Name	Type	Description
writeCallback	`WriteAttrViReal64_CallbackPtr`	The write callback function you want the IVI engine to invoke when you set the value of the attribute. Refer to the *Parameter Discussion* section.
rangeTable	`IviRangeTablePtr`	The range table you want the IVI engine to use to validate and coerce values for this attribute. Refer to the *Range Tables* section in Chapter 2, *IVI Architecture Overview*.
comparePrecision	`ViInt32`	The degree of decimal precision the default IVI compare callback function uses for the attribute. Refer to the *Comparison Precision* section in Chapter 2, *IVI Architecture Overview*. Valid Range: 0, 1 to 14. If you pass 0, the function uses 14.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

The **attributeName** string must always be the same as the defined constant name for the attribute. For example, if the defined constant for an attribute is `FL45_ATTR_RANGE`, pass `"FL45_ATTR_RANGE"` for **attributeName**.

The IVI engine uses the value you pass for **defaultValue** as the current value of the attribute if all of the following conditions are true:

- The user sets the `IVI_ATTR_SIMULATION attribute to VI_TRUE`.
- You specify no read callback for the attribute, or the `IVI_VAL_USE_CALLBACKS_FOR_SIMULATION` attribute is `VI_TRUE`.
- You query the attribute value before you ever set it.

Read Callback

The IVI engine invokes the read callback function to obtain the current value of an attribute from the instrument. If you do not want to use a read callback function for the attribute, you can pass VI_NULL for **readCallback**. Otherwise, you must define the read callback function in the source code for the specific instrument driver and pass its address as the **readCallback** parameter. The read callback function must have the following prototype:

```
ViStatus _VI_FUNC ReadCallback(ViSession vi, ViSession io,
                               ViConstString channelName,
                               ViAttr attributeId, ViReal64 *value);
```

Upon entry to the callback, ***value** contains the cache value. Upon exit from the callback, ***value** must contain the actual current value.

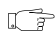
Note *If you want to use the* **Edit Instrument Attributes** *command to develop your instrument driver source code, retain the parameter names as shown in the prototype for the callback.*

You can use Ivi_SetAttrReadCallbackViReal64 to replace the read callback function after you create the attribute.

Write Callback

The IVI engine invokes the write callback function to obtain the current value of an attribute from the instrument. If you do not want to use a write callback function for the attribute, you can pass VI_NULL for **writeCallback**. Otherwise, you must define the write callback function in the source code for the specific instrument driver and pass its address as the **writeCallback** parameter. The write callback function must have the following prototype:

```
ViStatus _VI_FUNC WriteCallback(ViSession vi, ViSession io,
                                ViConstString channelName,
                                ViAttr attributeId, ViReal64 value);
```

 **Note** *If you want to use the* **Edit Instrument Attributes** *command to develop your instrument driver source code, retain the parameter names as shown in the prototype for the callback.*

You can use Ivi_SetAttrWriteCallbackViReal64 to replace the write callback function after you create the attribute.

See Also

Ivi_SetAttrReadCallbackViReal64, Ivi_SetAttrWriteCallbackViReal64, Ivi_DefaultCheckCallbackViReal64, Ivi_SetAttrCheckCallbackViReal64, Ivi_DefaultCoerceCallbackViReal64, Ivi_SetAttrCoerceCallbackViReal64, Ivi_DefaultCompareCallbackViReal64,

`Ivi_SetAttrCompareCallbackViReal64`, `Ivi_SetAttrRangeTableCallback`, `Ivi_GetAttrRangeTable`, `Ivi_GetStoredRangeTablePtr`, `Ivi_SetStoredRangeTablePtr`, `Ivi_SetAttrComparePrecision`, `Ivi_GetAttrComparePrecision`

Ivi_AddAttributeViSession

```
ViStatus status = Ivi_AddAttributeViSession (ViSession vi,
                          ViAttr newAttributeID,
                          ViConstString attributeName,
                          ViSession defaultValue, IviAttrFlags flags,
                          ReadAttrViSession_CallbackPtr readCallback,
                          WriteAttrViSession_CallbackPtr writeCallback);
```

Purpose

Creates and configures a new `ViSession` attribute for an instrument session.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
newAttributeID	`ViAttr`	The ID for the new attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.
attributeName	`ViConstString`	A string that contains the defined constant name of the attribute. Refer to the *Parameter Discussion* section.
defaultValue	`ViSession`	The default initial value for the attribute. Refer to the *Parameter Discussion* section.
flags	`IviAttrFlags`	Refer to the *Attribute Flags* section in Chapter 2, *IVI Architecture Overview*.
readCallback	`ReadAttrViSession_ CallbackPtr`	The read callback function you want the IVI engine to invoke when you request the current value of the attribute. Refer to the *Parameter Discussion* section.
writeCallback	`WriteAttr ViSession_ CallbackPtr`	The write callback function you want the IVI engine to invoke when you set the value of the attribute. Refer to the *Parameter Discussion* section.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

The **attributeName** string must always be the same as the defined constant name for the attribute. For example, if the defined constant for an attribute is `FL45_ATTR_RANGE`, pass `"FL45_ATTR_RANGE"` for **attributeName**.

The IVI engine uses the value you pass for **defaultValue** as the current value of the attribute if all of the following conditions are true:

- The user sets the `IVI_ATTR_SIMULATION` attribute to `VI_TRUE`.
- You specify no read callback for the attribute, or the `IVI_VAL_USE_CALLBACKS_FOR_SIMULATION` attribute is `VI_TRUE`.
- You query the attribute value before you ever set it.

Read Callback

The IVI engine invokes the read callback function to obtain the current value of an attribute from the instrument. If you do not want to use a read callback function for the attribute, you can pass `VI_NULL` for **readCallback**. Otherwise, you must define the read callback function in the source code for the specific instrument driver and pass its address as the **readCallback** parameter. The read callback function must have the following prototype:

```
ViStatus _VI_FUNC ReadCallback(ViSession vi, ViSession io,
                           ViConstString channelName,
                           ViAttr attributeId, ViSession *value);
```

Upon entry to the callback, ***value** contains the cache value. Upon exit from the callback, ***value** must contain the actual current value.

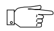
Note *If you want to use the* **Edit Instrument Attributes** *command to develop your instrument driver source code, retain the parameter names as shown in the prototype for the callback.*

You can use `Ivi_SetAttrReadCallbackViSession` to replace the read callback function after you create the attribute.

Write Callback

The IVI engine invokes the write callback function to obtain the current value of an attribute from the instrument. If you do not want to use a write callback function for the attribute, you can pass `VI_NULL` for **writeCallback**. Otherwise, you must define the write callback function in the source code for the specific instrument driver and pass its address as the **writeCallback** parameter. The write callback function must have the following prototype:

```
ViStatus _VI_FUNC WriteCallback(ViSession vi, ViSession io,
                                ViConstString channelName,
                                ViAttr attributeId, ViSession value);
```

 **Note** *If you want to use the* **Edit Instrument Attributes** *command to develop your instrument driver source code, retain the parameter names as shown in the prototype for the callback.*

You can use `Ivi_SetAttrWriteCallbackViSession` to replace the write callback function after you create the attribute.

See Also

`Ivi_SetAttrReadCallbackViSession`, `Ivi_SetAttrWriteCallbackViSession`

Ivi_AddAttributeViString

```
ViStatus status = Ivi_AddAttributeViString (ViSession vi,
                          ViAttr newAttributeID,
                          ViConstString attributeName,
                          ViConstString defaultValue, IviAttrFlags flags,
                          ReadAttrViString_CallbackPtr readCallback,
                          WriteAttrViString_CallbackPtr writeCallback);
```

Purpose

Creates and configures a new `ViString` attribute for an instrument session.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
newAttributeID	`ViAttr`	The ID for the new attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.
attributeName	`ViConstString`	A string that contains the defined constant name of the attribute. Refer to the *Parameter Discussion* section.
defaultValue	`ViConstString`	The default initial value for the attribute. Refer to the *Parameter Discussion* section.
flags	`IviAttrFlags`	Refer to the *Attribute Flags* section in Chapter 2, *IVI Architecture Overview*.
readCallback	`ReadAttrViString _CallbackPtr`	The read callback function you want the IVI engine to invoke when you request the current value of the attribute. Refer to the *Parameter Discussion* section.
writeCallback	`WriteAttrViString _CallbackPtr`	The write callback function you want the IVI engine to invoke when you set the value of the attribute. Refer to the *Parameter Discussion* section.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

The **attributeName** string must always be the same as the defined constant name for the attribute. For example, if the defined constant for an attribute is `FL45_ATTR_RANGE`, pass `"FL45_ATTR_RANGE"` for **attributeName**.

The IVI engine uses the value you pass for **defaultValue** as the current value of the attribute if all of the following conditions are true:

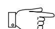
- The user sets the `IVI_ATTR_SIMULATION` attribute to `VI_TRUE`.
- You specify no read callback for the attribute, or the `IVI_VAL_USE_CALLBACKS_FOR_SIMULATION` attribute is `VI_TRUE`.
- You query the attribute value before you ever set it.

Read Callback

The IVI engine invokes the read callback function to obtain the current value of an attribute from the instrument. If you do not want to use a read callback function for the attribute, you can pass `VI_NULL` for **readCallback**. Otherwise, you must define the read callback function in the source code for the specific instrument driver and pass its address as the **readCallback** parameter. The read callback function must have the following prototype:

```
ViStatus _VI_FUNC ReadCallback(ViSession vi, ViSession io,
                          ViConstString channelName,
                          ViAttr attributeId,
                          const ViConstString *value);
```

Unlike the read callback functions for the other data types, the read callback for a `ViString` attribute does not report the current value to the caller through the last parameter. Instead, it reports the current value by passing it to `Ivi_SetValInStringCallback`.

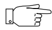
 **Note** ***If you want to use the* Edit Instrument Attributes *command to develop your instrument driver source code, retain the parameter names as shown in the prototype for the callback.***

You can use `Ivi_SetAttrReadCallbackViString` to replace the read callback function after you create the attribute.

Write Callback

The IVI engine invokes the write callback function to obtain the current value of an attribute from the instrument. If you do not want to use a write callback function for the attribute, you can pass `VI_NULL` for **writeCallback**. Otherwise, you must define the write callback function in the source code for the specific instrument driver and pass its address as the **writeCallback** parameter. The write callback function must have the following prototype:

```
ViStatus _VI_FUNC WriteCallback(ViSession vi, ViSession io,
                          ViConstString channelName,
                          ViAttr attributeId, ViConstString value);
```



Note *If you want to use the* **Edit Instrument Attributes** *command to develop your instrument driver source code, retain the parameter names as shown in the prototype for the callback.*

You can use `Ivi_SetAttrWriteCallbackViString` to replace the write callback function after you create the attribute.

See Also

`Ivi_SetValInStringCallback`, `Ivi_SetAttrReadCallbackViString`, `Ivi_SetAttrWriteCallbackViString`

Ivi_AddToChannelTable

```
ViStatus status = Ivi_AddToChannelTable
                        (ViSession vi, ViConstString channelStrings);
```

Purpose

Adds additional channel strings to the channel table you establish with `Ivi_BuildChannelTable`.

Refer to the *Channels* section in Chapter 2, *IVI Architecture Overview*, for more information on channel strings.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
channelStrings	`ViConstString`	A string containing a the list of channel strings you want to add to the channel table. Refer to the *Parameter Discussion* section.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

In the **channelStrings** parameter, you must separate multiple channel strings with commas. You can include spaces after the commas. For example, to add `"3"` and `"4"` as valid channel strings for the instrument session, you can pass `"3,4"`.

See Also

`Ivi_BuildChannelTable`, `Ivi_RestrictAttrToChannels`, `Ivi_ValidateAttrForChannel`, `Ivi_CoerceChannelName`, `Ivi_GetUserChannelName`, `Ivi_GetNthChannelString`

Ivi_Alloc

```
ViStatus status = Ivi_Alloc (ViSession vi, ViInt32 memoryBlockSize,
                             ViAddr *memoryBlockPointer);
```

Purpose

This function allocates memory for an object of the size you specify and initializes all bytes to zero. If you specify a non-NULL IVI session handle, the function associates the memory block with the session by inserting it into the list of memory blocks the IVI engine maintains for the session.

You can call `Ivi_Free` to free the memory block. You can call `Ivi_FreeAll` to free all of the memory blocks that you allocate for the session with `Ivi_Alloc` or `Ivi_RangeTableNew`. When you call `Ivi_Dispose` on the session, it calls `Ivi_FreeAll` for you.

If the function cannot allocate the space or you specify 0 for the **memoryBlockSize** parameter, the function sets the **memoryBlockPointer** parameter to `VI_NULL` and returns an error.

Parameters

Input

Name	Type	Description
vi	`ViSession`	If you want to associate the memory block with a particular IVI session, pass the IVI session handle that you obtain from `Ivi_SpecificDriverNew`. Otherwise, pass `VI_NULL`.
memoryBlockSize	`ViInt32`	The number of bytes you want to allocate. Must be greater than zero.
memoryBlockPointer	`ViAddr`	Returns a pointer to the memory block the function allocates.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_Free`, `Ivi_FreeAll`, `Ivi_Dispose`, `Ivi_RangeTableNew`

Ivi_AttributeIsCached

```
ViStatus status = Ivi_AttributeIsCached (ViSession vi,
                        ViConstString channel, ViAttr attributeID,
                        ViBoolean *cached);
```

Purpose

Indicates whether the IVI engine believes that the cache value of the attribute accurately reflects the state of the instrument.

The function returns `VI_FALSE` if the `IVI_VAL_NEVER_CACHE` flag for the attribute is set, there is no value in the cache, or the IVI engine has invalidated the cache value.

Parameters

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. Otherwise, pass `VI_NULL`.
channel	`ViConstString`	If the attribute is channel-based, pass a channel string or virtual channel name. Otherwise, pass `VI_NULL` or an empty string.
attributeID	`ViAttr`	The ID of the attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.

Output

Name	Type	Description
cached	`ViBoolean`	`1 = VI_TRUE` - cache reflects instrument state. `0 = VI_FALSE` - cache might not reflect instrument state.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Ivi_AttributeUpdateIsPending

```
ViStatus status = Ivi_AttributeUpdateIsPending (ViSession vi,
                           ViConstString channel, ViAttr attributeID,
                           ViBoolean *updatePending);
```

Purpose

Indicates whether a deferred update is pending for a specific attribute on a specific channel. At most one deferred update can be pending for an attribute on a particular channel. Refer to the *Deferred Update* section in Chapter 2, *IVI Architecture Overview*.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
channel	`ViConstString`	If the attribute is channel-based, pass a channel string or virtual channel name. Otherwise, pass `VI_NULL` or an empty string.
attributeID	`ViAttr`	The ID of the attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.

Output

Name	Type	Description
updatePending	`ViBoolean`	`1 = VI_TRUE` - update pending. `0 = VI_FALSE` - no update pending.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Ivi_BuildChannelTable

```
ViStatus status = Ivi_BuildChannelTable (ViSession vi,
                     ViConstString channelStrings,
                     ViBoolean allowUnknownChannels,
                     Ivi_ValidateChannelStringFunc validationCallback);
```

Purpose

Creates the initial channel table for an IVI session. A channel table consists of the valid channel strings for the instrument session. When you create attributes with the `Ivi_AddAttribute` functions, you set the `IVI_VAL_MULTI_CHANNEL` flag for attributes that have different values for each channel. You use this function to specify the set of channels.

If the instrument does not support multiple channels, then create an initial channel table with `"1"` as the only valid channel string.

You must call `Ivi_BuildChannelTable` in your *`Prefix`*`_IviInit function`. If you call it again at a later point, it discards the old channel table and builds a new one. To add channel strings to an existing channel table, call `Ivi_AddToChannelTable`. To restrict an attribute to a subset of channels, call `Ivi_RestrictAttrToChannels`.

The IVI Library maintains the channel table for the session. If the user defines any virtual channel names in the `ivi.ini` configuration file, the IVI Library associates the virtual names with the entries in the table. Refer to the *Channels* section in Chapter 2, *IVI Architecture Overview*.

The `Ivi_CoerceChannelName` function validates channel names and converts virtual channel names to specific driver channel strings. If your driver supports multiple channels, you must call `Ivi_CoerceChannelName` in driver functions that use the channel string directly. When you call an `Ivi_SetAttribute`, `Ivi_GetAttribute`, or `Ivi_CheckAttribute` function the IVI engine converts virtual channel names to specific driver channel strings before invoking read, write, check, coerce, compare, and range table callback functions.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
channelStrings	`ViConstString`	A single string containing a list of the strings that represent the channels of the instrument. Refer to the *Parameter Discussion* section.
allowUnknownChannels	`ViBoolean`	If `VI_TRUE`, users can specify channel strings that are not in the channel table. Typically, you pass `VI_FALSE`. Refer to the *Parameter Discussion* section.
validationCallback	`Ivi_ValidateChannelStringFunc`	If you pass `VI_TRUE` for **allowUnknownChannels**, specify the address of a channel string validation function. Otherwise, pass `VI_NULL`. Refer to the *Parameter Discussion* section.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

In the **channelStrings** parameter, separate the channel strings with commas. You can include spaces after the commas. For example, if the instrument has two channels and your driver accepts `"1"` or `"2"` as the channel string, you can pass `"1, 2"`.

Most instrument drivers use only a small number of channels, and the channels have simple names. A few drivers, however, allow a large number of channels, and the channel strings are complex. In the latter case, it is not practical for the driver to specify all of the possible channel strings in advance. Instead, the driver uses `Ivi_BuildChannelTable` to specify

only a basic set of channels. The driver calls Ivi_CoerceChannelName at the beginning of each user-callable function to determine if the channel parameter is a channel string or virtual channel name already in the channel table. If it is not in the table but it is a valid channel string, the driver calls Ivi_AddToChannelTable to add the channel string to the channel table.

If you pass VI_TRUE for **allowUnknownChannels**, you must pass the address a channel string validation function for the **validationCallback** parameter. Ivi_BuildChannelTable invokes the validation callback when a virtual channel name the user specifies in the ivi.ini configuration file refers to a channel string that is not in the **channelStrings** parameter. The channel string validation callback must have the following prototype:

```
ViStatus _VI_FUNC CheckChanStr(ViSession vi,
                           ViConstString channelString,
                           ViBoolean *isValid,
                           ViStatus *secondaryError);
```

The validation callback indicates whether the channel string is valid through the **isValid** parameter. If the channel string is invalid, the callback can specify a particular reason through the **secondaryError** parameter. Normally, the callback returns VI_SUCCESS even if the channel string is invalid. The callback returns an error code only if it cannot perform the validation.

See Also

Ivi_AddToChannelTable, Ivi_RestrictAttrToChannels, Ivi_CoerceChannelName, Ivi_ValidateAttrForChannel, Ivi_GetUserChannelName, Ivi_GetNthChannelString

Ivi_CheckAttributeViInt32
Ivi_CheckAttributeViReal64
Ivi_CheckAttributeViString
Ivi_CheckAttributeViBoolean
Ivi_CheckAttributeViSession
Ivi_CheckAttributeViAddr

```
ViStatus status = Ivi_CheckAttributeViInt32 (ViSession vi,
                           ViConstString channel, ViAttr attributeID,
                           ViInt32 optionFlags, ViInt32 attributeValue);
ViStatus status = Ivi_CheckAttributeViReal64 (ViSession vi,
                           ViConstString channel, ViAttr attributeID,
                           ViInt32 optionFlags, ViReal64 attributeValue);
ViStatus status = Ivi_CheckAttributeViString (ViSession vi,
                           ViConstString channel, ViAttr attributeID,
                           ViInt32 optionFlags,
                           ViConstString attributeValue);
ViStatus status = Ivi_CheckAttributeViBoolean (ViSession vi,
                           ViConstString channel, ViAttr attributeID,
                           ViInt32 optionFlags, ViBoolean attributeValue);
ViStatus status = Ivi_CheckAttributeViSession (ViSession vi,
                           ViConstString channel, ViAttr attributeID,
                           ViInt32 optionFlags, ViSession attributeValue);
ViStatus status = Ivi_CheckAttributeViAddr (ViSession vi,
                           ViConstString channel, ViAttr attributeID,
                           ViInt32 optionFlags, ViAddr attributeValue);
```

Purpose

Checks the validity of a value you specify for an attribute. A separate typesafe function exists for each possible attribute data type.

Each function performs the following actions:

- Checks whether the attribute is writable. If not, the function returns an error.
- Validates the **attributeValue** parameter. If you provide a check callback for the attribute, the function invokes the check callback to validate the value. If you do not provide a check callback but the data type of the attribute is `ViInt32` or `ViReal64` and you provide a range table or a range table callback for the attribute, the function invokes the default IVI check callback validate the value. If the value is invalid, the function returns an error.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
channel	`ViConstString`	If the attribute is channel-based, pass a channel string or virtual channel name. Otherwise, pass `VI_NULL` or an empty string.
attributeID	`ViAttr`	The ID of the attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.
optionFlags	`ViInt32`	`0` or `IVI_VAL_DIRECT_USER_CALL`. Refer to the *Parameter Discussion* section.
attributeValue	depends on the data type of the attribute	The value which you want to verify as a valid value for the attribute given the current settings of the instrument session.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. A positive value indicates a warning. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

Normally you pass 0 for **optionFlags**. Pass `IVI_VAL_DIRECT_USER_CALL` only when calling this function to implement one of the `Prefix_CheckAttribute` functions that your instrument driver exports to the user. When you pass `IVI_VAL_DIRECT_USER_CALL`, the function returns an error if the `IVI_VAL_NOT_WRITABLE` or `IVI_VAL_NOT_USER_WRITABLE` flag for the attribute is set.

Ivi_CheckBooleanRange

```
ViStatus status = Ivi_CheckBooleanRange (ViBoolean value,
                           ViStatus errorCode);
```

Purpose

Verifies that a `ViBoolean` value is either `VI_TRUE (1)` or `VI_FALSE (0)`. If it is not, the function returns an error code that you specify.

Parameters

Name	Type	Description
value	`ViBoolean`	The value you want to validate.
errorCode	`ViStatus`	The error code the function returns if value is not `VI_TRUE` or `VI_FALSE`

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. A positive value indicates a warning. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_CoerceBoolean`, `Ivi_CheckNumericRange`

Ivi_CheckNumericRange

```
ViStatus status = Ivi_CheckNumericRange (ViReal64 value, ViReal64 minimum,
                            ViReal64 maximum, ViStatus errorCode);
```

Purpose

Verifies that a `ViInt32` or `ViReal64` value falls within a specific range. If it is does not fall within the range, the function returns an error code that you specify.

The range is inclusive. In other words, the function returns the error code if the value is less than the minimum value or greater than the maximum value.

When you use this function on a parameter to a user-callable function in an instrument driver, use the `viCheckParm` macro around the function call. The following example shows how to use the `viCheckParm` macro around this function:

```
viCheckParm( CheckNumericRange(parmVal,min,max,errorCode),
                        parmPosition, parmName);
```

In this example, **parmPosition** is the 1-based position of the parameter in the parameter list of the user-callable function, and **parmName** is a string that contains the name of the parameter. `Ivi_CheckNumericRange` stores the **errorCode** you pass to it as the primary error code. `viCheckParm` converts **parmPosition** to one of the VXI*plug&play* error codes for invalid parameters and stores it as the secondary error code. It stores **parmName** as the error elaboration string. Refer to the *Error Reporting* and *Error Macros* sections earlier in this chapter for more information.

Parameters

Name	Type	Description
value	`ViReal64`	The value you want to check.
minimum	`ViReal64`	The minimum value of the range.
maximum	`ViReal64`	The maximum value of the range.
errorCode	`ViStatus`	The error code the function returns if **value** does not fall within the range.

Return Value

Name	Type	Description
status	`ViStatus`	errorCode if value is not valid. 0 if value is valid.

See Also

`Ivi_CheckBooleanRange`, `Ivi_CoerceBoolean`, `Ivi_CompareWithPrecision`

Ivi_ClearErrorInfo

```
ViStatus status = Ivi_ClearErrorInfo (ViSession vi);
```

Purpose

Clears the error information for the current execution thread and the IVI session you specify. If you pass `VI_NULL` for the **vi** parameter, this function clears the error information only for the current execution thread.

Instrument drivers export this function to the user through the `*Prefix*_ClearErrorInfo` function. Normally, the error information describes the first error that occurred since the user last called `*Prefix*_GetErrorInfo` or `*Prefix*_ClearErrorInfo`.

Avoid calling this function except to implement the `*Prefix*_ClearErrorInfo` function. Normally, it is the responsibility of the user to decide when to clear the error information. `Ivi_GetErrorInfo`, which the user calls through `*Prefix*_GetErrorInfo`, always clears the error information.

The error information includes a primary error code, secondary code error, and an error elaboration string. This function sets the primary and secondary error codes to `VI_SUCCESS (0)`, and sets the error elaboration string to `""`.

Refer to the *Error Reporting* section earlier in this chapter for details on the IVI error information.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew` or `VI_NULL`.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_SetErrorInfo`, `Ivi_GetErrorInfo`, `Ivi_GetErrorMessage`, `Ivi_GetSpecificDriverStatusDesc`

Ivi_ClearInstrSpecificErrorQueue

```
ViStatus status = Ivi_ClearInstrSpecificErrorQueue (ViSession vi);
```

Purpose

Removes all entries from the instrument-specific error queue for an IVI session.

Use the instrument-specific error queue if querying the instrument for its status causes the instrument to lose the error value. In your check status callback, call `Ivi_QueueInstrSpecificError` to insert the instrument error code in the queue, and then return the `IVI_ERROR_INSTR_SPECIFIC` error code from the callback. In your *`Prefix`*`_error_query` function, call `Ivi_InstrSpecificErrorQueueSize` to determine if there is an error in the queue. If not, invoke the check status callback directly. In either case, if there is an error, call `Ivi_DequeueInstrSpecificError` to retrieve the error.

Refer to the *Instruments without Error Queues* and the *Check Status Callback* sections of Chapter 2, *IVI Architecture Overview*, for more information.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_QueueInstrSpecificError`, `Ivi_DequeueInstrSpecificError`, `Ivi_InstrSpecificErrorQueueSize`

Ivi_CoerceBoolean

```
ViStatus status = Ivi_CoerceBoolean (ViBoolean *value);
```

Purpose

Coerces a value you specify to a valid `ViBoolean` value. If the value is non-zero, the function changes it to `VI_TRUE (1)`.

Parameters

Input/Output

Name	Type	Description
value	`ViBoolean`	The value you want to coerce.

Return Value

Name	Type	Description
status	`ViStatus`	`IVI_ERROR_INVALID_VALUE`, if value is `VI_NULL`. 0 otherwise.

See Also

`Ivi_CheckBooleanRange`

Ivi_CoerceChannelName

```
ViStatus status = Ivi_CoerceChannelName (ViSession vi,
                             ViConstString channelName,
                             ViConstString *channelString);
```

Purpose

Verifies that **channelName** is a valid channel string or virtual channel name. If it is a virtual channel name, `Ivi_CoerceChannelName` returns a pointer to specific driver channel string the virtual channel name represents.

Refer to the *Channels* section in Chapter 2, *IVI Architecture Overview*, for information on channel strings and virtual channel names.

If your driver supports multiple channels, you must call `Ivi_CoerceChannelName` in driver functions that use the channel string directly. When you call an `Ivi_SetAttribute`, `Ivi_GetAttribute`, or `Ivi_CheckAttribute` function, the IVI engine calls `Ivi_CoerceChannelName` internally before invoking the read, write, check, coerce, compare, and range table callback functions.

To be valid, **channelName** must be one of the following:

- `VI_NULL`, in which case the function returns `VI_NULL` in **channelString**.
- An empty string, in which case the function returns the address of that empty string in **channelString**.
- A specific driver channel string, in which case the function returns the address of the channel string. Specific instrument drivers define the valid channel strings using `Ivi_BuildChannelTable` and `Ivi_AddToChannelTable`.
- A virtual channel name that the user specifies in the `ivi.ini` configuration file. A virtual channel name is valid only if the user opens the session from a class driver and assigns a valid specific driver channel string to the virtual name in the `ivi.ini` file. If **channelName** is a virtual channel name, the function returns the address of the corresponding specific driver channel string.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
channelName	`ViConstString`	The channel name that you want to verify.

Output

Name	Type	Description
channelString	`ViConstString`	Returns a pointer to a specific driver channel string. Refer to the *Parameter Discussion* section.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

If an error occurs, the function does not modify the **channelString** parameter.

Caution ***Do not modify the contents of the string that* channelString *points to.***

If you want to use the `Ivi_CoerceChannelName` only to validate a channel name, you can pass `VI_NULL` for the **channelString** parameter.

See Also

`Ivi_BuildChannelTable`, `Ivi_AddToChannelTable`, `Ivi_RestrictAttrToChannels`, `Ivi_ValidateAttrForChannel`, `Ivi_GetUserChannelName`, `Ivi_GetNthChannelString`

Ivi_CompareWithPrecision

```
ViStatus status = Ivi_CompareWithPrecision (ViInt32 digitsOfPrecision,
                          ViReal64 a, ViReal64 b, ViInt32 *result);
```

Purpose

Compares two `ViReal64` values using the number of decimal digits of precision you specify.

If the two values are not exactly equal, the function uses the following logic, where **a** and **b** are the values you want to compare, and *d* is the digits of precision you specify.

if a == 0

  if $|b| < 10^{-(d-1)}$

    then a == b.

else /* a != 0 */

  if $\frac{|a-b|}{|a|} < 10^{-(d-1)}$

    then a == b

Parameters

Input

Name	Type	Description
digitsOfPrecision	`ViInt32`	Specify the number of decimal digits of precision you want to use to compare the two `ViReal64` values. Valid Range: 0, or 1 to 14. If you pass 0, the function sets the precision to the IVI default for this value, which is 14.
a	`ViReal64`	The first value you want to compare.
b	`ViReal64`	The second value you want to compare.

Output

Name	Type	Description
result	`viInt32`	Returns the result of the comparison. 0 if a == b –1 if a < b 1 if a > b

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_CheckNumericRange`

Ivi_DefaultBufferedIOCallback

```
ViStatus status = Ivi_DefaultBufferedIOCallback (ViSession vi, IviMessage
                          message);
```

Purpose

Responds to the messages that `Ivi_Update` sends when it processes deferred updates for the session. The IVI Library provides this default callback function to handle instruments that use VISA I/O.

You can change the buffered I/O callback by calling `Ivi_SetAttributeViAddr` function with the `IVI_ATTR_BUFFERED_IO_CALLBACK` attribute.

Refer to the *Deferred Updates* section in Chapter 2, *IVI Architecture Overview*, for detailed information on how `Ivi_Update` processes deferred updates, when it invokes the buffered I/O callback, and what `Ivi_DefaultBufferedIOCallback` does in response to each message.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
message	`IviMessage`	The type of action the callback takes. 1 = `IVI_MSG_START_UPDATE` 2 = `IVI_MSG_END_UPDATE` 3 = `IVI_MSG_SUSPEND` 4 = `IVI_MSG_RESUME` 5 = `IVI_MSG_FLUSH`

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_Update`

Ivi_DefaultCheckCallbackViInt32

```
ViStatus status = Ivi_DefaultCheckCallbackViInt32 (ViSession vi,
                      ViConstString channel, ViAttr attributeID,
                      ViInt32 attributeValue);
```

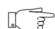
Purpose

Performs the default actions for checking the validity of a `ViInt32` attribute value. The IVI engine automatically installs this callback when you call `Ivi_AddAttributeViInt32` with a non-NULL range table or when you call `Ivi_SetAttrRangeTableCallback` to install a non-NULL range table callback function.

If you want to add to the actions of this callback, install your own callback with `Ivi_SetAttrCheckCallbackViInt32`, and invoke this function from your callback.

This function does the following:

1. Calls `Ivi_GetAttrRangeTable` to obtain the range table for the attribute. If the range table is invalid, the function returns an error. If there is no range table, the function returns `VI_SUCCESS`.
2. Calls `Ivi_GetViInt32EntryFromValue` to find an entry that matches the value.
3. Returns `VI_SUCCESS` if it can find an entry. Otherwise it returns an error.



Note ***Do not call this function directly unless you are calling it from your own check callback or you have already called*** `Ivi_LockSession`**.**

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
channel	`ViConstString`	If the attribute is channel-based, pass a channel string or virtual channel name. Otherwise, pass `VI_NULL` or an empty string.
attributeID	`ViAttr`	The ID of the attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.
attributeValue	`ViInt32`	The value you want to validate.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates the value is invalid or that some other error occurred. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_AddAttributeViInt32`, `Ivi_SetAttrCheckCallbackViInt32`, `Ivi_SetAttrRangeTableCallback`, `Ivi_GetAttrRangeTable`, `Ivi_GetStoredRangeTablePtr`, `Ivi_SetStoredRangeTablePtr`, `Ivi_GetViInt32EntryFromValue`

Ivi_DefaultCheckCallbackViReal64

```
ViStatus status = Ivi_DefaultCheckCallbackViReal64 (ViSession vi,
                      ViConstString channel, ViAttr attributeID,
                      ViReal64 attributeValue);
```

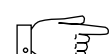
Purpose

Performs the default actions for checking the validity of a `ViReal64` attribute value. The IVI Library automatically installs this callback when you call `Ivi_AddAttributeViReal64` with a non-NULL range table or when you call `Ivi_SetAttrRangeTableCallback` to install a non-NULL range table callback function.

If you want to add to the actions of this callback, install your own callback with `Ivi_SetAttrCheckCallbackViReal64`, and invoke this function from your callback.

This function does the following:

1. Calls `Ivi_GetAttrRangeTable` to obtain the range table for the attribute. If the range table is invalid, the function returns an error. If there is no range table, the function returns `VI_SUCCESS`.
2. Calls `Ivi_GetViReal64EntryFromValue` to find an entry that matches the value.
3. Returns `VI_SUCCESS` if it can find an entry. Otherwise it returns an error.

 **Note** ***Do not call this function directly unless you are calling it from your own check callback or you have already called*** `Ivi_LockSession`.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
channel	`ViConstString`	If the attribute is channel-based, pass a channel string or virtual channel name. Otherwise, pass `VI_NULL` or an empty string.
attributeID	`ViAttr`	The ID of the attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.
attributeValue	`ViReal64`	The value you want to validate.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates the value is invalid or that some other error occurred. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_AddAttributeViReal64`, `Ivi_SetAttrCheckCallbackViReal64`, `Ivi_SetAttrRangeTableCallback`, `Ivi_GetAttrRangeTable`, `Ivi_GetStoredRangeTablePtr`, `Ivi_SetStoredRangeTablePtr`, `Ivi_GetViReal64EntryFromValue`

Ivi_DefaultCoerceCallbackViBoolean

```
ViStatus status = Ivi_DefaultCoerceCallbackViBoolean (ViSession vi,
                        ViConstString channel, ViAttr attributeID,
                        ViBoolean attributeValue,
                        ViBoolean *coercedValue);
```

Purpose

Performs the default actions for coercing a value for a `ViBoolean` attribute. The IVI Library automatically installs this callback when you call `Ivi_AddAttributeViBoolean`.

You can install your own coerce callback by calling `Ivi_SetAttrCoerceCallbackViBoolean`.

This function sets the coercedValue parameter to `VI_TRUE (1)` if the value you specify for **attributeValue** non-zero.

Note ***Do not call this function directly unless you are calling it from your own coerce callback or you have already called*** `Ivi_LockSession`**.**

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
channel	`ViConstString`	If the attribute is channel-based, pass a channel string or virtual channel name. Otherwise, pass `VI_NULL` or an empty string.
attributeID	`ViAttr`	The ID of the attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.
attributeValue	`ViBoolean`	The value you want to coerce.

Output

Name	Type	Description
coercedValue	`ViBoolean`	Returns the coerced value.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_AddAttributeViBoolean`, `Ivi_SetAttrCoerceCallbackViBoolean`

Ivi_DefaultCoerceCallbackViInt32

```
ViStatus status = Ivi_DefaultCoerceCallbackViInt32 (ViSession vi,
                         ViConstString channel, ViAttr attributeID,
                         ViInt32 attributeValue, ViInt32 *coercedValue);
```

Purpose

Performs the default actions for coercing a value for a `ViInt32` attribute. The IVI Library automatically installs this callback when you call `Ivi_AddAttributeViInt32` with a range table that has a type of `IVI_VAL_COERCED` or when you call `Ivi_SetAttrRangeTableCallback` to install a non-NULL range table callback function.

You can install your own coerce callback by calling `Ivi_SetAttrCoerceCallbackViInt32`.

This function does the following:

1. Calls `Ivi_GetAttrRange`Table to obtain the range table for the attribute. If the range table is invalid, the function returns an error.
2. If there is no range table or its type is not `IVI_VAL_COERCED`, the function sets the **coercedValue** parameter to the value you specify for **attributeValue**.
3. Calls `Ivi_GetViInt32EntryFromValue` to find an entry that matches the value.
4. If it can find an entry, it sets the **coercedValue** parameter to the value of the `coercedValue` field in the range table entry and returns `VI_SUCCESS`. Otherwise it returns an error.

Note ***Do not call this function directly unless you are calling it from your own coerce callback or you have already called*** `Ivi_LockSession`.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
channel	`ViConstString`	If the attribute is channel-based, pass a channel string or virtual channel name. Otherwise, pass `VI_NULL` or an empty string.
attributeID	`ViAttr`	The ID of the attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.
attributeValue	`ViInt32`	The value you want to coerce.

Output

Name	Type	Description
coercedValue	`ViInt32`	Returns the value to which the function coerces the input value based on the range table for the attribute.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_AddAttributeViInt32`, `Ivi_SetAttrCheckCallbackViInt32`, `Ivi_SetAttrRangeTableCallback`, `Ivi_GetAttrRangeTable`, `Ivi_GetStoredRangeTablePtr`, `Ivi_SetStoredRangeTablePtr`, `Ivi_GetViInt32EntryFromValue`

Ivi_DefaultCoerceCallbackViReal64

```
ViStatus status = Ivi_DefaultCoerceCallbackViReal64 (ViSession vi,
                        ViConstString channel, ViAttr attributeID,
                        ViReal64 attributeValue,ViReal64 *coercedValue);
```

Purpose

This function performs the default actions for coercing a value for a `ViReal64` attribute. The IVI Library automatically installs this callback when you call `Ivi_AddAttributeViReal64` with a range table that has a type of `IVI_VAL_COERCED` or when you call `Ivi_SetAttrRangeTableCallback` to install a non-NULL range table callback function.

You can install your own coerce callback by calling `Ivi_SetAttrCoerceCallbackViReal64`.

This function does the following:

1. Calls `Ivi_GetAttrRangeTable` to obtain the range table for the attribute. If the range table is invalid, the function returns an error.
2. If there is no range table or its type is not `IVI_VAL_COERCED`, the function sets the **coercedValue** parameter to the value you specify for **attributeValue**.
3. Calls `Ivi_GetViReal64EntryFromValue` to find an entry that matches the value.
4. If it can find an entry, it sets the **coercedValue** parameter to the value of the `coercedValue` field in the range table entry and returns `VI_SUCCESS`. Otherwise, it returns an error.

Note ***Do not call this function directly unless you are calling it from your own coerce callback or you have already called*** `Ivi_LockSession`**.**

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
channel	`ViConstString`	If the attribute is channel-based, pass a channel string or virtual channel name. Otherwise, pass `VI_NULL` or an empty string.
attributeID	`ViAttr`	The ID of the attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.
attributeValue	`ViReal64`	The value you want to coerce.

Output

Name	Type	Description
coercedValue	`ViInt32`	Returns the value to which the function coerces the input value based on the range table for the attribute.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_AddAttributeViReal64`, `Ivi_SetAttrCoerceCallbackViReal64`, `Ivi_SetAttrRangeTableCallback`, `Ivi_GetAttrRangeTable`, `Ivi_GetStoredRangeTablePtr`, `Ivi_SetStoredRangeTablePtr`, `Ivi_GetViReal64EntryFromValue`

Ivi_DefaultCompareCallbackViReal64

```
ViStatus status = Ivi_DefaultCompareCallbackViReal64 (ViSession vi,
                           ViConstString channel, ViAttr attributeID,
                           ViReal64 a, ViReal64 b, ViInt32 *result);
```

Purpose

Performs the default compare actions for a `ViReal64` attribute. The IVI engine invokes the compare callback to compare cache values it obtains from the instrument against new values you set the attribute to. If the compare callback determines that the two values are equal, the IVI engine does not call the write callback for the attribute.

The IVI Library automatically installs this callback when you call `Ivi_AddAttributeViReal64`. The IVI engine installs the default compare callback rather than comparing based on strict equality because of differences between computer and instrument floating-point representations.

You can install your own compare callback by calling `Ivi_SetAttrCompareCallbackViReal64`.

If the two values are not exactly equal, the function uses the following logic, where **a** and **b** are the values you want to compare, and *d* is the digits of precision you specify when you call `Ivi_AddAttributeViReal64` or `Ivi_SetAttrComparePrecision`.

if a == 0

if $|b| < 10^{-(d-1)}$

then a == b.

else /* a != 0 */

if $\frac{|a-b|}{|a|} < 10^{-(d-1)}$

then a == b

Note ***Do not call this function compare directly unless you are calling it from your own callback or you have already called*** `Ivi_LockSession`**.**

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
channel	`ViConstString`	If the attribute is channel-based, pass a channel string or virtual channel name. Otherwise, pass `VI_NULL` or an empty string.
attributeID	`ViAttr`	The ID of the attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.
a	`ViReal64`	The first value to compare. Normally, the new value to which you are trying to set the attribute.
b	`ViReal64`	The second value to compare. Normally, the current cache value of the attribute.

Output

Name	Type	Description
result	`viInt32`	Returns the result of the comparison. 0 if `a` == `b` –1 if `a` < `b` 1 if `a` > `b`

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

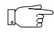
`Ivi_AddAttributeViReal64`, `Ivi_SetAttrCompareCallbackViReal64`, `Ivi_SetAttrRangeTableCallback`, `Ivi_GetAttrRangeTable`, `Ivi_GetStoredRangeTablePtr`, `Ivi_SetStoredRangeTablePtr`, `Ivi_SetAttrComparePrecision`, `Ivi_GetAttrComparePrecision`

Ivi_DefineClass

```
ViStatus status = Ivi_DefineClass (ViConstString className,
                         ViConstString simulationVInstrName);
```

Purpose

Defines a Class entry in the list of run-time configuration entries.



Note ***The*** `ivi.ini` ***configuration file contains the static configuration entries. Refer to*** *Configuration Entries* ***in*** Chapter 2, *IVI Architecture Overview*, ***for more information on static and run-time configuration entries.***

A Class entry specifies the `VInstr` entry of the default simulation driver for the class. If you open an IVI session though a class driver and you enable simulation, the class driver first tries to find the simulation driver through the `VInstr` entry for the specific instrument. If the `VInstr` entry for the specific instrument does not specify a simulation driver, the class driver uses the default simulation driver you specify in the Class entry. If the Class entry does not exist or it does not specify a simulation driver, the class driver uses a hard coded `VInstr` entry name for the default simulation driver.

After you define the Class entry, you cannot modify it. If you want to redefine the Class entry, you must first call `Ivi_UndefClass` and then call `Ivi_DefineClass` again.

You can specify a name for the Class entry that conflicts with a Class entry name in the `ivi.ini` configuration file. The Class entry you create with `Ivi_DefineClass` overrides the one in the configuration file.

Parameters

Input

Name	Type	Description
className	`ViConstString`	Specify the name of a `VInstr` entry that defines a simulation driver. Can be `VI_NULL`. Refer to the *Parameter Discussion* section.
simulationVInstr Name	`ViConstString`	The name of the `VInstr` entry for the default simulation driver for the class. Refer to the *Parameter Discussion* section.

Return Value

Name	Type	Description
status	ViStatus	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

The names of all Class configuration entries begin with "Class->". If **className** does not have the prefix, the function inserts it for you when it creates the Class entry. For example, to set the name to "Class->IviDmm", you can pass "IviDmm" or "Class->IviDmm".

You do not have to include the "VInstr->" prefix in the **simulationVInstrName** parameter. You can specify a VInstr entry that is in either the static or the run-time configuration.

See Also

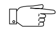
Ivi_UndefClass, Ivi_DefineVInstr, Ivi_WriteRunTimeDefinesToFile

Ivi_DefineDriver

```
ViStatus status = Ivi_DefineDriver (ViConstString driverName,
                                    ViConstString specificDriverPrefix,
                                    ViConstString modulePath);
```

Purpose

Defines a Driver entry in the list of run-time configuration entries.



Note ***The*** `ivi.ini` ***configuration file contains the static configuration entries. Refer to*** *Configuration Entries* **in** Chapter 2, *IVI Architecture Overview*, ***for more information on static and run-time configuration entries.***

A Driver entry specifies a specific instrument driver software module. Each `VInstr` configuration entry refers to a Driver entry.

After you define the Driver entry, you cannot modify it. If you want to redefine the Driver entry, you must first call `Ivi_UndefDriver` and then call `Ivi_DefineDriver` again.

You can specify a name for the Driver entry that conflicts with a Driver entry name in the `ivi.ini` configuration file. The Driver entry you create with this function overrides the one in the configuration file.

Parameters

Input

Name	Type	Description
driverName	`ViConstString`	The name you want to use to identify this new Driver entry. Refer to the *Parameter Discussion* section.
specificDriverPrefix	`ViConstString`	The specific instrument driver prefix. Refer to the *Parameter Discussion* section.
modulePath	`ViConstString`	The pathname of the software driver module. Refer to the *Parameter Discussion* section.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

The names of all Driver entries begin with `"Driver->"`. If **driverName** does not have the prefix, the function inserts it for you when it creates the Driver entry. For example, to set the name to `"Driver->FL45"`, you can pass `"FL45"` or `"Driver->FL45"`.

The names of all user-callable functions in the specific instrument driver begin with **specificDriverPrefix**. For example, if `"FL45_init"` is a user-callable function in the Fluke 45 driver, then `"FL45"` is the **specificDriverPrefix** for that driver. You must specify the correct case for each letter in the prefix. The prefix can contain a maximum of eight characters, and it cannot contain an underscore.

If you enter a literal string for the **modulePath** parameter under Windows, be sure to use double backslashes to represent one backslash in the pathname.

Module Types

The types of software modules that you can specify in **modulePath** varies based on the operating system and whether you are running an executable or in the LabWindows/CVI development environment.

Under Windows 95/NT, you can specify the following types of software modules:

- 32-bit Windows DLLs.
- Object (`.obj`) files, static library (`.lib`) files, or DLL import library (`.lib`) files that are in a format that LabWindows/CVI can load. To use these types of modules, you must run in the LabWindows/CVI development environment or you must install the LabWindows/CVI Run-time Engine on your computer.
- A source (`.c`) file in a LabWindows/CVI project, if you are running in the LabWindows/CVI development environment.

Under Windows 3.1, you can specify the following types of modules:

- Object (`.obj`) files that you create in LabWindows/CVI.
- Object (`.obj`) or library (`.lib`) files you create with the Watcom 32-bit compiler for Windows 3.1.
- A source (`.c`) file in a LabWindows/CVI project, if you are running in the LabWindows/CVI development environment.

Under Unix, you can specify the following types of modules:

- Shared libraries (`.so`, `.sl`), if you are running a standalone executable.
- Object (`.o`) files or static library (`.a`) files. To use these types of modules, you must run in the LabWindows/CVI development environment or you must install the LabWindows/CVI Run-time Engine on your computer.
- A source (`.c`) file in a LabWindows/CVI project, if you are running in the LabWindows/CVI development environment.

The following table summarizes this information, where CVIRTE stands for the LabWindows/CVI Run-time Engine.

Table 11-2. Module Types on Different Platforms

Platform	CVI Environment	Executable Using CVIRTE	Executable (no CVIRTE)
Win 95/NT			
`.dll`	Yes	Yes	Yes
`.obj/.lib`	Yes	Yes	
`.c`	Yes		
Win 3.1			
`.obj/.lib`	Yes	Yes	
`.c`	Yes		
Unix			
`.so/.sl`		Yes	Yes
`.o/.a`	Yes	Yes	
`.c`	Yes		

See Also

`Ivi_UndefDriver`, `Ivi_DefineVInstr`, `Ivi_WriteRunTimeDefinesToFile`

Ivi_DefineHardware

```
ViStatus status = Ivi_DefineHardware (ViConstString hardwareName,
                                      ViConstString resourceDescriptor);
```

Purpose

Defines a Hardware entry in the list of run-time configuration entries.

Note ***The*** `ivi.ini` ***configuration file contains the static configuration entries. Refer to*** *Configuration Entries* **in** Chapter 2, *IVI Architecture Overview*, ***for more information on static and run-time configuration entries.***

A Hardware entry specifies a physical device. Each `VInstr` entry can refer to a Hardware entry.

After you define the Hardware entry, you cannot modify it. If you want to redefine the Hardware entry, you must first call `Ivi_UndefHardware` and then call `Ivi_DefineHardware` again.

You can specify a name for the Hardware entry that conflicts with a Hardware entry name in the `ivi.ini` configuration file. The Hardware entry you create with this function overrides the one in the configuration file.

Parameters

Input

Name	Type	Description
hardwareName	`ViConstString`	The name you want to use to identify the new Hardware entry. Refer to the *Parameter Discussion* section.
resourceDescriptor	`ViConstString`	Identifies the physical device. If you use VISA I/O to communicate with the device, specify a VISA resource descriptor. For example, `"GPIB::7::INSTR"`.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

The names of all Hardware entries begin with `"Hardware->"`. If **hardwareName** does not have the prefix, the function inserts it for you when it creates the Hardware entry. For example, to set the name to `"Hardware->FL45"`, you can pass `"FL45"` or `"Hardware->FL45"`.

See Also

`Ivi_UndefHardware`, `Ivi_DefineVInstr`, `Ivi_WriteRunTimeDefinesToFile`

Ivi_DefineLogicalName

```
ViStatus status = Ivi_DefineLogicalName (ViConstString logicalName,
                           ViConstString vInstrName);
```

Purpose

Defines a logical name in the list of run-time configuration entries.

Note ***The*** `ivi.ini` ***configuration file contains the static configuration entries. Refer to*** *Configuration Entries* ***in*** Chapter 2, *IVI Architecture Overview*, ***for more information on static and run-time configuration entries.***

A logical name refers to a `VInstr`, which in turn specifies a physical device and a specific driver module. You pass logical names to a class driver initialization function to identify the device and driver you want to use in an IVI session.

After you define the logical name, you cannot modify it. If you want to redefine the logical name, you must first call `Ivi_UndefLogicalName` and then call `Ivi_DefineLogicalName` again.

You can specify a name that conflicts with a logical name in the `ivi.ini` configuration file. The logical name you create with this function overrides the one in the configuration file.

Parameters

Input

Name	Type	Description
logicalName	`ViConstString`	The logical name you want to define.
vInstrName	`ViConstString`	The name of a `VInstr` entry. Refer to the *Parameter Discussion* section.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

You do not have to include the `"VInstr->"` prefix in the **vInstrName** parameter. You can specify a `VInstr` entry that is in either the static or the run-time configuration.

See Also

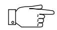
`Ivi_UndefLogicalName`, `Ivi_DefineVInstr`, `Ivi_WriteRunTimeDefinesToFile`, `Ivi_GetLogicalNamesList`, `Ivi_GetNthLogicalName`

Ivi_DefineVInstr

```
ViStatus Ivi_DefineVInstr (ViConstString vInstrName,
                           ViConstString driverName,
                           ViConstString hardwareName,
                           ViConstString virtualChannelNames,
                           ViConstString optionsString,
                           ViConstString simulationVInstrName);
```

Purpose

This function defines a `VInstr` entry in the list of run-time configuration entries.



Note ***The*** `ivi.ini` ***configuration file contains the static configuration entries. Refer to*** *Configuration Entries* ***in*** Chapter 2, *IVI Architecture Overview*, ***for more information on static and run-time configuration entries.***

A `VInstr` entry specifies a virtual instrument consisting of a physical device and a software driver module. You can pass a `VInstr` name to a class driver initialization function to identify the device and driver you want to use in a session, or you can define a logical name to refer to the `VInstr` entry and then pass the logical name to the class driver initialization function.

After you define the `VInstr` entry, you cannot modify it. If you want to redefine the `VInstr` entry, you must first call `Ivi_UndefVInstr` and then call `Ivi_DefineVInstr` again.

You can specify a name for the `VInstr` entry that conflicts with a `VInstr` entry name in the `ivi.ini` configuration file. The `VInstr` entry you create with this function overrides the one in the configuration file.

Parameters

Input

Name	Type	Description
vInstrName	`ViConstString`	The name you want to use to identify this new `VInstr` entry. Refer to the *Parameter Discussion* section.
driverName	`ViConstString`	The name of a Driver entry. A Driver entry specifies a software driver module. Refer to the *Parameter Discussion* section.
hardwareName	`ViConstString`	The name of a Hardware entry. The Hardware entry specifies a physical device. Can be `VI_NULL`. Refer to the *Parameter Discussion* section.
virtualChannel Names	`ViConstString`	Specify virtual channel names you want to use in place of specific driver channel strings in your program. Can be `VI_NULL`. Refer to the *Parameter Discussion* section.
optionsString	`ViConstString`	A string in which you can initialize the values of certain IVI attributes for sessions you create using this `VInstr` entry. Can be `VI_NULL`. Refer to the *Parameter Discussion* section.
simulationVInstr Name	`ViConstString`	The name of the `VInstr` entry for the simulation driver to use for the virtual instrument. Can be `VI_NULL`. Refer to the *Parameter Discussion* section.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

The names of all `VInstr` entries begin with `"VInstr->"`. If **vInstrName** does not contain the prefix, the function inserts it for you when it creates the `VInstr` entry. For example, to set the name to `"VInstr->FL45"`, you can pass `"FL45"` or `"VInstr->FL45"`.

You do not have to include the `"Driver->"` prefix in the **driverName** parameter. You can specify a Driver entry that is in either the static or the run-time configuration.

You do not have to include the `"Hardware->"` prefix in the **hardwareName** parameter. You can specify a Hardware entry that is in either the static or the run-time configuration. You can also pass `VI_NULL` or the empty string for **hardwareName**. This is useful when you want to define a simulation driver or a software-only virtual instrument.

Virtual Channel Names

If you use virtual channel names to refer to channels in your program rather than specific driver channel strings, **virtualChannelStrings** must contain one or more channel name assignments. Each assignment assigns a specific driver channel string to a virtual channel name. You can assign a specific driver channel string to multiple virtual channel names, but you cannot assign multiple specific driver channel strings to the same virtual channel name.

The format of each assignment is

`"`*`VirtualChannelName=SpecificDriverChannelString`*`"`.

To set multiple channels, separate the assignments with commas. You can include spaces after the commas and around the equals sign. The following example creates two virtual channel names, `"ch1"` and `"ch2"`.

```
"ch1 = 1, ch2 = 2"
```

The IVI engine processes all channel names in a case-insensitive manner.

Simulation VInstr

The **simulationVInstr** specifies the simulation driver that the class driver uses when you enable simulation. If you pass `VI_NULL` or an empty string for this parameter, the class driver session uses the simulation driver `VInstr` entry that the Class entry specifies. If the Class entry does not exist or does not specify a simulation driver, the class driver uses a hardcoded `VInstr` entry name for the simulation driver.

You do not have to include the `"VInstr->"` prefix in **simulationVInstrName**. You can specify a `VInstr` entry that is in either the static or the run-time configuration.

Options String

You can use the **optionsString** to set the initial value of certain inherent IVI attributes for sessions that you create. The following table lists the attributes, their default initial values, and the name you use in **optionsString** to identify the attribute.

Table 11-3. optionsString Values

Name	Attribute Defined Constant	Default
RangeCheck	`IVI_ATTR_RANGE_CHECK`	`VI_TRUE`
QueryInstrStatus	`IVI_ATTR_QUERY_INSTR_STATUS`	`VI_TRUE`
Cache	`IVI_ATTR_CACHE`	`VI_TRUE`
Simulate	`IVI_ATTR_SIMULATE`	`VI_FALSE`
RecordCoercions	`IVI_ATTR_RECORD_COERCIONS`	`VI_FALSE`
DriverSetup	`IVI_ATTR_DRIVER_SETUP`	`""`
InterchangeCheck	`IVI_ATTR_INTERCHANGE_CHECK`	`VI_TRUE`
Spy	`IVI_ATTR_SPY`	`VI_FALSE`
UseSpecificSimulation	`IVI_ATTR_USE_SPECIFIC_SIMULATION`	`VI_FALSE`

If you pass NULL or an empty string for **optionsString**, the session uses the default values. You can override the default values by assigning a value explicitly in **optionsString**.

The format of an assignment is, `"`*`Name=Value`*`"`, where *`Name`* is the first column in the table above and *`Value`* is any one of the following:

- To set the attribute to `VI_TRUE`, use `VI_TRUE`, True, or 1.
- To set the attribute to `VI_FALSE`, use `VI_FALSE`, False, or 0.

The function interprets the name and value in a case-insensitive manner.

To set multiple attributes, separate the assignments with commas.

You do not have to specify all of the attributes. If you do not specify one of the attributes, the session uses the default value.

See Also

`Ivi_UndefVInstr`, `Ivi_DefineLogicalName`, `Ivi_DefineDriver`, `Ivi_DefineClass`, `Ivi_WriteRunTimeDefinesToFile`, `Ivi_GetNextCoercionInfo`

Ivi_DeleteAttribute

```
ViStatus status = Ivi_DeleteAttribute (ViSession vi, ViAttr attributeID);
```

Purpose

Deletes an attribute. Typically, it is not necessary for you to call this function. `Ivi_Dispose` deletes all of the attributes for a session.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
attributeID	`ViAttr`	The ID of the attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_Dispose`

Ivi_DeleteAttributeInvalidation

```
ViStatus status = Ivi_DeleteAttributeInvalidation (ViSession vi,
                      ViAttr attributeID,
                      ViAttr dependentAttributeID);
```

Purpose

Removes the invalidation dependency relationship between two attributes. You establish invalidation dependency relationships using `Ivi_AddAttributeInvalidation`.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
attributeID	`ViAttr`	The ID of the attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.
dependentAttributeID	`ViAttr`	The ID of the attribute which the IVI engine invalidates when the value of the other attribute changes.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_AddAttributeInvalidation`

Ivi_DequeueInstrSpecificError

```
ViStatus status = Ivi_DequeueInstrSpecificError (ViSession vi,
                        ViInt32 *instrumentError,
                        ViChar errorMessage[]);
```

Purpose

Retrieves the error code and description string from the oldest entry in the instrument-specific error queue for an IVI session. It also removes the entry from the queue.

Use the instrument-specific error queue if querying the instrument for its status causes the instrument to lose the error value. In your check status callback, call `Ivi_QueueInstrSpecificError` to insert the instrument error code in the queue, and then return the `IVI_ERROR_INSTR_SPECIFIC` error code from the callback. In your *`Prefix`*`_error_query` function, call `Ivi_InstrSpecificErrorQueueSize` to determine if there is an error in the queue. If not, invoke the check status callback directly. In either case, if there is an error, call `Ivi_DequeueInstrSpecificError` to retrieve the error.

Refer to the *Instruments without Error Queues* discussion in the *Check Status Callback* section of Chapter 2, *IVI Architecture Overview*, for more information.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.

Output

Name	Type	Description
instrumentError	`ViInt32`	Returns the error code from the oldest entry in the queue. You can pass `VI_NULL`.
errorMessage	`ViChar` array	Returns the error message from the oldest entry in the queue. Refer to the *Parameter Discussion* section.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

If you are not interested in the error message, value, pass `VI_NULL` for the **errorMessage** parameter. Otherwise, pass a `ViChar` array that contains at least `IVI_MAX_MESSAGE_BUF_SIZE` (256) bytes.

See Also

`Ivi_QueueInstrSpecificError`, `Ivi_InstrSpecificErrorQueueSize`, `Ivi_ClearInstrSpecificErrorQueue`

Ivi_Dispose

```
ViStatus status = Ivi_Dispose (ViSession vi);
```

Purpose

Destroys the IVI session, all of its attributes, and the memory resources it uses.

This function does *not* close the instrument I/O session. You must close it yourself before calling this function.

You must unlock the session before calling `Ivi_Dispose`.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_SpecificDriverNew`, `Ivi_Alloc`, `Ivi_RangeTableNew`, `Ivi_FreeAll`

Ivi_DisposeInvalidationList

```
void Ivi_DisposeInvalidationList (IviInvalEntry *invalidationList);
```

Purpose

Deallocates an invalidation list you obtain from `Ivi_GetInvalidationList`.

Parameters

Input

Name	Type	Description
invalidationList	`IviInvalEntry *`	The pointer to an invalidation list you obtain from `Ivi_GetInvalidationList`.

Return Value

None

See Also

`Ivi_GetInvalidationList`

Ivi_DisposeLogicalNamesList

```
void Ivi_DisposeLogicalNamesList (IviLogicalNameEntry *logicalNamesList);
```

Purpose

Deallocates a logical names list you obtain from `Ivi_GetLogicalNamesList`.

Parameters

Input

Name	Type	Description
logicalNamesList	`IviLogicalNameEntry *`	The pointer to a logical names list you obtain from `Ivi_GetLogicalNamesList`.

Return Value

None

See Also

`Ivi_GetLogicalNamesList`

Ivi_Free

```
ViStatus status = Ivi_Free (ViSession vi, ViAddr memoryBlockPointer);
```

Purpose

Deallocates a memory block you allocate with `Ivi_Alloc`. If you specify a non-NULL IVI session handle, the function also removes the memory block from the list of memory blocks that the IVI engine maintains for the session.

Caution ***The* vi *parameter must be the same IVI session handle that you pass to* `Ivi_Alloc` *when you allocate the memory block. If you pass* `VI_NULL` *for the* vi *parameter to* `Ivi_Alloc`*, pass* `VI_NULL` *for the* vi *parameter to this function.***

Parameters

Input

Name	Type	Description
vi	`ViSession`	The same IVI Session handle that you pass to `Ivi_Alloc` when you allocate the memory block.
memoryBlockPointer	`ViAddr`	A pointer to the memory block you want to deallocate. If `Ivi_Alloc` did not allocate the memory block, `Ivi_Free` returns an error.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_Alloc`, `Ivi_FreeAll`, `Ivi_Dispose`

Ivi_FreeAll

```
ViStatus status = Ivi_FreeAll (ViSession vi);
```

Purpose

Deallocates all memory blocks you allocate with `Ivi_Alloc` or `Ivi_RangeTableNew` for the session.

When you call `Ivi_Dispose` on a session, it calls `Ivi_FreeAll` for you.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_Alloc`, `Ivi_Free`, `Ivi_Dispose`, `Ivi_RangeTableNew`, `Ivi_SetRangeTableEntry`, `Ivi_SetRangeTableEnd`, `Ivi_RangeTableFree`

Ivi_GetAttrComparePrecision

```
ViStatus status = Ivi_GetAttrComparePrecision (ViSession vi,
                                  ViAttr attributeID, ViInt32 *comparePrecision);
```

Purpose

Returns the degree of decimal precision the default IVI compare callback currently uses for this attribute. For more information refer to the *Comparison Precision* section in Chapter 2, *IVI Architecture Overview*.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
attributeID	`ViAttr`	The ID of the attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.

Output

Name	Type	Description
comparePrecision	`ViInt32`	The degree of precision the default IVI compare callback currently uses for this attribute. The value is in terms of decimal digits of precision.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_SetAttrComparePrecision`, `Ivi_AddAttributeViReal64`, `Ivi_DefaultCompareCallbackViReal64`

Ivi_GetAttributeFlags

```
ViStatus status = Ivi_GetAttributeFlags (ViSession vi, ViAttr attributeID,
                                         IviAttrFlags *flags);
```

Purpose

Obtains the current values of the flags for an attribute. Refer to the *Attribute Flags* section in Chapter 2, *IVI Architecture Overview*, for more information.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
attributeID	`ViAttr`	The ID of the attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.

Output

Name	Type	Description
flags	`IviAttrFlags`	Returns the current values of the flags of the attribute. Refer to the *Attribute Flags* section in Chapter 2, *IVI Architecture Overview*.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_SetAttributeFlags`, `Ivi_GetNumAttributes`, `Ivi_GetNthAttribute`, `Ivi_GetAttributeName`, `Ivi_GetAttributeType`

Ivi_GetAttributeName

```
ViStatus status = Ivi_GetAttributeName (ViSession vi, ViAttr attributeID,
                                        ViChar nameBuffer[], ViInt32 bufferSize);
```

Purpose

Obtains the name of an attribute.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
attributeID	`ViAttr`	The ID of the attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.
bufferSize	`ViInt32`	The number of bytes in the `ViChar` array you pass for **nameBuffer**.

Output

Name	Type	Description
nameBuffer	`ViChar` array	A buffer into which the function copies the name of the attribute.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

If the attribute name, including the terminating NUL byte, is longer than the number of bytes you specify in **bufferSize**, the function copies (**bufferSize** – 1) bytes into the buffer, and places an ASCII NUL byte at the end of the buffer.

See Also

`Ivi_GetNumAttributes`, `Ivi_GetNthAttribute`, `Ivi_GetAttributeFlags`, `Ivi_GetAttributeType`

Ivi_GetAttributeType

```
ViStatus status = Ivi_GetAttributeType (ViSession vi, ViAttr attributeID,
                            IviValueType *dataType);
```

Purpose

Obtains the data type of an attribute.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
attributeID	`ViAttr`	The ID of the attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.

Output

Name	Type	Description
dataType	`IviValueType`	Returns the data type of the attribute. Values: 1 = `IVI_VAL_INT32` 4 = `IVI_VAL_REAL64` 5 = `IVI_VAL_STRING` 10 = `IVI_VAL_ADDR` 11 = `IVI_VAL_SESSION` 13 = `IVI_VAL_BOOLEAN`

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_GetNumAttributes`, `Ivi_GetNthAttribute`, `Ivi_GetAttributeName`, `Ivi_GetAttributeFlags`

Ivi_GetAttributeViInt32
Ivi_GetAttributeViReal64
Ivi_GetAttributeViBoolean
Ivi_GetAttributeViSession
Ivi_GetAttributeViAddr

```
ViStatus status = Ivi_GetAttributeViInt32 (ViSession vi,
                           ViConstString channel, ViAttr attributeID,
                           ViInt32 optionFlags, ViInt32 *attributeValue);
ViStatus status = Ivi_GetAttributeViReal64 (ViSession vi,
                           ViConstString channel, ViAttr attributeID,
                           ViInt32 optionFlags, ViReal64 *attributeValue);
ViStatus status = Ivi_GetAttributeViBoolean (ViSession vi,
                           ViConstString channel, ViAttr attributeID,
                           ViInt32 optionFlags, ViBoolean *attributeValue);
ViStatus status = Ivi_GetAttributeViSession (ViSession vi,
                           ViConstString channel, ViAttr attributeID,
                           ViInt32 optionFlags, ViSession *attributeValue);
ViStatus status = Ivi_GetAttributeViAddr (ViSession vi,
                           ViConstString channel, ViAttr attributeID,
                           ViInt32 optionFlags, ViAddr *attributeValue);
```

Purpose

Obtains the current value of an attribute. A separate typesafe function exists for each possible attribute data type.

Note ***A separate function description exists for*** `Ivi_GetAttributeViString`**.**

Depending on the configuration of the attribute, each function performs the following actions:

1. Checks whether the attribute is readable. If not, the function returns an error.
2. If a deferred update is pending for the attribute, the `IVI_ATTR_RETURN_DEFERRED_VALUES` attribute is enabled, and the `IVI_ATTR_DONT_RETURN_DEFERRED_VALUE` flag for the attribute is 0, the function returns the deferred update value and skips the remainder of these actions.
3. If the attribute cache value is currently valid, the read callback for the attribute is `VI_NULL`, or the `IVI_ATTR_SIMULATE` attribute is enabled and the `IVI_ATTR_USE_CALLBACKS_FOR_SIMULATION` flag for the attribute is 0, the function returns the cache value and skips the remainder of these actions.
4. If the `IVI_VAL_WAIT_FOR_OPC_BEFORE_READS` flag is set for the attribute, the function invokes the operation complete (OPC) callback you provide for the session.

5. The function invokes the read callback for the attribute. Typically, the callback performs instrument I/O to obtain a new value. The IVI engine stores the new value in the cache.
6. If you set the IVI_VAL_DIRECT_USER_CALL bit in the **optionFlags** parameter, the IVI_ATTR_QUERY_INSTR_STATUS attribute is enabled, and the IVI_VAL_DONT_CHECK_STATUS flag for the attribute is 0, the function invokes the check status callback you provide for the session.

Parameters

Input

Name	Type	Description
vi	ViSession	The ViSession handle that you obtain from Ivi_SpecificDriverNew. The handle identifies a particular IVI session.
channel	ViConstString	If the attribute is channel-based, pass a channel string or virtual channel name. Otherwise, pass VI_NULL or an empty string.
attributeID	ViAttr	The ID of the attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.
optionFlags	ViInt32	Use this parameter to request special behavior. In most cases, you pass 0. Refer to the *Parameter Discussion* section.

Output

Name	Type	Description
attributeValue	Depends on the data type	Returns the current value of the attribute. Specify the address of a variable that has the same data type as the attribute.

Return Value

Name	Type	Description
status	ViStatus	A negative value indicates an error. A positive value indicates a warning. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

Option Flags

Normally, you pass 0 for **optionFlags**. Pass IVI_VAL_DIRECT_USER_CALL only when calling this function to implement one of the *Prefix*_GetAttribute functions that your instrument driver exports. When you pass IVI_VAL_DIRECT_USER_CALL, the function returns an error if the IVI_VAL_NOT_READABLE or IVI_VAL_NOT_USER_READABLE flag for the attribute is set. Also, the function invokes the check status callback when you pass IVI_VAL_DIRECT_USER_CALL but only if IVI_ATTR_QUERY_INSTR_STATUS is enabled for the session and the IVI_VAL_DONT_CHECK_STATUS flag for the attribute is 0.

Ivi_GetAttributeViString

```
ViStatus statusOrSize = Ivi_GetAttributeViString (ViSession vi,
                          ViConstString channel, ViAttr attributeID,
                          ViInt32 optionFlags, ViInt32 bufferSize,
                          ViChar attributeValue[]);
```

Purpose

Obtains the current value of a `ViString` attribute.

You must pass a `ViChar` array as the **attributeValue** parameter. The array serves as a buffer that receives the value. You pass the number of bytes in the buffer as the **bufferSize** parameter. If the current value of the attribute, including the terminating NUL byte, is larger than the size you indicate in **bufferSize**, the function copies (**bufferSize** – 1) bytes into the buffer, places an ASCII NUL byte at the end of the buffer, and returns the buffer size you must pass to get the entire value. For example, if the value is `"123456"` and **bufferSize** is 4, the function places `"123"` into the buffer and returns 7.

If you want the function to fill in the buffer regardless of the number of bytes in the value, pass a negative number for the **bufferSize** parameter. If you want to call this function just to get the required buffer size, you can pass 0 for the **bufferSize** and `VI_NULL` for the **attributeValue** buffer.

Remember that the `checkErr` and `viCheckErr` macros ignore positive return values. If you use one of these macros around a call to this function, you lose the required buffer size when the function returns it. To retain this information, declare a separate local variable to store the required buffer size, and use the macro around the assignment of the return value to the local variable. The following is an example:

```
ViStatus error = VI_SUCCESS;
ViInt32 requiredBufferSize;

checkErr( requiredBufferSize = Ivi_GetAttributeViString(vi, channel,
                               attr, 0, 0, VI_NULL));
```

Refer to the *Error Macros* section earlier in this chapter for more information on the `checkErr` and `viCheckErr` macros.

Depending on the configuration of the attribute, each function performs the following actions:

1. Checks whether the attribute is readable. If not, the function returns an error.
2. If a deferred update is pending for the attribute, the `IVI_ATTR_RETURN_DEFERRED_VALUES` attribute is enabled, and the `IVI_ATTR_DONT_RETURN_DEFERRED_VALUE` flag for the attribute is 0, the function returns the deferred update value and skips the remainder of these actions.

3. If the attribute cache value is currently valid, the read callback for the attribute is VI_NULL, or the IVI_ATTR_SIMULATE attribute is enabled and the IVI_ATTR_USE_CALLBACKS_FOR_SIMULATION flag for the attribute is 0, the function returns the cache value and skips the remainder of these actions.
4. If the IVI_VAL_WAIT_FOR_OPC_BEFORE_READS flag is set for the attribute, the function invokes the operation complete (OPC) callback you provide for the session.
5. The function invokes the read callback for the attribute. Typically, the callback performs instrument I/O to obtain a new value. The IVI engine stores the new value in the cache.
6. If you set the IVI_VAL_DIRECT_USER_CALL bit in the **optionFlags** parameter, the IVI_ATTR_QUERY_INSTR_STATUS attribute is enabled, and the IVI_VAL_DONT_CHECK_STATUS flag for the attribute is 0, the function invokes the check status callback you provide for the session.

Parameters

Input

Name	Type	Description
vi	ViSession	The ViSession handle that you obtain from Ivi_SpecificDriverNew. The handle identifies a particular IVI session.
channel	ViConstString	If the attribute is channel-based, pass a channel string or virtual channel name. Otherwise, pass VI_NULL or an empty string.
attributeID	ViAttr	The ID of the attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.
optionFlags	ViInt32	Use this parameter to request special behavior. In most cases, you pass 0. Refer to the *Parameter Discussion* section.
bufferSize	ViInt32	The number of bytes in the ViChar array you specify for the **attributeValue** parameter.

Output

Name	Type	Description
attributeValue	`ViChar` array	The buffer in which the function returns the current value of the attribute. Can be `VI_NULL` if **bufferSize** is 0.

Return Value

Name	Type	Description
statusOrSize	`ViStatus`	A negative value indicates an error. A positive value indicates that the buffer you passed is not large enough to hold the current value. The value is the size of the buffer you must pass to obtain the entire value. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

Option Flags

Normally, you pass 0 for **optionFlags**. Pass `IVI_VAL_DIRECT_USER_CALL` only when calling this function to implement one of the *`Prefix`*`_GetAttribute` functions that your instrument driver exports to the user. When you pass `IVI_VAL_DIRECT_USER_CALL`, the function returns an error if the `IVI_VAL_NOT_READABLE` or `IVI_VAL_NOT_USER_READABLE` flag for the attribute is not set. Also, the function invokes the check status callback when you pass `IVI_VAL_DIRECT_USER_CALL` but only if `IVI_ATTR_QUERY_INSTR_STATUS` is enabled for the session and the `IVI_VAL_DONT_CHECK_STATUS` flag for the attribute is 0.

Ivi_GetAttrMinMaxViInt32

```
ViStatus status = Ivi_GetAttrMinMaxViInt32 (ViSession vi,
                         ViConstString channel, ViAttr attributeID,
                         ViInt32 *minimum, ViInt32 *maximum,
                         ViBoolean *hasMin, ViBoolean *hasMax);
```

Purpose

Returns the minimum and maximum values that an instrument implements for a `ViInt32` attribute on a specific channel. The values represent the minimum and maximum values the driver or instrument actually uses rather than the possible values you can pass to `Ivi_SetAttributeViInt32`. In particular, for a coerced range table, the function uses the `coercedValue` field in each entry.

`Ivi_GetAttrMinMaxViInt32` calls `Ivi_GetAttrRangeTable` to obtain the range table for the attribute. If the attribute has no range table or the table is invalid, the function returns an error.

The `hasMin` and `hasMax` fields in the range table indicate whether, as a whole, the table contains a meaningful minimum value and a meaningful maximum value. `Ivi_GetAttrMinMaxViInt32` returns these indicators.

If the `hasMin` field in the table is non-zero, the function searches the table for the minimum value. For discrete and ranged tables, the function examines the `discreteOrMinValue` field in each entry. For coerced tables, the function examines the `coercedValue` field.

If the `hasMax` field in the table is non-zero, the function searches the table for the maximum value. For discrete tables, the function examines the `discreteOrMinValue` field in each entry. For ranged tables, the function examines the `maxValue` field. For coerced tables, the function examines the `coercedValue` field.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.

Name	Type	Description
channel	ViConstString	If the attribute is channel-based, pass a channel string or virtual channel name. Otherwise, pass VI_NULL or an empty string.
attributeID	ViAttr	The ID of the attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.

Output

Name	Type	Description
minimum	ViInt32	The minimum value in the table, if **hasMin** returns VI_TRUE. You can pass VI_NULL.
maximum	ViInt32	The maximum value in the table, if **hasMin** returns VI_TRUE. You can pass VI_NULL.
hasMin	ViBoolean	Returns VI_TRUE if the range table indicates that it has a meaningful minimum value. Otherwise, returns VI_FALSE. You can pass VI_NULL.
hasMax	ViBoolean	Returns VI_TRUE if the range table indicates that it has a meaningful maximum value. Otherwise, returns VI_FALSE. You can pass VI_NULL.

Return Value

Name	Type	Description
status	ViStatus	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

Ivi_GetAttrRangeTable, Ivi_ValidateRangeTable, Ivi_GetRangeTableNumEntries, Ivi_GetViInt32EntryFromValue, Ivi_GetAttrMinMaxViReal64

Ivi_GetAttrMinMaxViReal64

```
ViStatus status = Ivi_GetAttrMinMaxViReal64 (ViSession vi,
                           ViConstString channel, ViAttr attributeID,
                           ViReal64 *minimum, ViReal64 *maximum,
                           ViBoolean *hasMin, ViBoolean *hasMax);
```

Purpose

Returns the minimum and maximum values that an instrument implements for a `ViReal64` attribute on a specific `channel`. The values represent the minimum and maximum values the driver or instrument actually uses rather than the possible values you can pass to `Ivi_SetAttributeViReal64`. function. In particular, for a coerced range table, the function uses the `coercedValue` fields.

`Ivi_GetAttrMinMaxViReal64` calls `Ivi_GetAttrRangeTable` to obtain the range table for the attribute. If the attribute has no range table or the table is invalid, the function returns an error.

The `hasMin` and `hasMax` fields in the range table indicate whether, as a whole, the table contains a meaningful minimum value and a meaningful maximum value. `Ivi_GetAttrMinMaxViReal64` returns these indicators.

If the `hasMin` field in the table is non-zero, the function searches the table for the minimum value. For discrete and ranged tables, the function examines the `discreteOrMinValue` field in each entry. For coerced tables, the function examines the `coercedValue` field.

If the `hasMax` field in the table is non-zero, the function searches the table for the maximum value. For discrete tables, the function examines the `discreteOrMinValue` field in each entry. For ranged tables, the function examines the `maxValue` field. For coerced tables, the function examines the `coercedValue` field.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.

Name	Type	Description
channel	`ViConstString`	If the attribute is channel-based, pass a channel string or virtual channel name. Otherwise, pass `VI_NULL` or an empty string.
attributeID	`ViAttr`	The ID of the attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.

Output

Name	Type	Description
minimum	`ViReal64`	The minimum value in the table, if **hasMin** returns `VI_TRUE`. You can pass `VI_NULL`.
maximum	`ViReal64`	The maximum value in the table, if **hasMin** returns `VI_TRUE`. You can pass `VI_NULL`.
hasMin	`ViBoolean`	Returns `VI_TRUE` if the range table indicates that it has a meaningful minimum value. Otherwise, returns `VI_FALSE`. You can pass `VI_NULL`.
hasMax	`ViBoolean`	Returns `VI_TRUE` if the range table indicates that it has a meaningful maximum value. Otherwise, returns `VI_FALSE`. You can pass `VI_NULL`.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_GetAttrRangeTable`, `Ivi_ValidateRangeTable`, `Ivi_GetRangeTableNumEntries`, `Ivi_GetViReal64EntryFromValue`, `Ivi_GetAttrMinMaxViInt32`

Ivi_GetAttrRangeTable

```
ViStatus status = Ivi_GetAttrRangeTable (ViSession vi,
                      ViConstString channel, ViAttr attributeID,
                      IviRangeTablePtr *rangeTable);
```

Purpose

Returns a pointer to the range table for an attribute. If you call `Ivi_SetAttrRangeTableCallback` to install a range table callback function for the attribute, `Ivi_GetAttrRangeTable` invokes your range table callback with the **vi**, **attributeID**, and **channel** parameters. Otherwise, `Ivi_GetAttrRangeTable` returns the address of the range table you specify for the attribute, when you call `Ivi_AddAttributeViInt32`, `Ivi_AddAttributeViReal64`, or `Ivi_SetStoredRangeTablePtr`.

To bypass the range table callback and always return the range table you store for the attribute, call `Ivi_GetStoredRangeTablePtr`.

If you install your own check callback function, call `Ivi_GetAttrRangeTable` from the check callback to obtain a pointer to the range table.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
channel	`ViConstString`	If the attribute is channel-based, pass a channel string or virtual channel name. Otherwise, pass `VI_NULL` or an empty string.
attributeID	`ViAttr`	The ID of the attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.

Output

Name	Type	Description
rangeTable	`IviRangeTablePtr`	Returns the address of the range table that currently applies to the attribute on the channel you specify.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_SetAttrRangeTableCallback`, `Ivi_SetStoredRangeTablePtr`, `Ivi_GetStoredRangeTablePtr`, `Ivi_RangeTableNew`, `Ivi_ValidateRangeTable`, `Ivi_GetRangeTableNumEntries`, `Ivi_GetViInt32EntryFromValue`, `Ivi_GetViReal64EntryFromValue`, `Ivi_GetViInt32EntryFromIndex`, `Ivi_GetViReal64EntryFromIndex`

Ivi_GetErrorInfo

```
ViStatus status = Ivi_GetErrorInfo (ViSession vi, ViStatus *primaryError,
                           ViStatus *secondaryError,
                           ViChar errorElaboration[]);
```

Purpose

Retrieves and then clears the error information for an IVI session or for the current execution thread. If you specify a valid IVI session for the **vi** parameter, `Ivi_GetErrorInfo` retrieves and then clears the error information for the session. If you pass `VI_NULL` for the **vi** parameter, `Ivi_GetErrorInfo` retrieves and then clears the error information for the current execution thread.

Instrument drivers export `Ivi_GetErrorInfo` to the user through the *`Prefix`*`_GetErrorInfo` function. Normally, the error information describes the first error that occurred since the user last called *`Prefix`*`_GetErrorInfo` or *`Prefix`*`_ClearErrorInfo`.

You can call `Ivi_GetErrorMessage` to obtain text descriptions of the values the function returns in **primaryError** and **secondaryError**.

Refer to the *Error Reporting* section earlier in this chapter for details on IVI error information.

Parameters

Input

Name	Type	Description
vi	`ViSession`	To obtain error information for a particular IVI session, pass the session handle that you obtain from `Ivi_SpecificDriverNew`. To obtain the error information for the current thread, pass `VI_NULL`.

Output

Name	Type	Description
primaryError	`ViStatus`	Returns the primary error code. 0 indicates that no error occurred. A positive value indicates a warning. A negative value indicates an error. You can pass `VI_NULL`.
secondaryError	`ViStatus`	Returns the secondary error code. The secondary error code further describes a primary error or warning condition. 0 indicates no further description. You can pass `VI_NULL`.
errorElaboration	`ViChar` array	Buffer into which the function copies the error elaboration string. The elaboration string further describes a primary error or warning condition. The buffer must contain at least `IVI_MAX_MESSAGE_BUF_SIZE` (256) bytes. You can pass `VI_NULL`.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates that the **vi** parameter is invalid or that the IVI engine was not able to allocate thread-local variables for the error information. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_SetErrorInfo`, `Ivi_ClearErrorInfo`, `Ivi_GetErrorMessage`, `Ivi_GetSpecificDriverStatusDesc`

Ivi_GetErrorMessage

```
ViStatus status = Ivi_GetErrorMessage (ViStatus statusCode,
                            ViChar statusMessage[]);
```

Purpose

Converts an IVI or VISA status code into a meaningful message string. For all other values, it reports the **"Unknown status value"** message and returns the VI_WARN_UNKNOWN_STATUS warning code.

If you have a table of error codes and messages that are specific to the instrument driver, call Ivi_GetSpecificDriverStatusDesc instead of this function.

Parameters

Input

Name	Type	Description
statusCode	ViStatus	An IVI or VISA status code.

Output

Name	Type	Description
statusMessage	ViChar array	Buffer in which the function places a description of **statusCode**. The buffer must contain at least IVI_MAX_MESSAGE_BUF_SIZE (256) bytes. You can pass VI_NULL.

Return Value

Name	Type	Description
status	ViStatus	A negative value indicates an error. A positive value indicates a warning. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

Ivi_SetErrorInfo, Ivi_GetErrorInfo, Ivi_ClearErrorInfo, Ivi_GetSpecificDriverStatusDesc

Ivi_GetInvalidationList

```
ViStatus status = Ivi_GetInvalidationList (ViSession vi, ViAttr attributeID,
                          IviInvalEntry **invalidationList,
                          ViInt32 *numberOfEntries);
```

Purpose

Returns a list of all the invalidation dependency relationships for the session. The specific driver creates the dependency relationships using `Ivi_AddAttributeInvalidation`.

The `ivi.h` include file defines the structure of an entry in the list as follows.

```
typedef struct
   {
   ViAttr      attribute;
   ViBoolean   allChannels;
   } IviInvalEntry;
```

`Ivi_GetInvalidationList` dynamically allocates an array of `IviInvalEntry` structures and returns a pointer to it. The last entry in the array is a termination entry that has `IVI_ATTR_NONE` (-1) in the attribute field. The function also returns the number of items in the array, excluding the termination entry. When you are done with the list, you must free it by calling `Ivi_DisposeInvalidationList`.

You can pass `VI_NULL` for the **invalidationList** parameter, in which case the function just returns the number of dependency relationships.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
attributeID	`ViAttr`	The ID of the attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.

Output

Name	Type	Description
invalidationList	IviInvalEntry	Returns the pointer to an array that contains all the invalidation dependency relationships for the session. You can pass VI_NULL.
numberOfEntries	ViInt32	Returns the number of entries in the invalidation list, excluding the termination entry.

Return Value

Name	Type	Description
status	ViStatus	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

Ivi_DisposeInvalidationList, Ivi_AddAttributeInvalidation, Ivi_DeleteAttributeInvalidation

Ivi_GetIviIniDir

```
ViStatus status = Ivi_GetIviIniDir (ViChar directoryPath[]);
```

Purpose

Returns the pathname of the directory in which the IVI engine looks for the ivi.ini configuration file.

Refer to the Configuration Entries section in Chapter 2, *IVI Architecture Overview*, for information on ivi.ini.

Parameters

Output

Name	Type	Description
directoryPath	ViChar array	Buffer in which the function returns the directory where the IVI engine looks for the ivi.ini configuration file. Must contain at least IVI_MAX_PATHNAME_LEN bytes.

Return Value

Name	Type	Description
status	ViStatus	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

The ivi.h include file defines the value IVI_MAX_PATHNAME_LEN differently for each platform.

See Also

Ivi_GetLogicalNamesList, Ivi_GetNthLogicalName, Ivi_SetIviIniDir

Ivi_GetLogicalNamesList

```
ViStatus status = Ivi_GetLogicalNamesList
                          (IviLogicalNameEntry **logicalNamesList,
                          ViInt32 *numberOfEntries);
```

Purpose

Returns a list of the logical names that the IVI engine currently recognizes. You can define logical names in the IVI configuration file `ivi.ini`. You can also define logical names at run-time using `Ivi_DefineLogicalName`.

You pass logical names to class driver initialization functions to identify the physical device and specific driver module you want to use in a session.

The `ivi.h` include file defines the structure of an entry in the list as follows.

```
typedef struct
   {
   ViString    logicalName;
   ViBoolean   fromFile;
   } IviLogicalNameEntry;
```

The `fromFile` field is `VI_TRUE` if you define the logical name in the `ivi.ini` file rather than by calling `Ivi_DefineLogicalName`.

`Ivi_GetLogicalNamesList` dynamically allocates an array of `IviLogicalNameEntry` structures and returns a pointer to it. The logical names you define at run-time appear before the logical names from the configuration file. The last entry in the array is a termination entry that has `VI_NULL` in the `logicalName` field. The function also returns the number of logical names in the list, excluding the termination entry. When you are done with the list, you must free it by calling `Ivi_DisposeLogicalNamesList`.

Call `Ivi_GetNthLogicalName` to extract the data from an entry in the list. Do not change the values of any of the entries in the list.

You can pass `VI_NULL` for the **logicalNamesList** parameter, in which case the function returns only the number of logical names.

Parameters

Output

Name	Type	Description
logicalNamesList	`IviLogicalNameEntry *`	Returns the pointer to an array of the logical names that the IVI engine currently recognizes.
numberOfEntries	`ViInt32`	Returns the number of entries in the logical names list, excluding the termination entry.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_GetNthLogicalName`, `Ivi_DefineLogicalName`, `Ivi_GetIviIniDir`, `Ivi_SetIviIniDir`

Ivi_GetNextCoercionInfo

```
ViStatus status = Ivi_GetNextCoercionInfo (ViSession vi,
                         ViAttr *attributeID,
                         ViConstString *attributeName,
                         ViConstString *channelString,
                         IviValueType *attributeDataType,
                         ViReal64 *desiredValue, ViReal64 *coercedValue);
```

Purpose

Obtains information regarding the oldest instance in which the IVI engine coerced an attribute value you specified to another value. It then deletes that information.

If you enable the `IVI_ATTR_RECORD_COERCIONS` attribute for the session, the IVI engine keeps a list of all coercions it makes on values you pass to an `Ivi_SetAttribute` function for a `ViInt32` or `ViReal64` attribute. You can use `Ivi_GetNextCoercionInfo` to retrieve information from that list. Each time you call `Ivi_GetNextCoercionInfo`, it extracts and deletes the oldest coercion record for the session.

When no coercion records remain for the session, the function returns `IVI_ATTR_NONE` (-1) in the **attributeID** parameter and `VI_NULL` in the **attributeName** parameter.

`Ivi_GetNextCoercionInfo` returns all numeric values as `ViReal64` values, even for `ViInt32` attributes.

You can pass `VI_NULL` for any of the output parameters, except that you cannot pass `VI_NULL` for both the **attributeID** and **attributeName** parameters in the same call.

Caution *Do not modify the contents of the strings that the* **attributeName** *and* **channelString** *parameters return.*

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.

Output

Name	Type	Description
attributeID	`ViAttr`	Returns the ID of the attribute for which the value coercion occurred. Returns `IVI_ATTR_NONE` (–1) if no more coercion records exist for the session.
attributeName	`ViConstString`	Returns a pointer to the name of the attribute for which the value coercion occurred. Returns `VI_NULL` if no more coercion records exist for session.
channelString	`ViConstString`	If the attribute is channel-based, returns a pointer to the name of the channel on which the value coercion occurred. Otherwise, returns a pointer to an empty string.
attributeDataType	`IviValueType`	Returns the data type of the attribute. 1 = `IVI_VAL_INT32` 4 = `IVI_VAL_REAL64`
desiredValue	`ViReal64`	Returns the value to which you attempted to set the attribute.
coercedValue	`ViReal64`	Returns the value to which the IVI engine actually set the attribute.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_SpecificDriverNew`, `Ivi_DefineVInstr`

Ivi_GetNthAttribute

```
ViStatus status = Ivi_GetNthAttribute (ViSession vi, ViInt32 index,
                        ViAttr *attributeID);
```

Purpose

Obtains the ID of the attribute that is at the index you specify in the IVI session's internal list of attributes. The index is 1-based.

If the index you specify is greater than the number of attributes, `Ivi_GetNthAttribute` sets the **attributeID** parameter to `IVI_ATTR_NONE` (–1) and returns `VI_SUCCESS`.

Call `Ivi_GetNumAttributes` to obtain the number of attributes in the session.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
index	`ViInt32`	A 1-based index into the IVI session's internal list of attributes.

Output

Name	Type	Description
attributeID	`ViAttr`	Returns the ID of the attribute that is in the internal attribute list at the index you specify.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_GetNumAttributes`, `Ivi_GetAttributeType`, `Ivi_GetAttributeName`, `Ivi_GetAttributeFlags`

Ivi_GetNthChannelString

```
ViStatus status = Ivi_GetNthChannelString (ViSession vi, ViInt32 index,
                           ViConstString *channelString);
```

Purpose

Returns the channel string that is in the session's channel table at an index you specify. The specific instrument driver specifies the contents of the channel table using `Ivi_BuildChannelTable` and `Ivi_AddToChannelTable`, and the IVI Library maintains the table for the session.

If the index you specify is greater than the number of channel strings in the table, `Ivi_GetNthChannelString` sets the **channelString** parameter to `VI_NULL` and returns `VI_SUCCESS`.

Caution ***Do not modify the contents of the channel string.***

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
index	`ViInt32`	A 1-based index into the channel table.

Output

Name	Type	Description
channelString	`ViConstString`	Returns the channel string that is in the channel table at the index you specify.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_BuildChannelTable`, `Ivi_AddToChannelTable`, `Ivi_RestrictAttrToChannels`, `Ivi_ValidateAttrForChannel`, `Ivi_CoerceChannelName`, `Ivi_GetUserChannelName`

Ivi_GetNthLogicalName

```
ViStatus status = Ivi_GetNthLogicalName
                        (IviLogicalNameEntry *logicalNamesList,
                        ViInt32 index, ViChar logicalNameBuffer[],
                        ViInt32 bufferSize, ViBoolean *fromFile);
```

Purpose

Extracts the data from an entry in a logical names list you obtain from `Ivi_GetLogicalNamesList`. You specify the entry with a 1-based index.

If the index you specify is greater than the number of logical names, `Ivi_GetNthLogicalName` places an ASCII NUL byte at the beginning of **logicalNameBuffer** and returns `VI_SUCCESS`.

Parameters

Input

Name	Type	Description
logicalNamesList	`IviLogicalNameEntry *`	Specifies the pointer to the logical names list you obtain from `Ivi_GetLogicalNamesList`.
index	`ViInt32`	Specifies the 1-based index of the logical name list entry from which you want to extract data.
bufferSize	`ViInt32`	The number of bytes in the `ViChar` array you pass for the **logicalNameBuffer** parameter.

Output

Name	Type	Description
logicalNameBuffer	`ViChar` array	The buffer into which the function copies the logical name.
fromFile	`ViBoolean`	Returns `VI_TRUE` if you define the logical name in the `ivi.ini` file. Returns `VI_FALSE` if you define the logical name by calling `Ivi_DefineLogicalName`. You can pass `VI_NULL`.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_GetNthLogicalName`, `Ivi_DefineLogicalName`, `Ivi_GetIviIniDir`, `Ivi_SetIviIniDir`

Ivi_GetNumAttributes

```
ViStatus status = Ivi_GetNumAttributes (ViSession vi,
                            ViInt32 *numberOfAttributes);
```

Purpose

Obtains the total number of attributes in an IVI session. This includes all attributes that the IVI engine, the class driver, and the specific driver create, regardless of whether the IVI_VAL_NOT_SUPPORTED flag for the attribute is set.

Parameters

Input

Name	Type	Description
vi	ViSession	The ViSession handle that you obtain from Ivi_SpecificDriverNew. The handle identifies a particular IVI session.

Output

Name	Type	Description
numberOfAttributes	ViInt32	Returns the total number of attributes in the IVI session.

Return Value

Name	Type	Description
status	ViStatus	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

Ivi_GetNthAttribute, Ivi_GetAttributeType, Ivi_GetAttributeName, Ivi_GetAttributeFlags

Ivi_GetRangeTableNumEntries

```
ViStatus status = Ivi_GetRangeTableNumEntries (IviRangeTablePtr rangeTable,
                        ViInt32 *numberofEntries);
```

Purpose

Returns the number of entries in a range table, excluding the termination entry. If you pass `VI_NULL` for the **rangeTable** parameter, `Ivi_GetNumRangeTableEntries` returns 0 as the number of entries.

Parameters

Input

Name	Type	Description
rangeTable	`IviRangeTablePtr`	Specifies the address of the range table you want the function to examine. Can be `VI_NULL`.

Output

Name	Type	Description
numberOfEntries	`ViInt32`	Returns the total number of entries in the range table, excluding the termination entry. If **rangeTable** is `VI_NULL`, this parameter returns 0.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_GetAttrRangeTable`, `Ivi_ValidateRangeTable`,
`Ivi_GetViInt32EntryFromValue`, `Ivi_GetViReal64EntryFromValue`,
`Ivi_GetViInt32EntryFromIndex`, `Ivi_GetViReal64EntryFromIndex`

Ivi_GetSpecificDriverStatusDesc

```
ViStatus status = Ivi_GetSpecificDriverStatusDesc (ViSession vi,
                          ViStatus statusCode, ViChar statusMessage[],
                          IviStringValueTable additionalTableToSearch);
```

Purpose

Converts a status code that an instrument driver function returns into a meaningful message string. It interprets IVI and VISA status codes just as `Ivi_GetErrorMessage` does, but it also allows you to pass a table of error codes and messages that are specific to the instrument driver.

Use `Ivi_GetSpecificDriverStatusDesc` implement the *`Prefix`*`_error_message` function in the instrument driver.

If the function cannot find a description for the status code, it reports the **`"Unknown status value"`** message and returns the `VI_WARN_UNKNOWN_STATUS` warning code.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. You can pass `VI_NULL`.
statusCode	`ViStatus`	A status code that an instrument driver function returns.
additionalTableToSearch	`IviStringValueTable`	The address of a string/value table that contains status codes specific to the instrument driver. Refer to the *Parameter Discussion* section.

Output

Name	Type	Description
statusMessage	`ViChar` array	Buffer in which the function places a description of the value in **statusCode**. The buffer must contain at least `IVI_MAX_MESSAGE_BUF_SIZE` (256) bytes. You can pass `VI_NULL`.

Return Value

Name	Type	Description
status	`ViStatus`	A positive value indicates a warning. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

The **additionalTableToSearch** parameter is the address of a string/value table, which is an array of string/value entries. The `ivi.h` include file defines the structure of a string/value table as follows:

```
typedef struct
    {
    ViInt32      value;
    ViString     string;
    } IviStringValueEntry;
```

Terminate the table with an entry that has `VI_NULL` in the `string` field.

See Also

`Ivi_SetErrorInfo`, `Ivi_GetErrorInfo`, `Ivi_ClearErrorInfo`, `Ivi_GetErrorMessage`

Ivi_GetStoredRangeTablePtr

```
ViStatus status = Ivi_GetStoredRangeTablePtr (ViSession vi,
                           ViAttr attributeID,
                           IviRangeTablePtr *rangeTable);
```

Purpose

Obtains the address of the range table you store for the attribute when you call `Ivi_AddAttributeViInt32`, `Ivi_AddAttributeViReal64`, or `Ivi_SetStoredRangeTablePtr`.

Unlike `Ivi_GetAttrRangeTable`, `Ivi_GetStoredRangeTablePtr` never invokes the range table callback for the attribute.

Refer to the *Range Tables* section in Chapter 2, *IVI Architecture Overview*, for more information on range tables.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
attributeID	`ViAttr`	The ID of the attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.

Output

Name	Type	Description
rangeTable	`IviRangeTablePtr`	Returns the address of the range table that you store for this attribute. If you do not store a range table, this parameter returns `VI_NULL`.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_SetStoredRangeTablePtr`, `Ivi_AddAttributeViInt32`, `Ivi_AddAttributeViReal64`, `Ivi_SetAttrRangeTableCallback`, `Ivi_GetAttrRangeTable`, `Ivi_RangeTableNew`, `Ivi_ValidateRangeTable`, `Ivi_GetRangeTableNumEntries`, `Ivi_GetViInt32EntryFromValue`, `Ivi_GetViReal64EntryFromValue`

Ivi_GetStringFromTable

```
ViStatus status = Ivi_GetStringFromTable (IviStringValueTable stringTable,
                           ViInt32 value, ViString *string);
```

Purpose

Searches for a value in a string/value table and returns the string that corresponds to the value.

If `Ivi_GetStringFromTable` cannot find the value in the table, it returns the `IVI_ERROR_INVALID_VALUE` error code.

The `ivi.h` include file defines the structure of a string/value table entry as follows:

```
typedef struct
   {
   ViInt32     value;
   ViString    string;
   } IviStringValueEntry;
```

Caution ***Do not modify the contents of the string that the function returns.***

Parameters

Input

Name	Type	Description
stringTable	`IviStringValueTable`	The address of the string/value table in which you want to find the value.
value	`ViInt32`	The value you want to find in the string/value table.

Output

Name	Type	Description
string	`ViString`	Returns the address of the string in the first entry that contains the value you specify. Returns `VI_NULL` if no entries contain the value. You can pass `VI_NULL`.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_GetValueFromTable`

Ivi_GetUserChannelName

```
ViStatus status = Ivi_GetUserChannelName (ViSession vi,
                        ViConstString channelString,
                        ViConstString *userChannelName);
```

Purpose

Finds the highest-level channel name that corresponds to the specific driver channel string you specify.

If you specify a channel string that the user assigns to a virtual channel name in the `ivi.ini` configuration file, `Ivi_GetUserChannelName` returns a pointer to the virtual channel name. If the user assigns the channel string to multiple virtual channel names, the function returns a pointer to the first virtual channel name it finds.

If no virtual channel names correspond to the channel string and the channel string is in the channel table that the specific instrument driver defines, the function returns a pointer to the channel string. If the channel string is not in the table, the function sets the **userChannelName** output parameter to `VI_NULL` and returns an error code.

Caution ***Do not modify the contents of the string that the function returns in the*** **userChannelName** ***parameter.***

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
channelString	`ViConstString`	A specific driver channel string. The specific instrument driver specifies the valid channel strings using `Ivi_BuildChannelTable` and `Ivi_AddToChannelTable`.

Output

Name	Type	Description
userChannelName	`ViConstString`	The highest-level channel name that corresponds to the specific driver channel string you pass in the **channelString** parameter.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_BuildChannelTable`, `Ivi_AddToChannelTable`, `Ivi_RestrictAttrToChannels`, `Ivi_ValidateAttrForChannel`, `Ivi_CoerceChannelName`, `Ivi_GetNthChannelString`

Ivi_GetValueFromTable

```
ViStatus status = Ivi_GetValueFromTable (IviStringValueTable stringTable,
                         ViConstString string, ViInt32 *value);
```

Purpose

Searches for a string in a string/value table and returns the value that corresponds to the string.

If the string you specify terminates with a carriage return ('\r') or newline ('\n') character, the strings in the table do not have to contain the termination character. Ivi_GetValueFromTable considers the strings to match if the string you specify begins with the string in the table entry, followed by a carriage return, linefeed, or ASCII NUL byte.

If Ivi_GetValueFromTable cannot find the string in the table, it returns the IVI_ERROR_INVALID_VALUE error.

The ivi.h include file defines the structure of a string/value table entry as follows:

```
typedef struct
   {
   ViInt32     value;
   ViString    string;
   } IviStringValueEntry;
```

Parameters

Input

Name	Type	Description
stringTable	IviStringValueTable	The string/value table in which you want to find the string.
string	ViConstString	The string you want to find in the string/value table.

Output

Name	Type	Description
value	ViInt32	Returns the value in the first entry that contains the string you specify. Returns VI_NULL if no entries contain the string. You can pass VI_NULL.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_GetStringFromTable`

Ivi_GetViInt32EntryFromCmdValue

```
ViStatus status = Ivi_GetViInt32EntryFromCmdValue (ViInt32 commandValue,
                          IviRangeTablePtr rangeTable,
                          ViInt32 *discreteOrMinValue, ViInt32 *maxValue,
                          ViInt32 *coercedValue, ViInt32 *tableIndex,
                          ViString *commandString);
```

Purpose

Finds the first range table entry for which the `cmdValue` field is equal to the command value you specify. If the function finds an entry, it returns the contents of the entry. If it does not find an entry, it returns an `IVI_ERROR_INVALID_VALUE` error.

`Ivi_GetViInt32EntryFromCmdValue` returns the `discreteOrMinValue`, `maxValue`, and `coercedValue` fields as `ViInt32` values.

Caution ***Do not modify the contents of the string the function returns in the*** **commandString** ***parameter.***

Parameters

Input

Name	Type	Description
commandValue	`ViInt32`	The command value to search for in the range table.
rangeTable	`IviRangeTablePtr`	The address of the range table in which to search for the command value.

Output

Name	Type	Description
discreteOrMinValue	`ViInt32`	Returns the contents of the `discreteOrMinValue` field of the table entry the function locates. You can pass `VI_NULL`.
maxValue	`ViInt32`	Returns the contents of the `maxValue` field of the table entry the function locates. You can pass `VI_NULL`.
coercedValue	`ViInt32`	Returns the contents of the `coercedValue` field of the table entry the function locates. You can pass `VI_NULL`.

Name	Type	Description
tableIndex	`ViInt32`	Returns the 0-based index of the table entry the function locates. You can pass `VI_NULL`.
commandString	`ViString`	Returns the pointer in the `cmdString` field of the table entry the function locates. You can pass `VI_NULL`.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_GetAttrRangeTable`, `Ivi_ValidateRangeTable`, `Ivi_GetViInt32EntryFromValue`, `Ivi_GetViReal64EntryFromCmdValue`

Ivi_GetViInt32EntryFromCoercedVal

```
ViStatus status = Ivi_GetViInt32EntryFromCoercedVal (ViInt32 coercedValue,
                         IviRangeTablePtr rangeTable,
                         ViInt32 *discreteOrMinValue, ViInt32 *maxValue,
                         ViInt32 *tableIndex, ViString *commandString,
                         ViInt32 *commandValue);
```

Purpose

Finds the first range table entry for which the `coercedValue` field is equal to the coerced value you specify. If the function finds an entry, it returns the contents of the entry. If it does not find an entry, it returns an `IVI_ERROR_INVALID_VALUE` error.

`Ivi_GetViInt32EntryFromCoercedVal` returns the `discreteOrMinValue` and `maxValue` fields as `ViInt32` values.

Caution *Do not modify the contents of the string the function returns in the* **commandString** *parameter.*

Parameters

Input

Name	Type	Description
coercedValue	`ViInt32`	The coerced value to search for in the range table.
rangeTable	`IviRangeTablePtr`	The address of the range table in which to search for the coerced value.

Output

Name	Type	Description
discreteOrMinValue	`ViInt32`	Returns the contents of the `discreteOrMinValue` field of the table entry the function locates. You can pass `VI_NULL`.
maxValue	`ViInt32`	Returns the contents of the `maxValue` field of the table entry the function locates. You can pass `VI_NULL`.

Name	Type	Description
tableIndex	ViInt32	Returns the 0-based index of the table entry the function locates. You can pass VI_NULL.
commandString	ViString	Returns the pointer in the cmdString field of the table entry the function locates. You can pass VI_NULL.
commandValue	ViInt32	Returns the contents of the cmdValue field of the table entry the function locates. You can pass VI_NULL.

Return Value

Name	Type	Description
status	ViStatus	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

Ivi_GetAttrRangeTable, Ivi_ValidateRangeTable, Ivi_GetRangeTableNumEntries, Ivi_GetViReal64EntryFromCoercedVal

Ivi_GetViInt32EntryFromIndex

```
ViStatus status = Ivi_GetViInt32EntryFromIndex (ViInt32 tableIndex,
                        IviRangeTablePtr rangeTable,
                        ViInt32 *discreteOrMinValue, ViInt32 *maxValue,
                        ViInt32 *coercedValue, ViString *commandString,
                        ViInt32 *commandValue);
```

Purpose

Returns the contents of the range table entry that is at the 0-based index you specify.

If you specify an index that is less than 0 or greater than or equal to the number of entries in the table, `Ivi_GetViInt32EntryFromIndex` returns an `IVI_ERROR_INVALID_VALUE` error.

`Ivi_GetViInt32EntryFromIndex` the `discreteOrMinValue`, `maxValue`, and `coercedValue` fields as `ViInt32` values.

Caution *Do not modify the contents of the string the function returns in the* **commandString** *parameter.*

Parameters

Input

Name	Type	Description
tableIndex	`ViInt32`	The 0-based index of the range table entry you want to extract.
rangeTable	`IviRangeTablePtr`	The address of the range table from which to extract the entry at the index you specify.

Output

Name	Type	Description
discreteOrMinValue	`ViInt32`	Returns the contents of the `discreteOrMinValue` field of the table entry the function locates. You can pass `VI_NULL`.
maxValue	`ViInt32`	Returns the contents of the `maxValue` field of the table entry the function locates. You can pass `VI_NULL`.

Name	Type	Description
coercedValue	`ViInt32`	Returns the contents of the `coercedValue` field of the table entry the function locates. You can pass `VI_NULL`.
commandString	`ViString`	Returns the pointer in the `cmdString` field of the table entry the function locates. You can pass `VI_NULL`.
commandValue	`ViInt32`	Returns the contents of the `cmdValue` field of the table entry the function locates. You can pass `VI_NULL`.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_GetAttrRangeTable`, `Ivi_ValidateRangeTable`, `Ivi_GetRangeTableNumEntries`, `Ivi_GetViReal64EntryFromIndex`

Ivi_GetViInt32EntryFromString

```
ViStatus status = Ivi_GetViInt32EntryFromString
                         (ViConstString commandString,
                         IviRangeTablePtr rangeTable,
                         ViInt32 *discreteOrMinValue, ViInt32 *maxValue,
                         ViInt32 *coercedValue, ViInt32 *tableIndex,
                         ViInt32 *commandValue);
```

Purpose

Finds the first range table entry for which the `cmdString` field matches the command string you specify. If the function finds an entry, it returns the contents of the entry. If it does not find an entry, it returns an `IVI_ERROR_INVALID_VALUE` error.

`Ivi_GetViInt32EntryFromString` compares strings in a case-sensitive manner.

If the command string you specify terminates with a carriage return (`'\r'`) or newline (`'\n'`) character, the `cmdString` fields in the table do not have to contain the termination character. `Ivi_GetViInt32EntryFromString` considers the strings to match if the string you specify begins with the string in the `cmdString` field, followed by a carriage return, newline, or ASCII NUL byte.

`Ivi_GetViInt32EntryFromString` returns the `discreteOrMinValue`, `maxValue`, and `coercedValue` fields as `ViInt32` values.

Parameters

Input

Name	Type	Description
commandString	`ViConstString`	The command string to search for in the range table.
rangeTable	`IviRangeTablePtr`	The address of the range table in which to search for the command string.

Output

Name	Type	Description
discreteOrMinValue	ViInt32	Returns the contents of the discreteOrMinValue field of the table entry the function locates. You can pass VI_NULL.
maxValue	ViInt32	Returns the contents of the maxValue field of the table entry the function locates. You can pass VI_NULL.
coercedValue	ViInt32	Returns the contents of the coercedValue field of the table entry the function locates. You can pass VI_NULL.
tableIndex	ViInt32	Returns the 0-based index of the table entry the function locates. You can pass VI_NULL.
commandValue	ViInt32	Returns the contents of the cmdValue field of the table entry the function locates. You can pass VI_NULL.

Return Value

Name	Type	Description
status	ViStatus	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

Ivi_GetAttrRangeTable, Ivi_ValidateRangeTable, Ivi_GetRangeTableNumEntries, Ivi_GetViReal64EntryFromString

Ivi_GetViInt32EntryFromValue

```
ViStatus status = Ivi_GetViInt32EntryFromValue (ViInt32 value,
                         IviRangeTablePtr rangeTable,
                         ViInt32 *discreteOrMinValue, ViInt32 *maxValue,
                         ViInt32 *coercedValue, ViInt32 *tableIndex,
                         ViString *commandString, ViInt32 *commandValue);
```

Purpose

Finds the first range table entry that applies to the `ViInt32` value you specify. If the function finds an entry, it returns the contents of the entry. If it does not find an entry, it returns an `IVI_ERROR_INVALID_VALUE` error.

If the range table type is `IVI_VAL_DISCRETE`, function searches for a match on the `discreteOrMinValue` field of each entry. If the range table type is `IVI_VAL_RANGED` or `IVI_VAL_COERCED`, the function searches until the value you specify falls within the range between the `discreteOrMinValue` and `maxValue` fields of an entry. The value falls within the range if is greater than or equal to the `discreteOrMinValue` and less than or equal to the `maxValue`.

`Ivi_GetViInt32EntryFromValue` returns the `discreteOrMinValue`, `maxValue`, and `coercedValue` fields as `ViInt32` values.

Caution ***Do not modify the contents of the string the function returns in the*** **commandString** ***parameter.***

Parameters

Input

Name	Type	Description
value	`ViInt32`	The value to compare against entries in the range table.
rangeTable	`IviRangeTablePtr`	The address of the range table in which to search for an entry that applies to the value.

Output

Name	Type	Description
discreteOrMinValue	ViInt32	Returns the contents of the discreteOrMinValue field of the table entry the function locates. You can pass VI_NULL.
maxValue	ViInt32	Returns the contents of the maxValue field of the table entry the function locates. You can pass VI_NULL.
coercedValue	ViInt32	Returns the contents of the coercedValue field of the table entry the function locates. You can pass VI_NULL.
tableIndex	ViInt32	Returns the 0-based index of the table entry the function locates. You can pass VI_NULL.
commandString	ViString	Returns the pointer in the cmdString field of the table entry the function locates. You can pass VI_NULL.
commandValue	ViInt32	Returns the contents of the cmdValue field of the table entry the function locates. You can pass VI_NULL.

Return Value

Name	Type	Description
status	ViStatus	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

Ivi_GetAttrRangeTable, Ivi_ValidateRangeTable, Ivi_GetRangeTableNumEntries, Ivi_GetViReal64EntryFromValue

Ivi_GetViReal64EntryFromCmdValue

```
ViStatus status = Ivi_GetViReal64EntryFromCmdValue (ViInt32 commandValue,
                          IviRangeTablePtr rangeTable,
                          ViReal64 *discreteOrMinValue,
                          ViReal64 *maxValue, ViReal64 *coercedValue,
                          ViInt32 *tableIndex, ViString *commandString);
```

Purpose

Finds the first range table entry for which the `cmdValue` field is equal to the command value you specify. If the function finds an entry, it returns the contents of the entry. If it does not find an entry, it returns an `IVI_ERROR_INVALID_VALUE` error.

Caution ***Do not modify the contents of the string the function returns in the*** **commandString** ***parameter.***

Parameters

Input

Name	Type	Description
commandValue	`ViInt32`	The command value to search for in the range table.
rangeTable	`IviRangeTable Ptr`	The address of the range table in which to search for the command value.

Output

Name	Type	Description
discreteOrMinValue	`ViReal64`	Returns the contents of the `discreteOrMinValue` field of the table entry the function locates. You can pass `VI_NULL`.
maxValue	`ViReal64`	Returns the contents of the `maxValue` field of the table entry the function locates. You can pass `VI_NULL`.
coercedValue	`ViReal64`	Returns the contents of the `coercedValue` field of the table entry the function locates. You can pass `VI_NULL`.

Name	Type	Description
tableIndex	ViInt32	Returns the 0-based index of the table entry the function locates. You can pass VI_NULL.
commandString	ViString	Returns the pointer in the cmdString field of the table entry the function locates. You can pass VI_NULL.

Return Value

Name	Type	Description
status	ViStatus	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

Ivi_GetAttrRangeTable, Ivi_ValidateRangeTable, Ivi_GetRangeTableNumEntries, Ivi_GetViInt32EntryFromCmdValue

Ivi_GetViReal64EntryFromCoercedVal

```
ViStatus status = Ivi_GetViReal64EntryFromCoercedVal (ViReal64 coercedValue,
                          IviRangeTablePtr rangeTable,
                          ViReal64 *discreteOrMinValue,
                          ViReal64 *maxValue, ViInt32 *tableIndex,
                          ViString *commandString, ViInt32 *commandValue);
```

Purpose

Finds the first range table entry for which the `coercedValue` field is equal to the coerced value you specify. If the function finds an entry, it returns the contents of the entry. If it does not find an entry, it returns an `IVI_ERROR_INVALID_VALUE` error.

`Ivi_GetViReal64EntryFromCoercedVal` performs all `ViReal64` comparisons using a comparison precision of 14 decimal digits.

Caution ***Do not modify the contents of the string the function returns in the*** **commandString** ***parameter.***

Parameters

Input

Name	Type	Description
coercedValue	`ViReal64`	The coerced value to search for in the range table.
rangeTable	`IviRangeTablePtr`	The address of the range table in which to search for the coerced value.

Output

Name	Type	Description
discreteOrMinValue	`ViReal64`	Returns the contents of the `discreteOrMinValue` field of the table entry the function locates. You can pass `VI_NULL`.
maxValue	`ViReal64`	Returns the contents of the `maxValue` field of the table entry the function locates. You can pass `VI_NULL`.
tableIndex	`ViInt32`	Returns the 0-based index of the table entry the function locates. You can pass `VI_NULL`.

Name	Type	Description
commandString	ViString	Returns the pointer in the cmdString field of the table entry the function locates. You can pass VI_NULL.
commandValue	ViInt32	Returns the contents of the cmdValue field of the table entry the function locates. You can pass VI_NULL.

Return Value

Name	Type	Description
status	ViStatus	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

Ivi_GetAttrRangeTable, Ivi_ValidateRangeTable, Ivi_GetRangeTableNumEntries, Ivi_GetViInt32EntryFromCoercedVal

Ivi_GetViReal64EntryFromIndex

```
ViStatus status = Ivi_GetViReal64EntryFromIndex (ViInt32 tableIndex,
                           IviRangeTablePtr rangeTable,
                           ViReal64 *discreteOrMinValue,
                           ViReal64 *maxValue, ViReal64 *coercedValue,
                           ViString *commandString, ViInt32 *commandValue);
```

Purpose

Returns the contents of the range table entry that is at the 0-based index you specify.

If you specify an index that is less than 0 or greater than or equal to the number of entries in the table, `Ivi_GetViReal64EntryFromIndex` returns an `IVI_ERROR_INVALID_VALUE` error.

Caution ***Do not modify the contents of the string the function returns in the* commandString *parameter.***

Parameters

Input

Name	Type	Description
tableIndex	`ViInt32`	The 0-based index of the range table entry you want to extract.
rangeTable	`IviRangeTablePtr`	The address of the range table from which to extract the entry at the index you specify.

Output

Name	Type	Description
discreteOrMinValue	`ViReal64`	Returns the contents of the `discreteOrMinValue` field of the table entry the function locates. You can pass `VI_NULL`.
maxValue	`ViReal64`	Returns the contents of the `maxValue` field of the table entry the function locates. You can pass `VI_NULL`.
coercedValue	`ViReal64`	Returns the contents of the `coercedValue` field of the table entry the function locates. You can pass `VI_NULL`.

Name	Type	Description
commandString	`ViString`	Returns the pointer in the `cmdString` field of the table entry the function locates. You can pass `VI_NULL`.
commandValue	`ViInt32`	Returns the contents of the `cmdValue` field of the table entry the function locates. You can pass `VI_NULL`.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_GetAttrRangeTable`, `Ivi_ValidateRangeTable`, `Ivi_GetRangeTableNumEntries`, `Ivi_GetViInt32EntryFromIndex`

Ivi_GetViReal64EntryFromString

```
ViStatus status = Ivi_GetViReal64EntryFromString
                           (ViConstString commandString,
                           IviRangeTablePtr rangeTable,
                           ViReal64 *discreteOrMinValue,
                           ViReal64 *maxValue, ViReal64 *coercedValue,
                           ViInt32 *tableIndex, ViInt32 *commandValue);
```

Purpose

Finds the first range table entry for which the `cmdString` field matches the command string you specify. If the function finds an entry, it returns the contents of the entry. If it does not find an entry, it returns an `IVI_ERROR_INVALID_VALUE` error.

`Ivi_GetViReal64EntryFromString` compares strings in a case-sensitive manner.

If the command string you specify terminates with a carriage return (`'\r'`) or newline (`'\n'`) character, the `cmdString` fields in the table do not have to contain the termination character. `Ivi_GetViReal64EntryFromString` considers the strings to match if the string you specify begins with the string in the `cmdString` field, followed by a carriage return, newline, or ASCII NUL byte.

Parameters

Input

Name	Type	Description
commandString	`ViString`	The command string to search for in the range table.
rangeTable	`IviRangeTablePtr`	The address of the range table in which to search for the command string.

Output

Name	Type	Description
discreteOrMinValue	`ViReal64`	Returns the contents of the `discreteOrMinValue` field of the table entry the function locates. You can pass `VI_NULL`.
maxValue	`ViReal64`	Returns the contents of the `maxValue` field of the table entry the function locates. You can pass `VI_NULL`.

Name	Type	Description
coercedValue	`ViReal64`	Returns the contents of the `coercedValue` field of the table entry the function locates. You can pass `VI_NULL`.
tableIndex	`ViInt32`	Returns the 0-based index the function locates. You can pass `VI_NULL`.
commandValue	`ViInt32`	Returns the contents of the `cmdValue` field of the table entry the function locates. You can pass `VI_NULL`.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_GetAttrRangeTable`, `Ivi_ValidateRangeTable`, `Ivi_GetRangeTableNumEntries`, `Ivi_GetViInt32EntryFromString`

Ivi_GetViReal64EntryFromValue

```
ViStatus status = Ivi_GetViReal64EntryFromValue (ViReal64 value,
                          IviRangeTablePtr rangeTable,
                          ViReal64 *discreteOrMinValue,
                          ViReal64 *maxValue, ViReal64 *coercedValue,
                          ViInt32 *tableIndex, ViString *commandString,
                          ViInt32 *commandValue);
```

Purpose

Function finds the first range table entry that applies to the `ViReal64` value you specify. If the function finds an entry, it returns the contents of the entry. If it does not find an entry, it returns an `IVI_ERROR_INVALID_VALUE` error.

If the range table type is `IVI_VAL_DISCRETE`, function searches for a match on the `discreteOrMinValue` field of each entry. If the range table type is `IVI_VAL_RANGED` or `IVI_VAL_COERCED`, the function searches until the value you specify falls within the range between the `discreteOrMinValue` and `maxValue` fields of an entry. The value falls within the range if is greater than or equal to the `discreteOrMinValue` and less than or equal to the `maxValue`.

`Ivi_GetViReal64EntryFromValue` performs all `ViReal64` comparisons using a comparison precision of 14 decimal digits.

Caution ***Do not modify the contents of the string the function returns in the*** **commandString** ***parameter.***

Parameters

Input

Name	Type	Description
value	`ViReal64`	The value to compare against entries in the range table.
rangeTable	`IviRangeTablePtr`	The address of the range table in which to search for an entry that applies to the value.

Output

Name	Type	Description
discreteOrMinValue	ViReal64	Returns the contents of the discreteOrMinValue field of the table entry the function locates. You can pass VI_NULL.
maxValue	ViReal64	Returns the contents of the maxValue field of the table entry the function locates. You can pass VI_NULL.
coercedValue	ViReal64	Returns the contents of the coercedValue field of the table entry the function locates. You can pass VI_NULL.
tableIndex	ViInt32	Returns the 0-based index of the range table entry the function locates. You can pass VI_NULL.
commandString	ViString	Returns the pointer in the cmdString field of the table entry the function locates. You can pass VI_NULL.
commandValue	ViInt32	Returns the contents of the cmdValue field of the table entry the function locates. You can pass VI_NULL.

See Also

Ivi_GetAttrRangeTable, Ivi_ValidateRangeTable,
Ivi_GetRangeTableNumEntries, Ivi_GetViInt32EntryFromValue

Ivi_InstrSpecificErrorQueueSize

```
ViStatus status = Ivi_InstrSpecificErrorQueueSize (ViSession vi,
                                   ViInt32 *errorQueueSize);
```

Purpose

Returns the number of entries currently in the instrument-specific error queue.

Use the instrument-specific error queue if querying the instrument for its status causes the instrument to lose the error value. In your check status callback, call `Ivi_QueueInstrSpecificError` to insert the instrument error code in the queue, and then return the `IVI_ERROR_INSTR_SPECIFIC` error code from the callback. In your `Prefix_error_query` function, call `Ivi_InstrSpecificErrorQueueSize` to determine if there is an error in the queue. If not, invoke the check status callback directly. In either case, if there is an error, call `Ivi_DequeueInstrSpecificError` to retrieve the error.

Refer to the *Instruments without Error Queues* discussion in the *Check Status Callback* section of Chapter 2, *IVI Architecture Overview*, for more information,

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.

Output

Name	Type	Description
errorQueueSize	`ViInt32`	Returns the number of errors currently in the error queue.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_QueueInstrSpecificError`, `Ivi_DequeueInstrSpecificError`, `Ivi_ClearInstrSpecificErrorQueue`

Ivi_InterchangeCheck

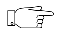
```
ViBoolean interchangeCheck = Ivi_InterchangeCheck (ViSession vi);
```

Purpose

Returns the current value of the IVI_ATTR_INTERCHANGE_CHECK attribute for the IVI session you specify. The attribute determines whether class drivers perform interchangeability checking. The specification for each instrument class defines the rules for interchangeability checking for that class.

High-level functions in class instrument drivers use Ivi_InterchangeCheck. Ivi_InterchangeCheck provides fast, convenient access to the IVI_ATTR_INTERCHANGE_CHECK attribute because it performs no error checking and does not lock the session.

If you pass an invalid session handle, Ivi_InterchangeCheck returns VI_FALSE.


Note ***Do not call*** Ivi_InterchangeCheck ***unless you have already locked the session.***

Parameters

Input

Name	Type	Description
vi	ViSession	The ViSession handle that you obtain from Ivi_SpecificDriverNew. The handle identifies a particular IVI session.

Return Value

Name	Type	Description
interchangeCheck	ViBoolean	1 = VI_TRUE - Interchange checking on 0 = VI_FALSE - Interchange checking off

Ivi_InvalidateAllAttributes

```
ViStatus status = Ivi_InvalidateAllAttributes (ViSession vi);
```

Purpose

Invalidates the cache values of all attributes for an IVI session. For each channel-based attribute, the function invalidates the cache values for all channels.

Invalidating a cache value for an attribute ensures that the next call to an `Ivi_GetAttribute` or `Ivi_SetAttribute` function on the attribute invokes the read or write callback for the attribute.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_AddAttributeInvalidation`, `Ivi_DeleteAttributeInvalidation`, `Ivi_InvalidateAttribute`, `Ivi_GetInvalidationList`

Ivi_InvalidateAttribute

```
ViStatus status = Ivi_InvalidateAttribute (ViSession vi,
                           ViConstString channel, ViAttr attributeID);
```

Purpose

Marks the cache value of an attribute as invalid. This ensures that the next call to an Ivi_GetAttribute or Ivi_SetAttribute function on the attribute invokes the read or write callback for the attribute.

For a channel-based attribute, you can invalidate the attribute on a specific channel or on all channels.

Parameters

Input

Name	Type	Description
vi	ViSession	The ViSession handle that you obtain from Ivi_SpecificDriverNew. The handle identifies a particular IVI session.
channel	ViConstString	The name of a particular channel, IVI_VAL_ALL_CHANNELS, VI_NULL, or an empty string. Refer to the *Parameter Discussion* section.
attributeID	ViAttr	The ID of the attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.

Return Value

Name	Type	Description
status	ViStatus	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

If the attribute is channel-based, you can specify a particular channel or you can pass IVI_VAL_ALL_CHANNELS for the channel parameter. To specify a particular channel, you can pass one of the channel strings that the specific instrument driver defines, or you can pass a virtual channel name the user defines in the ivi.ini configuration file.

If the attribute you specify is not channel-based, pass VI_NULL or an empty string for the channel parameter.

See Also

Ivi_AddAttributeInvalidation, Ivi_DeleteAttributeInvalidation, Ivi_InvalidateAllAttributes, Ivi_GetInvalidationList

Ivi_IOSession

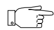
```
ViSession ioSession = Ivi_IOSession (ViSession vi);
```

Purpose

Returns the current value of the IVI_ATTR_IO_SESSION attribute for the IVI session you specify.

High-level functions in specific instrument drivers use Ivi_IOSession. Ivi_IOSession provides fast, convenient access to the IVI_ATTR_IO_SESSION attribute because it performs no error checking and does not lock the session.

If you pass an invalid session handle, Ivi_IOSession returns VI_NULL.



Note ***Do not call*** Ivi_IOSession ***unless you have already locked the session.***

Parameters

Input

Name	Type	Description
vi	ViSession	The ViSession handle that you obtain from Ivi_SpecificDriverNew. The handle identifies a particular IVI session.

Return Value

Name	Type	Description
ioSession	ViSession	The current value of IVI_ATTR_IO_SESSION.

See Also

Ivi_WriteInstrData, Ivi_ReadInstrData, Ivi_ReadToFile, Ivi_WriteFromFile

Ivi_LockSession

```
ViStatus status = Ivi_LockSession (ViSession vi, ViBoolean *callerHasLock);
```

Purpose

Obtains a multithread lock on the instrument session. Before it does so, `Ivi_LockSession` waits until all other execution threads have released their locks on the instrument session.

You can use `Ivi_LockSession` to protect a section of code which requires exclusive access to the instrument. This occurs when you take multiple actions that affect the instrument and you want to ensure that other execution threads do not disturb the instrument state until all of your actions execute. For example, if you set various instrument attributes and then trigger a measurement, you must ensure no other execution thread modifies the attribute values until you finish taking the measurement.

You can safely make nested calls to `Ivi_LockSession` within the same thread. To completely unlock the session, you must balance each call to `Ivi_LockSession` with a call to `Ivi_UnlockSession`. If, however, you use the **callerHasLock** parameter in all calls to `Ivi_LockSession` and `Ivi_UnlockSession` within a function, the IVI Library locks the session only once within the function regardless of the number of calls you make to `Ivi_LockSession`. This allows you to call `Ivi_UnlockSession` just once at the end of the function.

User applications, instrument drivers, and the IVI Library functions all have the ability to obtain a lock. The IVI Library functions always release the lock before they return. Instrument driver functions must do the same.

Instrument drivers export `Ivi_LockSession` to the user through the *`Prefix`*`_LockSession` function.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.

Input/Output

Name	Type	Description
callerHasLock	ViBoolean	Indicates whether the calling function currently has a lock on the IVI session. You can pass VI_NULL. Refer to the *Parameter Discussion* section.

Return Value

Name	Type	Description
status	ViStatus	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

The **callerHasLock** parameter serves as a convenience. If you do not want to use it, pass VI_NULL.

Use **callerHasLock** in complex functions to keep track of whether you obtain a lock and therefore need to unlock the session. Pass the address of a local ViBoolean variable. Initialize the local variable to VI_FALSE when you declare it. Pass the same address to any other calls you make to Ivi_LockSession or Ivi_UnlockSession in the same function.

Ivi_LockSession and Ivi_UnlockSession each inspect the current value of **callerHasLock** and take the following actions:

- If the value is VI_TRUE, Ivi_LockSession does not lock the session again. If the value is VI_FALSE, Ivi_LockSession obtains the lock and sets the value of the parameter to VI_TRUE.
- If the value is VI_FALSE, Ivi_UnlockSession does not attempt to unlock the session. If the value is VI_TRUE, Ivi_UnlockSession releases the lock and sets the value of the parameter to VI_FALSE.

Thus, you can call `Ivi_UnlockSession` at the end of your function without worrying about whether you actually have a lock. The following example code shows how to use the **callerHasLock** parameter.

```
ViStatus Prefix_Func (ViSession vi, ViInt32 flags)
{
    ViStatus error = VI_SUCCESS;
    ViBoolean haveLock = VI_FALSE;
    if (flags & BIT_1)
        {
        viCheckErr( Ivi_LockSession(vi, &haveLock));
        viCheckErr( TakeAction1(vi));
        if (flags & BIT_2)
            {
            viCheckErr( Ivi_UnlockSession(vi, &haveLock));
            viCheckErr( TakeAction2(vi));
            viCheckErr( Ivi_LockSession(vi, &haveLock);
            }
        if (flags & BIT_3)
            viCheckErr( TakeAction3(vi));
        }
Error:
    /*
        At this point, you cannot really be sure that
        you have the lock. Fortunately, the haveLock
        variable takes care of that for you.
    */
    Ivi_UnlockSession(vi, &haveLock);
    return error;
}
```

See Also

`Ivi_UnlockSession`, `Ivi_SpecificDriverNew`, `Ivi_Dispose`

Ivi_NeedToCheckStatus

```
ViBoolean needToCheckStatus = Ivi_NeedToCheckStatus (ViSession vi);
```

Purpose

Returns an indication of whether the instrument driver has interacted with the instrument since the last time the IVI engine or the driver checked the status of the instrument.

Typically, the *Prefix*_CheckStatus function that is internal to an instrument driver calls Ivi_NeedToCheckStatus to help determine whether it is necessary to invoke the check status callback for the session.

The IVI engine maintains an internal needToCheckStatus variable for each session indicating whether it is necessary to check the status of the instrument. When you create a new session, the initial value of the variable is VI_TRUE. The IVI engine sets the needToCheckStatus variable to VI_TRUE when it invokes the read or write callback for an attribute for which the IVI_VAL_DONT_CHECK_STATUS flag is 0. The Ivi_WriteInstrData and Ivi_WriteFromFile functions also set the variable to VI_TRUE. The IVI engine sets the variable to VI_FALSE after it invokes the check status callback successfully.

The Ivi_SetNeedToCheckStatus function allows an instrument driver to set the state of the internal needToCheckStatus variable. A driver typically sets the variable to VI_TRUE before it attempts direct instrument I/O. It sets it to VI_FALSE after it calls the check status callback successfully.

Ivi_NeedToCheckStatus returns the value of the internal needToCheckStatus variable. If the **vi** parameter is invalid, Ivi_NeedToCheckStatus returns VI_FALSE.

Note ***Do not call*** Ivi_NeedToCheckStatus ***unless you have already locked the session.***

Parameter List

Input

Name	Type	Description
vi	ViSession	The ViSession handle that you obtain from Ivi_SpecificDriverNew. The handle identifies a particular IVI session.

Return Value

Name	Type	Description
needToCheckStatus	ViBoolean	1 = VI_TRUE - Interacted with instrument since last status check 0 = VI_FALSE - No interaction with instrument since last status check

See Also

Ivi_SetNeedToCheckStatus

Ivi_QueryInstrStatus

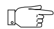
```
ViBoolean queryInstrStatus = Ivi_QueryInstrStatus (ViSession vi);
```

Purpose

Returns the current value of the IVI_ATTR_QUERY_INSTR_STATUS attribute for the IVI session you specify. The attribute determines whether or not to query the instrument error status after each operation.

High-level functions in specific instrument drivers use Ivi_QueryInstrStatus. Ivi_QueryInstrStatus provides fast, convenient access to the IVI_ATTR_QUERY_INSTR_STATUS attribute because it performs no error checking and does not lock the session.

If you pass an invalid session handle, Ivi_QueryInstrStatus returns VI_FALSE.

 **Note** ***Do not call*** Ivi_QueryInstrStatus ***unless you have already locked the session.***

Parameters

Input

Name	Type	Description
vi	ViSession	The ViSession handle that you obtain from Ivi_SpecificDriverNew. The handle identifies a particular IVI session.

Return Value

Name	Type	Description
queryInstrStatus	ViBoolean	1 = VI_TRUE - Query instrument status 0 = VI_FALSE - Do not query instrument status

Ivi_QueueInstrSpecificError

```
ViStatus status = Ivi_QueueInstrSpecificError (ViSession vi,
                           ViInt32 instrumentError, ViString errorMessage);
```

Purpose

Inserts a new entry at the end of the instrument-specific error queue. The instrument-specific error queue is a software record of the error values you retrieve from the instrument.

Use the instrument-specific error queue if querying the instrument for its status causes the instrument to lose the error value. In your check status callback, call `Ivi_QueueInstrSpecificError` to insert the instrument error code in the queue, and then return the `IVI_ERROR_INSTR_SPECIFIC` error code from the callback. In your `Prefix_error_query` function, call `Ivi_InstrSpecificErrorQueueSize` to determine if there is an error in the queue. If not, invoke the check status callback directly. In either case, if there is an error, call `Ivi_DequeueInstrSpecificError` to retrieve the error.

Refer to the *Instruments without Error Queues* and the *Check Status Callback* sections of Chapter 2, *IVI Architecture Overview*, for more information.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
instrumentError	`ViInt32`	Pass the numeric error code to insert at the end of the instrument-specific error queue.
errorMessage	`ViString`	Pass the error description string to insert at the end of the instrument-specific error queue.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_DequeueInstrSpecificError`, `Ivi_InstrSpecificErrorQueueSize`, `Ivi_ClearInstrSpecificErrorQueue`

Ivi_RangeChecking

```
ViBoolean rangeCheck = Ivi_RangeChecking (ViSession vi);
```

Purpose

Returns the current value of the `IVI_ATTR_RANGE_CHECK` attribute for the IVI session you specify. The attribute determines whether or not to range check parameters to instrument driver functions.

High-level functions in specific instrument drivers use `Ivi_RangeChecking`. `Ivi_RangeChecking` provides fast, convenient access to the `IVI_ATTR_RANGE_CHECK` attribute because it performs no error checking and does not lock the session.

If you pass an invalid session handle, `Ivi_RangeChecking` returns `VI_FALSE`.

Note ***Do not call*** `Ivi_RangeChecking` ***unless you have already locked the session.***

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.

Return Value

Name	Type	Description
rangeCheck	`ViBoolean`	1 = `VI_TRUE` - Range check 0 = `VI_FALSE` - Do not range check

Ivi_RangeTableFree

```
ViStatus status = Ivi_RangeTableFree (ViSession vi,
                      IviRangeTablePtr rangeTable,
                      ViBoolean freeCommandStrings);
```

Purpose

Deallocates a range table you create dynamically with `Ivi_RangeTableNew`. It calls `Ivi_Free` to free the `IviRangeTable` structure and the array of `IviRangeTableEntry` structures. You can specify whether to free the `cmdString` field in each entry.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
rangeTable	`IviRangeTable Ptr`	The table pointer you obtain from `Ivi_RangeTableNew`.
freeCommandStrings	`ViBoolean`	Pass `VI_TRUE` if you want the function to call `Ivi_Free` on the `cmdString` field in each table entry. Otherwise, pass `VI_FALSE`. Do not pass `VI_TRUE` unless you allocate the command strings using `Ivi_Alloc`.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_RangeTableNew`, `Ivi_SetRangeTableEntry`, `Ivi_SetRangeTableEnd`

Ivi_RangeTableNew

```
ViStatus status = Ivi_RangeTableNew (ViSession vi, ViInt32 numberOfEntries,
                        ViInt32 typeOfTable, ViBoolean hasMinimum,
                        ViBoolean hasMaximum,
                        IviRangeTablePtr *rangeTable);
```

Purpose

Dynamically allocates a range table. Range tables you create with `Ivi_RangeTableNew` are called *dynamic range tables*. Range tables you define statically in your source code are called *static range tables*.

If the values in the range table for a particular attribute can change depending on the settings of other attributes, you must create it as a dynamic range table. To allow for multithreading and multiple instances of the same instrument type, you create a separate dynamic range table for the attribute in each IVI session.

`Ivi_RangeTableNew` allocates the `IviRangeTable` structure and an array of `IviRangeTableEntry` structures. It allocates space in the array for the number of entries you specify, which must include the termination entry. It sets the last entry as the termination entry. Use the `Ivi_SetRangeTableEntry` function to set the values within the entries.

If the number of entries in the table varies, specify the maximum number of entries that it can contain. Use the `Ivi_SetRangeTableEnd` function to change the location of the termination entry.

The IVI engine keeps track of the memory you allocate with `Ivi_RangeTableNew` in each session. It automatically frees the memory when you call `Ivi_Dispose` on the session.

If you want to deallocate the table before the session ends, call the `Ivi_RangeTableFree` function.

Refer to the *Range Tables* section in Chapter 2, *IVI Architecture Overview*, for more information on these fields.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
numberOfEntries	`ViInt32`	The number of entries you want in the range table, including the termination entry.
typeOfTable	`ViInt32`	The type of range table you want to create. Values: 0 = `IVI_VAL_DISCRETE` 1 = `IVI_VAL_RANGED` 2 = `IVI_VAL_COERCED` Refer to the *Parameter Discussion* section.
hasMinimum	`ViBoolean`	Pass `VI_TRUE` if the table, as a whole, contains a meaningful minimum value. Pass `VI_FALSE` otherwise. If **typeOfTable** is `IVI_VAL_COERCED`, the minimum value represents the minimum coerced value.
hasMaximum	`ViBoolean`	Pass `VI_TRUE` if the table, as a whole, contains a meaningful maximum value. Pass `VI_FALSE` otherwise. If **typeOfTable** is `IVI_VAL_COERCED`, the maximum value represents the maximum coerced value.

Output

Name	Type	Description
rangeTable	`IviRangeTablePtr`	Returns a pointer to the range table the function dynamically allocates.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

The typeOfTable parameter indicates how the IVI engine interprets the discreteOrMinValue, maxValue, and coercedValue fields in each entry. The following describes each range table type.

- IVI_VAL_DISCRETE—Each table entry defines a discrete value. The discreteOrMinValue field contains the discrete value. The maxValue and coercedValue fields are not used.
- IVI_VAL_RANGED—Each table entry defines a range with a minimum and a maximum value. The discreteOrMinValue field holds the minimum value, and 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, create the range table with only one entry other than the terminating entry.
- IVI_VAL_COERCED—Each table 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 specify to the instrument using one discrete value. The discreteOrMinValue holds the minimum value of the range, maxValue holds the maximum value, and coercedValue holds the discrete value that represents the range.

See Also

Ivi_RangeTableFree, Ivi_SetRangeTableEntry, Ivi_SetRangeTableEnd, Ivi_Alloc, Ivi_FreeAll, Ivi_Dispose, Ivi_SetAttrRangeTableCallback, Ivi_GetAttrRangeTable, Ivi_GetStoredRangeTablePtr, Ivi_SetStoredRangeTablePtr, Ivi_ValidateRangeTable, Ivi_GetRangeTableNumEntries

Ivi_ReadInstrData

```
ViStatus status = Ivi_ReadInstrData (ViSession vi, ViInt32 numBytesToRead,
                         ViChar readBuffer[], ViInt32 *numBytesRead);
```

Purpose

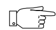
Reads data directly from an instrument using VISA I/O. The function bypasses the IVI engine's attribute-state-caching mechanism. Use the `Ivi_ReadInstrData` function only to implement the *`Prefix`*`_ReadInstrData` function that your instrument driver exports to the user.

`Ivi_ReadInstrData` assumes that the `IVI_ATTR_IO_SESSION` attribute for the IVI session you specify holds a valid VISA session for the instrument.

If the value you specify for **numBytesToRead** is less than the number of bytes in the instrument's output buffer, you must call `Ivi_ReadInstrData` again to empty the output buffer. If you do not empty the instrument's output buffer, the instrument might return invalid data in response to subsequent requests.

If the actual number of bytes you receive is less than the number of bytes you specify in the **numBytesToRead** parameter, the instrument's output buffer has probably emptied. If the number of bytes received is 0, the most probable cause is that no data was available at the instrument's output buffer.

If data is not available at the instrument's output buffer when you call `Ivi_ReadInstrData`, the instrument might not respond. In that case, the function does not return until the VISA I/O call times out. If you disable the VISA timeout, the function hangs indefinitely.



Note `Ivi_ReadInstrData` ***does not place an ASCII NUL byte in*** **readBuffer** ***to terminate the data, nor does it clear the buffer beyond the bytes it actually receives from the instrument.***

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
numBytesToRead	`ViInt32`	The maximum number of bytes to read from the instrument.

Output

Name	Type	Description
readBuffer	`ViChar` array	The buffer in which the function places the data it receives from the instrument. Must contain at least as many bytes as you specify in **numBytesToRead**.
numBytesRead	`ViInt32`	Returns the actual number of bytes the function received from the instrument. This is the value that the VISA `viRead` function returns.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_WriteInstrData`, `Ivi_ReadToFile`, `Ivi_WriteFromFile`, `Ivi_IOSession`

Ivi_ReadToFile

```
ViStatus status = Ivi_ReadToFile (ViSession vi, ViConstString filename,
                         ViInt32 readNumberofBytes,
                         ViInt32 fileAction,ViInt32 *returnCount);
```

Purpose

Reads data from an instrument using VISA I/O and writes it to a file you specify. Use `Ivi_ReadToFile` internally in your instrument driver.

`Ivi_ReadToFile` assumes that the `IVI_ATTR_IO_SESSION` attribute for the IVI session you specify holds a valid VISA session for the instrument.

`Ivi_ReadToFile` opens the file in binary mode.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
filename	`ViConstString`	The pathname of the file to write the data to. Refer to the *Parameter Discussion* section.
readNumberofBytes	`ViInt32`	The maximum number of bytes to read from the instrument.
fileAction	`ViInt32`	Specifies whether you want the function to append the data it receives from the instrument to an existing file or to create a new file. Values: 1 = `IVI_VAL_TRUNCATE` 2 = `IVI_VAL_APPEND` Refer to the *Parameter Discussion* section.

Output

Name	Type	Description
returnCount	`ViInt32`	Returns the number of bytes the function successfully writes to the file.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

For the **filename** parameter, you can specify an absolute pathname, a relative pathname, or a simple filename. The function treats relative pathnames and simple filenames as relative to the current working directory.

If you specify a literal string for the **filename** parameter under Windows, be sure to use double backslashes to represent one backslash in the pathname.

If the file does not currently exist, `Ivi_ReadToFile` creates it and writes the instrument data to it. If the file does exist, the value you specify in **fileAction** controls what `Ivi_ReadToFile` does. If you pass `IVI_VAL_TRUNCATE` for **fileAction**, `Ivi_ReadToFile` deletes the contents of the file and replaces it with the instrument data. If you pass `IVI_VAL_APPEND`, `Ivi_ReadToFile` appends the instrument data to the end of the file.

See Also

`Ivi_WriteFromFile`, `Ivi_ReadInstrData`, `Ivi_WriteInstrData`, `Ivi_IOSession`

Ivi_RestrictAttrToChannels

```
ViStatus status = Ivi_RestrictAttrToChannels (ViSession vi,
                         ViAttr attributeID,
                         ViConstString channelStrings);
```

Purpose

Restricts an attribute to specific channels, thereby preventing you from using the attribute on other channels.

You can call `Ivi_RestrictAttrToChannels` only on attributes for which you enable the `IVI_VAL_MULTI_CHANNEL` flag.

When you initially add an attribute, it applies to all channels. If you want it to apply to only a subset, call `Ivi_RestrictAttrToChannels`.

When you add channels to the channel table incrementally by calling `Ivi_AddToChannelTable`, each attribute for which the `IVI_VAL_MULTI_CHANNEL` flag is enabled applies to each of the new channels. To restrict an attribute to a subset of the new channels, you can call `Ivi_RestrictAttrToChannels` again. Do not include in the channel string any of the channels that existed at the time of your last call to `Ivi_RestrictAttrToChannels` on that attribute. Each call to `Ivi_RestrictAttrToChannels` applies only to channels that you have added since your last call to `Ivi_RestrictAttrToChannels` on that attribute. To disable the attribute for all of the new channels, pass an empty string for the **channelStrings** parameter.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
attributeID	`ViAttr`	The ID of the attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.
channelStrings	`ViConstString`	A list of the channel strings to which you want to restrict the attribute you specify. You must separate channel strings with commas. You can include spaces after the commas.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Example

After the following four function calls execute, the `PREFIX_ATTR_RANGE` attribute is valid only for channels `"1"`, `"2"`, and `"7"`.

```
Ivi_BuildChannelTable (vi, "1,2,3,4", VI_FALSE, VI_NULL);
Ivi_RestrictAttrToChannels (vi, PREFIX_ATTR_RANGE, "1,2");
Ivi_AddToChannelTable (vi, "5,6,7,8", VI_FALSE, VI_NULL);
Ivi_RestrictAttrToChannels (vi, PREFIX_ATTR_RANGE, "7");
```

See Also

`Ivi_BuildChannelTable`, `Ivi_AddToChannelTable`,
`Ivi_ValidateAttrForChannel`, `Ivi_CoerceChannelName`,
`Ivi_GetUserChannelName`, `Ivi_GetNthChannelString`

Ivi_SetAttrCheckCallbackViInt32
Ivi_SetAttrCheckCallbackViReal64
Ivi_SetAttrCheckCallbackViString
Ivi_SetAttrCheckCallbackViBoolean
Ivi_SetAttrCheckCallbackViSession
Ivi_SetAttrCheckCallbackViAddr

```
ViStatus status = Ivi_SetAttrCheckCallbackViReal64 (ViSession vi,
                           ViAttr attributeID,
                           CheckAttrViReal64_CallbackPtr checkCallback);
ViStatus status = Ivi_SetAttrCheckCallbackViInt32 (ViSession vi,
                           ViAttr attributeID,
                           CheckAttrViInt32_CallbackPtr checkCallback);
ViStatus status = Ivi_SetAttrCheckCallbackViString (ViSession vi,
                           ViAttr attributeID,
                           CheckAttrViString_CallbackPtr checkCallback);
ViStatus status = Ivi_SetAttrCheckCallbackViBoolean (ViSession vi,
                           ViAttr attributeID,
                           CheckAttrViBoolean_CallbackPtr checkCallback);
ViStatus status = Ivi_SetAttrCheckCallbackViSession (ViSession vi,
                           ViAttr attributeID,
                           CheckAttrViSession_CallbackPtr checkCallback);
ViStatus status = Ivi_SetAttrCheckCallbackViAddr (ViSession vi,
                           ViAttr attributeID, CheckAttrViAddr_CallbackPtr
                           checkCallback);
```

Purpose

Sets the check callback function for an attribute. The IVI engine invokes the check callback function to validate new values that you attempt to set the attribute to.

The IVI engine supplies default check callbacks for `ViInt32` and `ViReal64` attributes. The callbacks use the range table for the attribute to validate the value. The IVI engine automatically installs a default check callback when you create a `ViInt32` or `ViReal64` attribute with a non-NULL range table. It also does so when the attribute does not have a check callback and you install a range table callback for it. If you want to specify your own callback function but you want to use the default check callback within your function, you can call `Ivi_DefaultCheckCallbackViInt32` or `Ivi_DefaultCheckCallbackViReal64`.

If you do not want the IVI engine to invoke a check callback for the attribute, pass `VI_NULL` for the **checkCallback** parameter.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
attributeID	depends on the data type of the attribute	The ID of the attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.
checkCallback	depends on the data type of the attribute	The check callback function you want the IVI engine to invoke to validate attribute values. Can be `VI_NULL`.

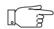
Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

The function you specify for the **checkCallback** parameter must have one of the following prototypes, based on data type:

```
ViStatus _VI_FUNC ViInt32CheckCallback(ViSession vi,
                           ViConstString channelName,
                           ViAttr attributeId, ViInt32 value);

ViStatus _VI_FUNC ViReal64CheckCallback(ViSession vi,
                           ViConstString channelName,
                           ViAttr attributeId, ViReal64 value);

ViStatus _VI_FUNC ViStringCheckCallback(ViSession vi,
                           ViConstString channelName,
                           ViAttr attributeId, ViConstString value);

ViStatus _VI_FUNC ViBooleanCheckCallback(ViSession vi,
                           ViConstString channelName,
                           ViAttr attributeId, ViBoolean value);

ViStatus _VI_FUNC ViSessionCheckCallback(ViSession vi,
                           ViConstString channelName,
                           ViAttr attributeId, ViSession value);

ViStatus _VI_FUNC ViAddrCheckCallback(ViSession vi,
                           ViConstString channelName,
                           ViAttr attributeId, ViAddr value);
```

 **Note** *If you want to use the* **Edit Instrument Attributes** *command to develop your instrument driver source code, retain the parameter names as shown in the prototype for the callback.*

See Also

Ivi_AddAttributeViInt32, Ivi_AddAttributeViReal64, Ivi_DefaultCheckCallbackViInt32, Ivi_DefaultCheckCallbackViReal64, Ivi_ValidateRangeTable, Ivi_GetRangeTableNumEntries, Ivi_GetViInt32EntryFromValue, Ivi_GetViReal64EntryFromValue

Ivi_SetAttrCoerceCallbackViInt32
Ivi_SetAttrCoerceCallbackViReal64
Ivi_SetAttrCoerceCallbackViString
Ivi_SetAttrCoerceCallbackViBoolean
Ivi_SetAttrCoerceCallbackViSession
Ivi_SetAttrCoerceCallbackViAddr

```
ViStatus status = Ivi_SetAttrCoerceCallbackViInt32 (ViSession vi,
                           ViAttr attributeID,
                           CoerceAttrViInt32_CallbackPtr coerceCallback);
ViStatus status = Ivi_SetAttrCoerceCallbackViReal64 (ViSession vi,
                           ViAttr attributeID,
                           CoerceAttrViReal64_CallbackPtr coerceCallback);
ViStatus status = Ivi_SetAttrCoerceCallbackViString (ViSession vi,
                           ViAttr attributeID,
                           CoerceAttrViString_CallbackPtr coerceCallback);
ViStatus status = Ivi_SetAttrCoerceCallbackViBoolean (ViSession vi,
                           ViAttr attributeID,
                           CoerceAttrViBoolean_CallbackPtr coerceCallback);
ViStatus status = Ivi_SetAttrCoerceCallbackViSession (ViSession vi,
                           ViAttr attributeID,
                           CoerceAttrViSession_CallbackPtr coerceCallback);
ViStatus status = Ivi_SetAttrCoerceCallbackViAddr (ViSession vi,
                           ViAttr attributeID,
                           CoerceAttrViAddr_CallbackPtr coerceCallback);
```

Purpose

Sets the coerce callback function for an attribute. The IVI engine invokes the coerce callback function when you attempt to set the 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 a value to send to the instrument.

The IVI engine supplies default coerce callbacks for `ViInt32` and `ViReal64` attributes. The default coerce callbacks use the range table for the attribute. The IVI engine automatically installs a default coerce callback when you create a `ViInt32` or `ViReal64` attribute with a coerced range table. It also does so if the attribute does not have a coerce callback and you install a range table callback for it. If you want to specify your own callback function for the attribute but you want to call the default coerce callback within your function, you can call `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 default callback when you create a ViBoolean attribute.

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

If you do not want the IVI engine to invoke a coerce callback for the attribute, pass VI_NULL for the **coerceCallback** parameter.

Parameters

Input

Name	Type	Description
vi	ViSession	The ViSession handle that you obtain from Ivi_SpecificDriverNew. The handle identifies a particular IVI session.
attributeID	Depends on the data type of the attribute	The ID of the attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.
coerceCallback	Depends on the data type of the attribute	The coerce callback function you want the IVI engine to invoke when you attempt to set the attribute to a new value. Can be VI_NULL.

Return Value

Name	Type	Description
status	ViStatus	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

Except for ViString attributes, the function you specify for the **coerceCallback** parameter must have one of the following prototypes, based on data type:

```
ViStatus _VI_FUNC ViInt32CoerceCallback(ViSession vi,
                        ViConstString channelName,
                        ViAttr attributeId, ViAddr value,
                        ViInt32 *coercedValue);
```

```
ViStatus _VI_FUNC ViReal64CoerceCallback(ViSession vi,
                          ViConstString channelName,
                          ViAttr attributeId, ViAddr value,
                          ViReal64 *coercedValue);
ViStatus _VI_FUNC ViBooleanCoerceCallback(ViSession vi,
                          ViConstString channelName,
                          ViAttr attributeId, ViAddr value,
                          ViBoolean *coercedValue);
ViStatus _VI_FUNC ViSessionCoerceCallback(ViSession vi,
                          ViConstString channelName,
                          ViAttr attributeId, ViAddr value,
                          ViSession *coercedValue);
ViStatus _VI_FUNC ViAddrCoerceCallback(ViSession vi,
                          ViConstString channelName,
                          ViAttr attributeId, ViAddr value,
                          ViAddr *coercedValue);
```

For a `ViString` attribute, the function you specify in the **coerceCallback** parameter must have one the following prototype:

```
ViStatus _VI_FUNC ViStringCoerceCallback(ViSession vi,
                          ViConstString channelName,
                          ViAttr attributeId,
                          const ViConstString value);
```

Unlike the coerce callback functions for the other data types, the coerce callback for a `ViString` attribute does not report the coerced value to the caller through the last parameter. Instead, it reports the coerced value by passing it to `Ivi_SetValInStringCallback`.

Note *If you want to use the* **Edit Instrument Attributes** *command to develop your instrument driver source code, retain the parameter names as shown in the prototype for the callback.*

See Also

`Ivi_AddAttributeViInt32`, `Ivi_AddAttributeViReal64`, `Ivi_AddAttributeViBoolean`, `Ivi_DefaultCoerceCallbackViInt32`, `Ivi_DefaultCoerceCallbackViReal64`, `Ivi_DefaultCoerceCallbackViBoolean`, `Ivi_ValidateRangeTable`, `Ivi_GetRangeTableNumEntries`, `Ivi_GetViInt32EntryFromValue`, `Ivi_GetViReal64EntryFromValue`, `Ivi_SetValInStringCallback`

Ivi_SetAttrCompareCallbackViInt32
Ivi_SetAttrCompareCallbackViReal64
Ivi_SetAttrCompareCallbackViString
Ivi_SetAttrCompareCallbackViBoolean
Ivi_SetAttrCompareCallbackViSession
Ivi_SetAttrCompareCallbackViAddr

```
ViStatus status = Ivi_SetAttrCompareCallbackViInt32 (ViSession vi,
                          ViAttr attributeID,
                          CompareAttrViInt32_CallbackPtr compareCallback);
ViStatus status = Ivi_SetAttrCompareCallbackViReal64 (ViSession vi,
                          ViAttr attributeID,
                          CompareAttrViReal64_CallbackPtr
                          compareCallback);
ViStatus status = Ivi_SetAttrCompareCallbackViString (ViSession vi,
                          ViAttr attributeID,
                          CompareAttrViString_CallbackPtr
                          compareCallback);
ViStatus status = Ivi_SetAttrCompareCallbackViBoolean (ViSession vi,
                          ViAttr attributeID,
                          CompareAttrViBoolean_CallbackPtr
                          compareCallback);
ViStatus status = Ivi_SetAttrCompareCallbackViSession (ViSession vi,
                          ViAttr attributeID,
                          CompareAttrViSession_CallbackPtr
                          compareCallback);
ViStatus status = Ivi_SetAttrCompareCallbackViAddr (ViSession vi,
                          ViAttr attributeID,
                          CompareAttrViAddr_CallbackPtr compareCallback);
```

Purpose

Sets the compare callback function for an attribute. The IVI engine invokes the compare callback when comparing cache values it obtains from the instrument against new values you set the attribute to. If the compare callback determines that the two values are equal, the IVI engine does not call the write callback for the attribute.

A compare callback is useful when the instrument can return several values that you consider to have the same meaning, and you do not want to coerce the instrument value in your read callback.

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. For more information, refer to the *Comparison Precision* section in Chapter 2, *IVI Architecture Overview*.

If you pass VI_NULL for the **compareCallback** parameter, the IVI engine makes the comparison based on strict equality. For all attributes other than ViReal64 attributes, the IVI engine makes the comparison based on strict equality by default.

Parameters

Input

Name	Type	Description
vi	ViSession	The ViSession handle that you obtain from Ivi_SpecificDriverNew. The handle identifies a particular IVI session.
attributeID	depends on the data type of the attribute	The ID of the attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.
compareCallback	depends on the data type of the attribute	The compare callback function you want the IVI engine to invoke to compare a cache value you obtained from the instrument against a new value you want to set the attribute to. Can be VI_NULL.

Return Value

Name	Type	Description
status	ViStatus	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

The function you specify for the **compareCallback** parameter must have one of the following prototypes, based on data type:

```
ViStatus _VI_FUNC ViInt32CompareCallback(ViSession vi,
                           ViConstString channelName,
                           ViAttr attributeId,
                           ViInt32 coercedNewValue,
                           ViInt32 cacheValue ViInt32 *result);

ViStatus _VI_FUNC ViReal64CompareCallback(ViSession vi,
                           ViConstString channelName,
                           ViAttr attributeId,
                           ViReal64 coercedNewValue,
                           ViReal64 cacheValue, ViInt32 *result);

ViStatus _VI_FUNC ViStringCompareCallback(ViSession vi,
                           ViConstString channelName,
                           ViAttr attributeId,
                           ViConstString coercedNewValue,
                           ViConstString cacheValue,
                           ViInt32 *result);

ViStatus _VI_FUNC ViBooleanCompareCallback(ViSession vi,
                           ViConstString channelName,
                           ViAttr attributeId,
                           ViBoolean coercedNewValue,
                           ViBoolean cacheValue, ViInt32 *result);

ViStatus _VI_FUNC ViSessionCompareCallback(ViSession vi,
                           ViConstString channelName,
                           ViAttr attributeId,
                           ViSession coercedNewValue,
                           ViSession cacheValue, ViInt32 *result);

ViStatus _VI_FUNC ViAddrCompareCallback(ViSession vi,
                           ViConstString channelName,
                           ViAttr attributeId,
                           ViAddr coercedNewValue, ViAddr cacheValue,
                           ViInt32 *result);
```

The callback must set ***result** to a zero if **coercedNewValue** and **cacheValue** are equal. Otherwise, it must set ***result** to a nonzero value.

Note *If you want to use the* **Edit Instrument Attributes** *command to develop your instrument driver source code, retain the parameter names as shown in the prototype for the callback.*

See Also

Ivi_AddAttributeViReal64, Ivi_DefaultCompareCallbackViReal64

Ivi_SetAttrComparePrecision

```
ViStatus status = Ivi_SetAttrComparePrecision (ViSession vi,
                           ViAttr attributeID, ViInt32 comparePrecision);
```

Purpose

Changes the degree of decimal precision the default IVI compare callback uses for a specific attribute.

`Ivi_SetAttrComparePrecision` is useful only for `ViReal64` attributes. You set the initial comparison precision level for an attribute as a parameter to the `Ivi_AddAttributeViReal64` function.

Unless you call `Ivi_SetAttrCompareCallbackViReal64` to install your own compare callback function, the IVI engine invokes the default compare callback when comparing cache values it obtains from the instrument against new values you set the attribute to. If the values are equal within the degree of precision you specify, the IVI engine does not call the write callback for the attribute.

The IVI engine uses this method instead of strict equality because of differences between computer and instrument floating-point representations.

If the compare callback for the attribute is currently `VI_NULL`, this function installs the default IVI compare callback.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
attributeID	`ViAttr`	The ID of the attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.
comparePrecision	`ViInt32`	The degree of decimal precision the default IVI compare callback function uses for the attribute. Refer to the *Comparison Precision* section in Chapter 2, *IVI Architecture Overview*. Valid Range: 0, 1 to 14. If you pass 0, the function uses 14.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_GetAttrComparePrecision`, `Ivi_AddAttributeViReal64`,
`Ivi_DefaultCompareCallbackViReal64`

Ivi_SetAttributeFlags

```
ViStatus status = Ivi_SetAttributeFlags (ViSession vi, ViAttr attributeID,
                                         IviAttrFlags flags);
```

Purpose

Sets the flags of an attribute to new values. `Ivi_SetAttributeFlags` always sets all of the flags. If you want to change one flag, use `Ivi_GetAttributeFlags` to obtain the current values of all the flags, modify the bit for the flag you want to change, and then call `Ivi_SetAttributeFlags`.

Refer to the *Attribute Flags* section in Chapter 2, *IVI Architecture Overview*, for more information.

Note ***You cannot modify the value of the*** `IVI_VAL_MULTI_CHANNEL` ***flag.***

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
attributeID	`ViAttr`	The ID of the attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.
flags	`IviAttrFlags`	The new values of the flags for the attribute. Refer to the *Attribute Flags* section in Chapter 2, *IVI Architecture Overview*.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Example

The following code shows how to change the IVI_VAL_NEVER_CACHE flag from 1 to 0.

```
IviAttrFlags oldFlags, newFlags;

Ivi_GetAttributeFlags (vi, attributeID, &oldFlags);
newFlags = oldFlags & ~IVI_VAL_NEVER_CACHE;
Ivi_SetAttributeFlags (vi, attributeID, newFlags);
```

See Also

Ivi_GetAttributeFlags

Ivi_SetAttributeViInt32
Ivi_SetAttributeViReal64
Ivi_SetAttributeViString
Ivi_SetAttributeViBoolean
Ivi_SetAttributeViSession
Ivi_SetAttributeViAddr

```
ViStatus status = Ivi_SetAttributeViInt32 (ViSession vi,
                           ViConstString channel, ViAttr attributeID,
                           ViInt32 optionFlags, ViInt32 attributeValue);
ViStatus status = Ivi_SetAttributeViReal64 (ViSession vi,
                           ViConstString channel, ViAttr attributeID,
                           ViInt32 optionFlags, ViReal64 attributeValue);
ViStatus status = Ivi_SetAttributeViBoolean (ViSession vi,
                           ViConstString channel, ViAttr attributeID,
                           ViInt32 optionFlags, ViBoolean attributeValue);
ViStatus status = Ivi_SetAttributeViString (ViSession vi,
                           ViConstString channel, ViAttr attributeID,
                           ViInt32 optionFlags,
                           ViConstString attributeValue);
ViStatus status = Ivi_SetAttributeViSession (ViSession vi,
                           ViConstString channel, ViAttr attributeID,
                           ViInt32 optionFlags, ViSession attributeValue);
ViStatus status = Ivi_SetAttributeViAddr (ViSession vi,
                           ViConstString channel, ViAttr attributeID,
                           ViInt32 optionFlags, ViAddr attributeValue);
```

Purpose

Sets an attribute to a new value.

Depending on the configuration of the attribute, the function performs the following actions:

1. Checks whether the attribute is writable. If not, the function returns an error.
2. Validates the value you specify if `IVI_ATTR_RANGE_CHECK` is enabled for the session. If you provide a check callback, the function invokes the callback to validate the value. If you do not provide a check callback but the data type of the attribute is `ViInt32` or `ViReal64` and you provide a range table or a range table callback, the function invokes the default IVI check callback to validate the value. If the value is invalid, the function returns an error.

3. Coerces the value you specify into a canonical value the instrument accepts. If you provide a coerce callback, the function invokes the callback to coerce the value. If you do not provide a coerce callback, the data type of the attribute is `ViInt32` or ViReal, and you provide a coerced range table directly or through a range table callback, the function invokes the default IVI coerce callback for the data type. The IVI engine automatically installs the default IVI coerce callback for `ViBoolean` attributes. Generally, `ViSession`, `ViAddr`, and `ViString` attributes do not have coerce callbacks.
4. If the `IVI_ATTR_DEFER_UPDATE` attribute is enabled for the session, the `IVI_VAL_NO_DEFERRED_UPDATE` flag is not set for the attribute, and you do not set the `IVI_VAL_SET_CACHE_ONLY` flag in the **optionFlags** parameter, the function posts a deferred update. It also keeps a record of the deferred update value with the attribute, and skips the remainder of these actions. You can perform deferred updates at a later time by calling `Ivi_Update`.
5. Compares the new value with the current cache value for the attribute to see if they are equal. The method it uses depends on the source of the cache value. If the cache contains a value you previously sent to the instrument, the function compares the two values using strict equality. If the cache contains a value you obtained from the instrument, the function invokes the compare callback. If you provide no compare callback and the data type of the attribute is not `ViReal64`, the function makes the comparison based on strict equality. If the data type is `ViReal64`, the function invokes the default IVI compare callback, which uses the comparison precision you specify when you create the attribute.
6. If the cache value is valid and equal to the new value, the function returns `VI_SUCCESS` and skips the remainder of these actions.
7. If you set the `IVI_VAL_SET_CACHE_ONLY` bit in the **optionFlags** parameter, or if the `IVI_ATTR_SIMULATE` attribute is enabled and the `IVI_VAL_USE_CALLBACKS_FOR_SIMULATION` flag for the attribute is 0, the function does not call the write callback, the operation complete callback, or the check status callback. It merely updates the cache value of the attribute.
8. If the new value is not equal to the cache value or the cache value is invalid, the function invokes the write callback for the attribute. The write callback might perform I/O to send the value to the instrument. The IVI engine stores the new value in the cache. If the function coerces the value, the function caches the coerced value rather than the value you pass. For `ViString` attributes, the function allocates a copy of the string to keep in the cache.
9. If the `IVI_VAL_WAIT_FOR_OPC_AFTER_WRITES` flag is set for the attribute, the function invokes the operation complete (OPC) callback you provide for the session.
10. If you set the `IVI_VAL_DIRECT_USER_CALL` bit in the **optionFlags** parameter, the `IVI_ATTR_QUERY_INSTR_STATUS` attribute is enabled, and the `IVI_VAL_DONT_CHECK_STATUS` flag for the attribute is 0, the function invokes the check status callback you provide for the session.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
channel	`ViConstString`	If the attribute is channel-based, pass a channel string or virtual channel name. Otherwise, pass `VI_NULL` or an empty string.
attributeID	`ViAttr`	The ID of the attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.
optionFlags	`ViInt32`	Use this parameter to request special behavior. In most cases, you pass 0. Refer to the *Parameter Discussion* section.
attributeValue	Depends on the data type of the attribute	The value to which you want to set the attribute.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Option Flags

You can specify individual bit flags in the **optionFlags** parameter to request special behavior. If you want to specify multiple flags, you can bitwise OR them together. Normally you pass 0 for **optionFlags**.

Specify the `IVI_VAL_DIRECT_USER_CALL` flag only when calling this function to implement one of the *`Prefix`*`_GetAttribute` functions that your instrument driver exports to the user. When you pass `IVI_VAL_DIRECT_USER_CALL`, the function returns an error if the `IVI_VAL_NOT_WRITABLE` or `IVI_VAL_NOT_USER_WRITABLE` flag for the attribute is set. Also, the function invokes the check status callback when you pass

IVI_VAL_DIRECT_USER_CALL but only if IVI_ATTR_QUERY_INSTR_STATUS is enabled for the session and the IVI_VAL_DONT_CHECK_STATUS flag for the attribute is 0.

Specify the IVI_VAL_SET_CACHE_ONLY flag only when you want to set the value in the attribute cache without invoking the write callback for the attribute. This can be useful if one instrument I/O command sets multiple attributes in the instrument. In the write callback function that performs the instrument I/O, after the instrument I/O succeeds, call an Ivi_SetAttribute function for each of the other attributes, and set the IVI_VAL_SET_CACHE_ONLY bit set to 1 in the **optionFlags** parameter.

Specify the IVI_VAL_DONT_MARK_AS_SET_BY_USER flag only when you want to set an attribute value even though the user has not requested you do so directly through a *Prefix*_SetAttribute function call or indirectly through a high-level function that sets multiple attributes. This case occurs very rarely. It affects interchangeability checking in class drivers. To pass interchangeability checking, either all attributes in an extension group must be marked as "set by user" or none of them must be marked as "set by user".

See Also

Ivi_Update

Ivi_SetAttrRangeTableCallback

```
ViStatus status = Ivi_SetAttrRangeTableCallback (ViSession vi,
                      ViAttr attributeID,
                      RangeTable_CallbackPtr rangeTableCallback);
```

Purpose

Sets the callback that the IVI engine invokes to obtain a pointer to the range table for an attribute. Although any attribute can have a range table, normally range tables are useful only for `ViInt32` or `ViReal64` attributes.

When you create a `ViInt32` or `ViReal64` attribute, you can specify a single range table for the IVI engine to use to validate values for the attribute. Normally, one range table is sufficient. If this is the case, you do not need a range table callback function. By default, the range table callback for each attribute is `VI_NULL`.

Sometimes, however, you might want to use different range tables depending on the current settings of other attributes. In that case, call `Ivi_SetAttrRangeTableCallback` to install a callback for the IVI engine to invoke. In the callback, you determine which range table you want to use, and you return a pointer to it.

When you specify a non-NULL range table callback for a `ViInt32` or `ViReal64` attribute, the IVI engine automatically installs its default check and coerce callbacks if these callback are currently `VI_NULL`.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
attributeID	`ViAttr`	The ID of the attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.
rangeTableCallback	`RangeTable_CallbackPtr`	The range table callback function you want the IVI engine to invoke to obtain a range table for the attribute. Can be `VI_NULL`. Refer to the *Parameter Discussion* section.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

The function you specify for the **rangeTableCallback** parameter must have the following prototype:

```
ViStatus _VI_FUNC Callback(ViSession vi, ViConstString channelName,
                           ViAttr attributeId,
                           IviRangeTablePtr *rangeTablePtr);
```

If you do not want to use a range table callback function, pass `VI_NULL`.

See Also

`Ivi_RangeTableNew`, `Ivi_GetAttrRangeTable`, `Ivi_GetStoredRangeTablePtr`, `Ivi_SetStoredRangeTablePtr`, `Ivi_DefaultCheckCallbackViInt32`, `Ivi_DefaultCheckCallbackViReal64`, `Ivi_DefaultCoerceCallbackViInt32`, `Ivi_DefaultCoerceCallbackViReal64`, `Ivi_AddAttributeViInt32`, `Ivi_AddAttributeViReal64`

Ivi_SetAttrReadCallbackViInt32
Ivi_SetAttrReadCallbackViReal64
Ivi_SetAttrReadCallbackViString
Ivi_SetAttrReadCallbackViBoolean
Ivi_SetAttrReadCallbackViSession
Ivi_SetAttrReadCallbackViAddr

```
ViStatus status = Ivi_SetAttrReadCallbackViInt32 (ViSession vi,
                           ViAttr attributeID,
                           ReadAttrViInt32_CallbackPtr readCallback);
ViStatus status = Ivi_SetAttrReadCallbackViReal64 (ViSession vi,
                           ViAttr attributeID,
                           ReadAttrViReal64_CallbackPtr readCallback);
ViStatus status = Ivi_SetAttrReadCallbackViString (ViSession vi,
                           ViAttr attributeID,
                           ReadAttrViString_CallbackPtr readCallback);
ViStatus status = Ivi_SetAttrReadCallbackViBoolean (ViSession vi,
                           ViAttr attributeID,
                           ReadAttrViBoolean_CallbackPtr readCallback);
ViStatus status = Ivi_SetAttrReadCallbackViSession (ViSession vi,
                           ViAttr attributeID,
                           ReadAttrViSession_CallbackPtr readCallback);
ViStatus status = Ivi_SetAttrReadCallbackViAddr (ViSession vi,
                           ViAttr attributeID,
                           ReadAttrViAddr_CallbackPtr readCallback);
```

Purpose

Sets the read callback function for a `ViAddr` attribute. The IVI engine invokes the read callback function when you request the current value of the attribute and the cache value is invalid.

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

Normally, you specify the read callback function when you create the attribute with `Ivi_AddAttributeViAddr`. Use this attribute to assign a different read callback function to the attribute.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
attributeID	Depends on the data type of the attribute	The ID of the attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.
readCallback	Depends on the data type of the attribute	The read callback function you want the IVI engine to invoke when you request the current value of the attribute. Can be `VI_NULL`.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

The function you specify for the **readCallback** parameter must be in the instrument driver source code. Except for `ViString` attributes, the function must have one of the following prototypes, based on data type:

```
ViStatus _VI_FUNC ViInt32ReadCallback(ViSession vi, ViSession io,
                          ViConstString channelName,
                          ViAttr attributeId, ViInt32 *value);
```

```
ViStatus _VI_FUNC ViReal64ReadCallback(ViSession vi, ViSession io,
                          ViConstString channelName,
                          ViAttr attributeId, ViReal64 *value);
```

```
ViStatus _VI_FUNC ViBooleanReadCallback(ViSession vi, ViSession io,
                          ViConstString channelName,
                          ViAttr attributeId, ViBoolean *value);
```

```
ViStatus _VI_FUNC ViSessionCallback(ViSession vi, ViSession io,
                          ViConstString channelName,
                          ViAttr attributeId, ViSession *value);
ViStatus _VI_FUNC ViAddrReadCallback(ViSession vi, ViSession io,
                          ViConstString channelName,
                          ViAttr attributeId, ViAddr *value);
```

Upon entry to the callback, ***value** contains the cache value. Upon exit from the callback, ***value** must contain the actual current value.

For a `ViString` attribute, the function you specify in the **readCallback** parameter must have the following prototype:

```
ViStatus _VI_FUNC ViStringReadCallback(ViSession vi, ViSession io,
                          ViConstString channelName,
                          ViAttr attributeId,
                          const ViConstString cacheValue);
```

Unlike the read callback functions for the other data types, the read callback for a `ViString` attribute does not report the current value to the caller through the last parameter. Instead, it reports the current value by passing it to `Ivi_SetValInStringCallback`.

Note *If you want to use the* **Edit Instrument Attributes** *command to develop your instrument driver source code, retain the parameter names as shown in the prototype for the callback.*

See Also

`Ivi_AddAttributeViInt32`, `Ivi_AddAttributeViReal64`, `Ivi_AddAttributeViString`, `Ivi_AddAttributeViBoolean`, `Ivi_AddAttributeViSession`, `Ivi_AddAttributeViAddr`, `Ivi_SetValInStringCallback`

Ivi_SetAttrWriteCallbackViInt32
Ivi_SetAttrWriteCallbackViReal64
Ivi_SetAttrWriteCallbackViString
Ivi_SetAttrWriteCallbackViBoolean
Ivi_SetAttrWriteCallbackViSession
Ivi_SetAttrWriteCallbackViAddr

```
ViStatus status = Ivi_SetAttrWriteCallbackViInt32 (ViSession vi,
                          ViAttr attributeID,
                          WriteAttrViInt32_CallbackPtr writeCallback);
ViStatus status = Ivi_SetAttrWriteCallbackViReal64 (ViSession vi,
                          ViAttr attributeID,
                          WriteAttrViReal64_CallbackPtr writeCallback);
ViStatus status = Ivi_SetAttrWriteCallbackViString (ViSession vi,
                          ViAttr attributeID,
                          WriteAttrViString_CallbackPtr writeCallback);
ViStatus status = Ivi_SetAttrWriteCallbackViBoolean (ViSession vi,
                          ViAttr attributeID,
                          WriteAttrViBoolean_CallbackPtr writeCallback);
ViStatus status = Ivi_SetAttrWriteCallbackViSession (ViSession vi,
                          ViAttr attributeID,
                          WriteAttrViSession_CallbackPtr writeCallback);
ViStatus status = Ivi_SetAttrWriteCallbackViAddr (ViSession vi,
                          ViAttr attributeID,
                          WriteAttrViAddr_CallbackPtr writeCallback);
```

Purpose

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

Normally, you specify the write callback function when you create the attribute with `Ivi_AddAttributeViAddr`. Use this attribute to assign a different write callback function to the attribute.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
attributeID	Depends on the data type of the attribute	The ID of the attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.
writeCallback	Depends on the data type of the attribute	The write callback function you want the IVI engine to invoke when you set the attribute to a new value. Can be `VI_NULL`.

Return Value

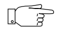
Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

The function you specify for the writeCallback parameter must be in the instrument driver source code. The function must have one of the following prototypes, based on data type:

```
ViStatus _VI_FUNC ViInt32WriteCallback(ViSession vi, ViSession io,
                         ViConstString channelName,
                         ViAttr attributeId, ViInt32 value);

ViStatus _VI_FUNC ViReal64WriteCallback(ViSession vi, ViSession io,
                         ViConstString channelName,
                         ViAttr attributeId, ViReal64 value);

ViStatus _VI_FUNC ViStringWriteCallback(ViSession vi, ViSession io,
                         ViConstString channelName,
                         ViAttr attributeId, ViConstString value);

ViStatus _VI_FUNC ViBooleanWriteCallback(ViSession vi, ViSession io,
                         ViConstString channelName,
                         ViAttr attributeId, ViBoolean value);
```

```
ViStatus _VI_FUNC ViSessionWriteCallback(ViSession vi, ViSession io,
                           ViConstString channelName,
                           ViAttr attributeId, ViSession value);
ViStatus _VI_FUNC ViAddrWriteCallback(ViSession vi, ViSession io,
                           ViConstString channelName,
                           ViAttr attributeId, ViAddr value);
```

 **Note** ***If you want to use the* Edit Instrument Attributes *command to develop your instrument driver source code, retain the parameter names as shown in the prototype for the callback.***

See Also

Ivi_AddAttributeViInt32, Ivi_AddAttributeViReal64, Ivi_AddAttributeViString, Ivi_AddAttributeViBoolean, Ivi_AddAttributeViSession, Ivi_AddAttributeViAddr

Ivi_SetErrorInfo

```
ViStatus status = Ivi_SetErrorInfo (ViSession vi, ViBoolean overwrite,
                           ViStatus primaryError, ViStatus secondaryError,
                           ViConstString elaboration);
```

Purpose

Sets the error information for the current execution thread and the IVI session you specify. If you pass `VI_NULL` for the **vi** parameter, `Ivi_SetErrorInfo` sets the error information only for the current execution thread.

Normally, the error information describes the first error that occurred since the user last called *`Prefix`*`_GetErrorInfo` or *`Prefix`*`_ClearErrorInfo`.

Refer to the *Error Reporting* section earlier in this chapter for details on IVI error information. Also refer to the *Error Macros* section earlier in this chapter for details on the IVI error macros, some of which use `Ivi_SetErrorInfo`.

Parameters

Input

Name	Type	Description
vi	`ViSession`	To set the error information for a particular IVI session and the current thread, pass the `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. To set only the error information for the current thread, pass `VI_NULL`.
overwrite	`ViBoolean`	Specifies whether you want the error information you pass to `Ivi_SetErrorInfo` to supersede the existing information regardless of the contents of the existing error information. Pass `VI_TRUE` to always overwrite the existing error information. Refer to the *Parameter Discussion* section for a description of what `Ivi_SetErrorInfo` does when you pass `VI_FALSE`.

Name	Type	Description
primaryError	`ViStatus`	A status code describing the primary error condition. Use `VI_SUCCESS` (0) to indicate no error or warning. Use a positive value to indicate a warning. Use a negative value to indicate an error.
secondaryError	`ViStatus`	A status code that further describes the error or warning condition. If you have no further description, pass `VI_SUCCESS` (0).
elaboration	`ViConstString`	An elaboration string that further describes the error or warning condition. The IVI engine stores the entire string for the session you specify, but it retains only `IVI_MAX_MESSAGE_BUF_SIZE-1` (255) characters for the current execution thread. If you have no further description, pass `VI_NULL` or an empty string.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

Normally, users expect the error information to describe the first error that occurred since their last call to `*Prefix*_GetErrorInfo` or `*Prefix*_ClearErrorInfo`. So avoid passing `VI_TRUE` for the overwrite parameter.

When you pass `VI_FALSE` for the overwrite parameter, `Ivi_SetErrorInfo` uses the following logic to determine whether to overwrite the existing error information:

1. Overwrite the primary error code if one of the following two conditions is true:
    - The existing primary code is `VI_SUCCESS`.
    - The existing primary code is a positive warning code and the primary error code you specify is a negative error code.

2. Overwrite the secondary error code if one of the following two conditions is true:
   - The function overwrites the old primary error code with a different value in Step 1.
   - The existing secondary code is `VI_SUCCESS` and the primary code you specify is either `VI_SUCCESS` or equal to the old primary error code.
3. Overwrite the elaboration string if one of the following two conditions is true:
   - The function overwrites the old primary error code with a different value in Step 1.
   - The existing elaboration string is empty and the primary code you specify is either `VI_SUCCESS` or equal to the old primary error code.

This behavior allows you to make multiple calls to `Ivi_SetErrorInfo` at different levels in your instrument driver source code without the risk of losing important error information. For instance, if you set the primary code to a negative error value, subsequent calls to `Ivi_SetErrorInfo` do not change the value. Consequently, `Ivi_GetErrorInfo` always returns the first error that you reported.

At the same time, you can make subsequent calls to `Ivi_SetErrorInfo` to add further information. If your first call to `Ivi_SetErrorInfo` specifies a negative primary error code, a zero secondary error code, and no elaboration string, you can later add a secondary error code and an elaboration string by calling `Ivi_SetErrorInfo` with the same primary error code.

See Also

`Ivi_GetErrorInfo`, `Ivi_ClearErrorInfo`, `Ivi_GetErrorMessage`, `Ivi_GetSpecificDriverStatusDesc`

Ivi_SetIviIniDir

```
ViStatus status = Ivi_SetIviIniDir (ViConstString directoryPath);
```

Purpose

Sets the directory where the IVI engine looks for the `ivi.ini` configuration file.

If you not call `Ivi_SetIviIniDir`, or if you pass `VI_NULL` or the empty string for **directoryPath**, the IVI engine looks for the `ivi.ini` file in the `NIivi` directory under the VXI*plug&play* framework directory.

Refer to the *Configuration Entries* section in Chapter 2, *IVI Architecture Overview*, for information on `ivi.ini`.

Parameters

Input

Name	Type	Description
directoryPath	`ViConstString`	The pathname of the directory where you want the IVI engine to search for the `ivi.ini` configuration file. Pass `VI_NULL` or the empty string to restore the default path.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

If you specify a literal string for the **directoryPath** parameter under Windows, be sure to use double backslashes to represent one backslash in the pathname.

See Also

`Ivi_GetIviIniDir`, `Ivi_GetLogicalNamesList`, `Ivi_GetNthLogicalName`

Ivi_SetNeedToCheckStatus

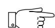
```
ViStatus Ivi_SetNeedToCheckStatus (ViSession vi,
                                   ViBoolean needToCheckStatus);
```

Purpose

Allows an instrument driver to indicate whether it is necessary to check the status of the instrument.

The IVI engine maintains an internal needToCheckStatus variable for each session indicating whether it is necessary to check the status of the instrument. When you create a new session, the initial value of the variable is VI_TRUE. The IVI engine sets the needToCheckStatus variable to VI_TRUE when it invokes the read or write callback for an attribute for which the IVI_VAL_DONT_CHECK_STATUS flag is 0. The Ivi_WriteInstrData and Ivi_WriteFromFile functions also set the variable to VI_TRUE. The IVI engine sets the variable to VI_FALSE after it invokes the check status callback successfully.

The Ivi_SetNeedToCheckStatus function allows an instrument driver to set the state of the internal needToCheckStatus variable. A driver typically sets the variable to VI_TRUE before it attempts direct instrument I/O. It sets it to VI_FALSE after it calls the check status callback successfully.



Note ***Do not call*** Ivi_SetNeedToCheckStatus ***unless you have already locked the session.***

Parameter List

Input

Name	Type	Description
vi	ViSession	The ViSession handle that you obtain from Ivi_SpecificDriverNew. The handle identifies a particular IVI session.
needToCheckStatus	ViBoolean	Refer to the *Parameter Discussion* section.

Return Value

Name	Type	Description
status	ViStatus	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

Before you attempt to interact with the instrument directly, call `Ivi_SetNeedToCheckStatus` with `VI_TRUE` for the **needToCheckStatus** parameter.

After you invoke the check status callback successfully, call `Ivi_SetNeedToCheckStatus` with `VI_FALSE` for the **needToCheckStatus** parameter.

See Also

`Ivi_NeedToCheckStatus`

Ivi_SetRangeTableEnd

```
ViStatus status = Ivi_SetRangeTableEnd (IviRangeTablePtr rangeTable,
                                        ViInt32 index);
```

Purpose

Sets the termination entry for a dynamic range table you create with `Ivi_RangeTableNew`.

`Ivi_RangeTableNew` automatically sets the last entry you create to be the termination entry. For example, if you specify 10 entries, `Ivi_RangeTableNew` marks the entry at index 9 to be the termination entry. Use `Ivi_SetRangeTableEnd` function if you want to move the termination entry to a lower index.

Parameters

Input

Name	Type	Description
rangeTable	`IviRangeTable Ptr`	The range table pointer you obtain from `Ivi_RangeTableNew`.
index	`ViInt32`	The 0-based index of the entry to become the termination entry.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_RangeTableNew`, `Ivi_SetRangeTableEntry`, `Ivi_RangeTableFree`, `Ivi_GetRangeTableNumEntries`

Ivi_SetRangeTableEntry

```
ViStatus status = Ivi_SetRangeTableEntry (IviRangeTablePtr rangeTable,
                          ViInt32 index, ViReal64 discreteOrMinValue,
                          ViReal64 maxValue, ViReal64 coercedValue,
                          ViConstString cmdString, ViInt32 cmdValue);
```

Purpose

Configures the values in a range table entry. To set the terminating entry, call `Ivi_SetRangeTableEnd`.

Parameters

Input

Name	Type	Description
rangeTable	`IviRangeTablePtr`	The range table pointer you obtain from `Ivi_RangeTableNew`.
index	`ViInt32`	The 0-based index of the entry to be the termination entry. Refer to the *Parameter Discussion* section.
discreteOrMinValue	`ViReal64`	The value you want to assign to the `discreteOrMinValue` field of the entry. Refer to the *Parameter Discussion* section.
maxValue	`ViReal64`	The value you want to assign to the `maxValue` field of the entry. Refer to the *Parameter Discussion* section.
coercedValue	`ViReal64`	The value you want to assign to the `coercedValue` field of the entry. Refer to the *Parameter Discussion* section.
cmdString	`ViConstString`	The string you want to assign to the `cmdString` field of the range table entry. Refer to the *Parameter Discussion* section.
cmdValue	`ViInt32`	The integer value you want to assign to the `cmdValue` field of the range table entry. Refer to the *Parameter Discussion* section.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

The index parameter can range from 0 to two less than the number of entries in the table. For example, if you specify 10 entries when you call `Ivi_RangeTableNew`, you can call `Ivi_SetRangeTableEntry` with index values from 0 through 8. The termination entry is at index 9, unless you place it at a lower index using `Ivi_SetRangeTableEnd`. If you call `Ivi_SetRangeTableEnd` to change the location of the termination entry, the **index** parameter to `Ivi_SetRangeTableEntry` must be less than the index of the termination entry.

If the range table type is `IVI_VAL_DISCRETE`, the **discreteOrMinValue** parameter holds the discrete value. If the range table type is `IVI_VAL_RANGED` or `IVI_VAL_COERCED`, the **discreteOrMinValue** parameter holds the minimum value. The **maxValue** parameter holds the maximum value and is valid only if the table type is `IVI_VAL_RANGED` or `IVI_VAL_COERCED`. The **coercedValue** parameter is valid only if the table type is `IVI_VAL_COERCED`. Refer to the *Range Tables* section in Chapter 2, *IVI Architecture Overview* for more information on these parameters.

The `cmdString` field in the range table entry is optional. You can use it to hold the command string that an attribute write callback sends to the instrument when you set the attribute to the value or range of values that the entry defines. If you do not want to specify a command string, pass `VI_NULL`.

If you want to dynamically allocate the command string, use `Ivi_Alloc`. Pass the pointer you obtain from `Ivi_Alloc` as the **cmdString** parameter to `Ivi_SetRangeTableEntry`. If you call `Ivi_RangeTableFree` to deallocate the range table, you can request that it call `Ivi_Free` on each non-NULL command string in the table.

The **cmdValue** field in the range table entry is optional. You can use it to hold a value that an attribute write callback formats into an instrument command string when you set the attribute to the value or range of values that the entry defines. If you do not want to specify a command value, pass 0.

See Also

`Ivi_RangeTableNew`, `Ivi_SetRangeTableEnd`, `Ivi_RangeTableFree`, `Ivi_GetRangeTableNumEntries`

Ivi_SetStoredRangeTablePtr

```
ViStatus status = Ivi_SetStoredRangeTablePtr (ViSession vi,
                         ViAttr attributeID,
                         IviRangeTablePtr rangeTable);
```

Purpose

Sets range table for an attribute. You can specify a range table when you call `Ivi_AddAttributeViInt32` or `Ivi_AddAttributeViReal64` to create the attribute. Use this function to replace the original range table with a different one.

If data type of the attribute is `ViInt32` or `ViReal64`, **rangeTable** is non-NULL, and attribute does not have a check callback, `Ivi_SetStoredRangeTablePtr` installs the default IVI check callback for the attribute. If the type of the range table is `IVI_VAL_COERCED` and the attribute does not have a coerce callback, the function also installs the default IVI coerce callback for the attribute. The default callbacks use the range table to validate and coerce values for the attribute.

Refer to the *Range Tables* section in Chapter 2, *IVI Architecture Overview*, for more information on range tables.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
attributeID	`ViAttr`	The ID of the attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.
rangeTable	`IviRangeTablePtr`	Specify the address the range table that you want to use for the attribute. If you do not want a range table, pass `VI_NULL`.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_GetStoredRangeTablePtr`, `Ivi_AddAttributeViInt32`, `Ivi_AddAttributeViReal64`, `Ivi_SetAttrRangeTableCallback`, `Ivi_GetAttrRangeTable`, `Ivi_RangeTableNew`, `Ivi_ValidateRangeTable`, `Ivi_GetRangeTableNumEntries`, `Ivi_GetViInt32EntryFromValue`, `Ivi_GetViReal64EntryFromValue`

Ivi_SetValInStringCallback

```
ViStatus status = Ivi_SetValInStringCallback (ViSession vi,
                          ViAttr attributeID, ViConstString value);
```

Purpose

Sets the value of a ViString attribute in the context of the read or coerce callback function for the attribute.

All read functions for ViString attributes must use Ivi_SetValInStringcallback to report the new value of the attribute. All coerce functions for ViString attributes must use Ivi_SetValInStringcallback to report the coerced value.

Parameters

Input

Name	Type	Description
vi	ViSession	The ViSession handle that you obtain from Ivi_SpecificDriverNew. The handle identifies a particular IVI session.
attributeID	ViAttr	The attribute ID that the read or coerce callback for the attribute receives.
value	ViConstString	The value that you want to report from the read or coerce callback.

Return Value

Name	Type	Description
status	ViStatus	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

Ivi_AddAttributeViString, Ivi_SetAttrReadCallbackViString, Ivi_SetAttrCoerceCallbackViViString

Ivi_Simulating

```
ViBoolean simulating = Ivi_Simulating (ViSession vi);
```

Purpose

Returns the current value of the `IVI_ATTR_SIMULATE` attribute for the IVI session you specify. The attribute determines whether or not to simulate instrument driver I/O operations.

High-level functions in specific instrument drivers use `Ivi_Simulating`. `Ivi_Simulate` provides fast, convenient access to the `IVI_ATTR_SIMULATE` attribute because it performs no error checking and does not lock the session.

If you pass an invalid session handle, `Ivi_Simulating` returns `VI_FALSE`.

Note ***Do not call*** `Ivi_Simulating` ***unless you have already locked the session.***

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.

Return Value

Name	Type	Description
simulate	`ViBoolean`	1 = `VI_TRUE` - Simulate 0 = `VI_FALSE` - Do not simulate

See Also

`Ivi_UseSpecificSimulation`

Ivi_SpecificDriverNew

```
ViStatus status = Ivi_SpecificDriverNew (ViConstString specificDriverPrefix,
                          ViConstString optionsString, ViSession *vi);
```

Purpose

Creates a new IVI session to a specific instrument driver and returns a handle that you use in subsequent function calls to identify the session.

When you call `Ivi_SpecificDriverNew`, you can set the initial state of one or more of the following attributes:

```
IVI_ATTR_RANGE_CHECK
IVI_ATTR_QUERY_INSTR_STATUS
IVI_ATTR_CACHE
IVI_ATTR_SIMULATE
IVI_ATTR_RECORD_COERCIONS
IVI_ATTR_DRIVER_SETUP
```

`Ivi_SpecificDriverNew` creates a new session each time you invoke it. Although you can open more than one IVI session for the same resource, it is best not to do so. Instead, you can use the same session in multiple program threads. Use the functions `Ivi_LockSession` and `Ivi_UnlockSession` to protect sections of code that require exclusive access to the resource.

`Ivi_SpecificDriverNew` does not create a VISA session to any instrument resources. If you use VISA to communicate to the instrument, you must create a VISA session yourself and set the `IVI_ATTR_IO_SESSION` attribute to that value. If you do not use VISA, you can use the `IVI_ATTR_IO_SESSION` attribute to hold a handle to whatever communications resource you use.

Parameters

Input

Name	Type	Description
specificDriverPrefix	`ViConstString`	The prefix of the specific instrument driver. Refer to the *Parameter Discussion* section.
optionsString	`ViConstString`	A string in which you can initialize the values of certain IVI attributes for the session. Can be `VI_NULL`. Refer to the *Parameter Discussion* section.

Output

Name	Type	Description
vi	`ViSession`	Returns a `ViSession` handle that you use to identify the session in subsequent function calls.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

Every user-callable function in the instrument driver starts with **specificDriverPrefix** as part of its function name. For example, if the Fluke 45 driver has a user-callable function named `"FL45_init"`, then `"FL45"` is the prefix for that driver. An instrument prefix can contain up to eight characters.

Options String

You can use the **optionsString** parameter to set the initial value of certain IVI attributes for the session. The following table lists the attributes, their default initial values, and the name you use in this parameter to identify the attribute.

Table 11-4. optionsString Values

Name	Attribute Defined Constant	Default
RangeCheck	`IVI_ATTR_RANGE_CHECK`	`VI_TRUE`
QueryInstrStatus	`IVI_ATTR_QUERY_INSTR_STATUS`	`VI_TRUE`
Cache	`IVI_ATTR_CACHE`	`VI_TRUE`
Simulate	`IVI_ATTR_SIMULATE`	`VI_FALSE`
RecordCoercions	`IVI_ATTR_RECORD_COERCIONS`	`VI_FALSE`
DriverSetup	`IVI_ATTR_DRIVER_SETUP`	`""`

If you pass `VI_NULL` or an empty string for this parameter, the session uses the default values. You can override the default values by assigning a value explicitly in a string you pass for this parameter.

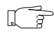
The format of an assignment is, *"Name=Value"* where *Name* is the first column in the table above, and *Value* is any one of the following.

- To set the attribute to VI_TRUE, use VI_TRUE, True, or 1.
- To set the attribute to VI_FALSE, use VI_FALSE, False, or 0.

Ivi_SpecificDriverNew interprets *Name* and *Value* in a case-insensitive manner.

To set multiple attributes, separate the assignments with commas.

You do not have to specify all of the attributes. If you do not specify one of the attributes, the IVI session uses the default value.



Note ***Normally, you use this function to implement*** *Prefix*_init ***and*** *Prefix*_InitWithOptions ***in the specific instrument driver. In*** *Prefix*_init***, you pass an empty string for the*** **optionsString*****. In*** Prefix_InitWithOptions***, you pass the*** **optionsString** ***parameter that the user passed to*** *Prefix*_InitWithOptions.

See Also

Ivi_Dispose, Ivi_LockSession, Ivi_UnlockSession, Ivi_ValidateSession, Ivi_IOSession, Ivi_RangeChecking, Ivi_QueryInstrStatus, Ivi_Simulating, Ivi_UseSpecificSimulation, Ivi_GetNextCoercionInfo

Ivi_Spying

```
ViBoolean spying = Ivi_Spying (ViSession vi);
```

Purpose

Returns the current value of the `IVI_ATTR_SPY` attribute for the IVI session you specify. The attribute determines whether class instrument drivers use the NI-Spy utility to record calls to class driver functions.

High-level functions in class instrument drivers use `Ivi_IOSession`. `Ivi_Spying` provides fast, convenient access to the `IVI_ATTR_SPY` attribute because it performs no error checking and does not lock the session.

If you pass an invalid session handle, `Ivi_Spying` returns `VI_FALSE`.

Note ***Do not call*** `Ivi_Spying` ***unless you have already locked the session.***

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.

Return Value

Name	Type	Description
spying	`ViBoolean`	1 = `VI_TRUE` - Record calls to class driver 0 = `VI_FALSE` - Do not record calls to class driver

Ivi_UndefClass

```
void Ivi_UndefClass (ViConstString className);
```

Purpose

Deletes a Class entry from the list of run-time configuration entries.

You cannot delete Class entries that the IVI engine reads from the `ivi.ini` configuration file.

Parameters

Input

Name	Type	Description
className	`ViConstString`	The name of a Class entry that you create using `Ivi_DefineClass`. You do not have to include the `"Class->"` prefix in the name.

Return Value

None

See Also

`Ivi_DefineClass`

Ivi_UndefDriver

```
void Ivi_UndefDriver (ViConstString driverName);
```

Purpose

Deletes a Driver entry from the list of run-time configuration entries.

You cannot delete Driver entries that the IVI engine reads from the `ivi.ini` configuration file.

Parameters

Input

Name	Type	Description
driverName	`ViConstString`	The name of a Driver entry that you create using `Ivi_DefineDriver`. You do not have to include the `"Driver->"` prefix in the name.

Return Value

None

See Also

`Ivi_DefineDriver`

Ivi_UndefHardware

```
void Ivi_UndefHardware (ViConstString hardwareName);
```

Purpose

Deletes a Hardware entry from the list of run-time configuration entries.

You cannot delete Hardware entries that the IVI engine reads from the `ivi.ini` configuration file.

Parameters

Input

Name	Type	Description
hardwareName	`ViConstString`	The name of a Hardware entry that you create using `Ivi_DefineHardware`. You do not have to include the `"Hardware->"` prefix in the name.

Return Value

None

See Also

`Ivi_DefineHardware`

Ivi_UndefLogicalName

```
void Ivi_UndefLogicalName (ViConstString logicalName);
```

Purpose

Deletes a logical name from the list of run-time configuration entries.

You cannot delete logical names that the IVI engine reads from the `ivi.ini` configuration file.

Parameters

Input

Name	Type	Description
logicalName	`ViConstString`	A logical name that you create using `Ivi_DefineLogicalName`.

Return Value

None

See Also

`Ivi_DefineLogicalName`

Ivi_UndefVInstr

```
void Ivi_UndefVInstr (ViConstString vInstrName);
```

Purpose

Deletes a `VInstr` entry from the list of run-time configuration entries.

You cannot delete `VInstr` entries that the IVI engine reads from the `ivi.ini` configuration file.

Parameters

Input

Name	Type	Description
vInstrName	`ViConstString`	The name of a `VInstr` entry that you create using `Ivi_DefineVInstr`. You do not have to include the `"VInstr->"` prefix in the name.

Return Value

None

See Also

`Ivi_DefineVInstr`

Ivi_UnlockSession

```
ViStatus status = Ivi_UnlockSession (ViSession vi,
                      ViBoolean *callerHasLock);
```

Purpose

Releases a lock that you acquire on an IVI session using `Ivi_LockSession`. Refer to `Ivi_LockSession` for additional information on IVI session locks.

Instrument drivers export `Ivi_UnlockSession` to the user through the *`Prefix`*`_UnlockSession` function.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.

Input/Output

Name	Type	Description
callerHasLock	`ViBoolean`	Indicates if the calling function currently has a lock on the IVI session. You can pass `VI_NULL`. Refer to function description for `Ivi_LockSession` for more information.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_LockSession`, `Ivi_SpecificDriverNew`, `Ivi_Dispose`

Ivi_Update

```
ViStatus status = Ivi_Update (ViSession vi);
```

Purpose

Performs the attribute updates that you previously deferred.

You can arrange for one or more `Ivi_SetAttribute` calls to postpone the actual updating of attribute values. You can then call `Ivi_Update` to trigger all of the updates to occur at once. This capability is called Deferring Updates. Refer to the *Deferred Updates* section in Chapter 2, *IVI Architecture Overview*, for general information on deferring updates.

This section describes the behavior of `Ivi_Update` in detail. While processing deferred updates, `Ivi_Update` calls the IVI session's buffered I/O callback with various messages. By default, the IVI engine installs the `Ivi_DefaultBufferedIOCallback` as the buffered I/O callback for the session. To provide a more complete explanation of the update process, this section also describes the actions that `Ivi_DefaultBufferedIOCallback` performs in response to each of the messages it receives from `Ivi_Update`.

Before it processes any deferred updates, `IVI_Update` invokes the buffered I/O callback with the `IVI_MSG_START_UPDATE` message. `Ivi_DefaultBufferedIOCallback` responds to this message by configuring the VISA I/O session so that it does not send `END` after each VISA write call and so that it flushes the write buffer only when it is full.

For each deferred update, `Ivi_Update` performs the following actions:

1. Compares the new value with the current cache value of the attribute. If the cache value is a value that the IVI engine obtained by querying the instrument and the attribute has a compare callback, `Ivi_Update` invokes the compare callback. Otherwise, it makes the comparison based on strict equality.
2. If the new value is not equal to the cache value or the cache value is invalid, `Ivi_Update` invokes the write callback for the attribute. The write callback might perform I/O to send the data to the instrument.
3. If the `IVI_VAL_WAIT_FOR_OPC_AFTER_WRITES` flag is set for the attribute, `Ivi_Update` invokes the operation complete (OPC) callback you provide for the session. Before it does so, it calls the buffered I/O callback with the `IVI_MSG_FLUSH` message. `Ivi_DefaultBufferedIOCallback` responds to this message by flushing the VISA I/O buffer.
4. If the `IVI_VAL_FLUSH_ON_WRITE` flag is set for the attribute and `Ivi_Update` does not invoke the operation complete callback for attribute, `Ivi_Update` calls the buffered I/O callback with the `IVI_MSG_FLUSH` message.

During this process, you might call one of the Ivi_GetAttribute functions from a compare or write callback. If you do so, Ivi_Update invokes the buffered I/O callback with the IVI_MSG_SUSPEND message before it invokes the read callback, and it sends the IVI_MSG_RESUME message afterward. Ivi_Update also sends the suspend and resume messages around any calls it makes to the operation complete callback.

When Ivi_DefaultBufferedIOCallback receives an IVI_MSG_SUSPEND message, it restores the VISA I/O session to its original configuration. When it receives an IVI_MSG_RESUME message, it configures the VISA I/O session in the same way it does when it receives the IVI_MSG_START_UPDATE message.

The buffered I/O callback can receive multiple suspend messages before it receives a resume message. It must take action only on the first of these suspend messages, and it must not take action on resume messages until the number of resume messages matches the number of suspend messages.

After Ivi_Update performs all deferred updates, it performs the following actions:

1. Invokes the check status callback for the session if the IVI_ATTR_QUERY_INSTR_STATUS attribute is enabled and the IVI_VAL_DONT_CHECK_STATUS flag is 0 for at least one of the attributes that it updates. Before invoking the check status callback, Ivi_Update invokes the buffered I/O callback with the IVI_MSG_SUSPEND message.
2. Invokes the buffered I/O callback with the IVI_MSG_FLUSH message.
3. Invokes the buffered I/O callback with the IVI_MSG_END_UPDATE message. Ivi_DefaultBufferedIOCallback responds to this message by restoring the VISA I/O session to its original configuration.

Parameters

Input

Name	Type	Description
vi	ViSession	The ViSession handle that you obtain from Ivi_SpecificDriverNew. The handle identifies a particular IVI session.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_DefaultBufferredIOCallback`, `Ivi_SetAttributeViInt32`, `Ivi_SetAttributeViReal64`, `Ivi_SetAttributeViString`, `Ivi_SetAttributeViBoolean`, `Ivi_SetAttributeViSession`, `Ivi_GetAttributeViAddr`

Ivi_UseSpecificSimulation

```
ViBoolean useSpecificSimulation = Ivi_UseSpecificSimulation (ViSession vi);
```

Purpose

Returns the current value of the IVI_ATTR_USE_SPECIFIC_SIMULATION attribute for the IVI session you specify. The attribute controls whether the specific driver or the class driver simulates I/O operations when simulation is enabled.

High-level functions in class and specific instrument drivers use Ivi_UseSpecificSimulation. Ivi_UseSpecificSimulation provides fast, convenient access to the IVI_ATTR_USE_SPECIFIC_SIMULATION attribute because it performs no error checking and does not lock the session.

If you pass an invalid session handle, Ivi_UseSpecificSimulation returns VI_FALSE.

Note ***Do not call*** Ivi_UseSpecificSimulation ***unless you have already locked the session.***

Parameters

Input

Name	Type	Description
vi	ViSession	The ViSession handle that you obtain from Ivi_SpecificDriverNew. The handle identifies a particular IVI session.

Return Value

Name	Type	Description
useSpecificSimulation	ViBoolean	1 = VI_TRUE - Simulate in specific driver 0 = VI_FALSE - Simulate in class driver

See Also

Ivi_Simulating

Ivi_ValidateAttrForChannel

```
ViStatus status = Ivi_ValidateAttrForChannel (ViSession vi,
                           ViConstString channelName, ViAttr attributeID);
```

Purpose

Checks whether you can use an attribute on a particular channel. If either the **attributeID** or **channelName** parameter is invalid for the session, `Ivi_ValidateAttrForChannel` returns an error. Otherwise, it checks for the following cases in which the combination of **attributeID** and **channelName** is invalid:

- The channel name is `VI_NULL` or the empty string and the attribute is channel-based. An attribute is channel-based if its `IVI_VAL_MULTI_CHANNEL` flag is set. In this case, `Ivi_ValidateAttrForChannel` returns the `IVI_ERROR_CHANNEL_NAME_NOT_ALLOWED` error code.
- The channel name refers to a specific channel and the attribute is not channel-based. In this case, `Ivi_ValidateAttrForChannel` returns the `IVI_ERROR_CHANNEL_NAME_REQUIRED` error code.
- The channel name refers to a specific channel, the attribute is channel-based, but the instrument driver calls `Ivi_RestrictAttrToChannel` to exclude the channel from using the attribute. In this case, `Ivi_ValidateAttrForChannel` returns the `IVI_ERROR_ATTR_NOT_VALID_FOR_CHANNEL` error code.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
channelName	`ViConstString`	The channel name that you want to verify as valid for a particular channel. Refer to the *Parameter Discussion* section.
attributeID	`ViAttr`	The ID of the attribute. Refer to the *Attribute IDs* section in Chapter 2, *IVI Architecture Overview*.

Return Value

Name	Type	Description
status	ViStatus	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

The **channelName** parameter can be any of the following:

- VI_NULL or an empty string.
- A specific driver channel string, which is one that the specific instrument driver specifies as valid using Ivi_BuildChannelTable or Ivi_AddToChannelTable.
- A virtual channel name that the user specifies in the ivi.ini configuration file.

See Also

Ivi_BuildChannelTable, Ivi_AddToChannelTable, Ivi_RestrictAttrToChannels, Ivi_CoerceChannelName, Ivi_GetUserChannelName, Ivi_GetNthChannelString

Ivi_ValidateRangeTable

```
ViStatus status = Ivi_ValidateRangeTable (IviRangeTablePtr rangeTable);
```

Purpose

Validates a range table. If you pass `VI_NULL` for the **rangeTable** parameter, the `Ivi_ValidateRangeTable` returns `VI_SUCCESS`. If you specify a non-NULL range table, the function returns an error when the range table type is not valid or the number of entries is zero.

The valid range table types are the following:

```
IVI_VAL_DISCRETE     0
IVI_VAL_RANGED       1
IVI_VAL_COERCED      2
```

A range table has zero entries if the first entry has the following value in the **cmdString** field:

```
IVI_RANGE_TABLE_END_STRING ((ViString)(-1))
```

Refer to the *Range Tables* section in Chapter 2, *IVI Architecture Overview*, for more information.

Parameters

Input

Name	Type	Description
rangeTable	`IviRangeTablePtr`	The address of the range table you want to validate. Can be `VI_NULL`.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates that the range table is not valid. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_GetAttrRangeTable`, `Ivi_GetRangeTableNumEntries`, `Ivi_RangeTableNew`, `Ivi_SetRangeTableEntry`, `Ivi_SetRangeTableEnd`

Ivi_ValidateSession

```
ViStatus status = Ivi_ValidateSession (ViSession vi);
```

Purpose

Checks an IVI session handle for validity. If the session is invalid, `Ivi_ValidateSession` returns an error code but does not set the primary error code, secondary error code, or error elaboration string for the current thread.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates that the session handle is not valid. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_SpecificDriverNew`

Ivi_WriteFromFile

```
ViStatus status = Ivi_WriteFromFile (ViSession vi, ViConstString filename,
                           ViInt32 writeNumberofBytes, ViInt32 byteOffset,
                           ViInt32 *returnCount);
```

Purpose

Reads data from a file you specify and writes it to an instrument using VISA I/O. Use `Ivi_WriteFromFile` internally in your instrument driver.

`Ivi_WriteFromFile` assumes that the `IVI_ATTR_IO_SESSION` attribute for the IVI session you specify holds a valid VISA session for the instrument.

`Ivi_WriteFromFile` opens the file in binary mode.

`Ivi_WriteFromFile` calls `Ivi_SetNeedToCheckStatus` with `VI_TRUE`.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
filename	`ViConstString`	The pathname of the file to write the data to. Refer to the *Parameter Discussion* section.
writeNumberofBytes	`ViInt32`	The maximum number of bytes to read from the file and write to the instrument.
byteOffset	`ViInt32`	The byte offset in the file at which to start reading. Refer to the *Parameter Discussion* section.

Output

Name	Type	Description
returnCount	`ViInt32`	Returns the number of bytes the function successfully writes to the instrument.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

For the filename parameter, you can specify an absolute pathname, a relative pathname, or a simple filename. `Ivi_WriteFromFile` treats relative pathnames and simple filenames as relative to the current working directory.

If you specify a literal string for the filename parameter under Windows, be sure to use double backslashes to represent one backslash in the pathname.

If the file contains header information that you do not want to write to the instrument, you can skip over the header by passing the number of bytes in the header for the **byteOffset** parameter. To write from the beginning of the file, pass 0.

See Also

`Ivi_ReadToFile`, `Ivi_WriteInstrData`, `Ivi_ReadInstrData`, `Ivi_IOSession`, `Ivi_SetNeedToCheckStatus`

Ivi_WriteInstrData

```
ViStatus status = Ivi_WriteInstrData (ViSession vi,
                          ViConstString writeBuffer);
```

Purpose

Writes a command string directly to an instrument using VISA I/O. `Ivi_WriteInstrData` bypasses the IVI engine's attribute-state-caching mechanism and therefore always invalidates all attribute cache values for the session. Use `Ivi_WriteInstrData` only to implement the *`Prefix`*`_WriteInstrData` function that your instrument driver exports to the user.

`Ivi_WriteInstrData` assumes that the `IVI_ATTR_IO_SESSION` attribute for the IVI session you specify holds a valid VISA session for the instrument.

`Ivi_WriteInstrData` calls `Ivi_SetNeedToCheckStatus` with `VI_TRUE`.

Parameters

Input

Name	Type	Description
vi	`ViSession`	The `ViSession` handle that you obtain from `Ivi_SpecificDriverNew`. The handle identifies a particular IVI session.
writeBuffer	`ViConstString`	The string you want to send to the instrument.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

See Also

`Ivi_ReadInstrData`, `Ivi_ReadToFile`, `Ivi_WriteFromFile`, `Ivi_IOSession`, `Ivi_SetNeedToCheckStatus`

Ivi_WriteRunTimeDefinesToFile

```
ViStatus status = Ivi_WriteRunTimeDefinesToFile (ViConstString pathname);
```

Purpose

Writes a file that contains the configuration entries you create at run-time using `Ivi_DefineLogicalName`, `Ivi_DefineVInstr`, `Ivi_DefineDriver`, `Ivi_DefineHardware`, and `Ivi_DefineClass`.

You cannot use the filename `"ivi.ini"` for the file, regardless of case and regardless of directory path. The IVI Library reserves that name.

`Ivi_WriteRunTimeDefinesToFile` creates the file if it does not already exist, or truncates the file if it already exists.

Parameters

Name	Type	Description
pathname	`ViConstString`	The pathname of the file in which you want to save the list of run-time configuration entries.

Return Value

Name	Type	Description
status	`ViStatus`	A negative value indicates an error. 0 indicates success. Refer to the *Status Codes* section at the end of this chapter.

Parameter Discussion

If you specify a literal string for the pathname parameter under Windows, be sure to use double backslashes to represent one backslash in the pathname.

See Also

`Ivi_DefineLogicalName`, `Ivi_DefineVInstr`, `Ivi_DefineDriver`, `Ivi_DefineHardware`, `Ivi_DefineClass`

Status Codes

IVI Library functions can return error and warning values from several sets of status codes. Some status codes are unique to the IVI Library. Other status codes are the same codes that VISA Library functions return. Still others are error or warning values that functions in specific instrument drivers return. Each set of status codes has its own numeric range.

Regardless of the source of the status code, 0 always indicates success, a positive value indicates a warning, and a negative value indicates an error.

The following table defines the different ranges of status codes. The table lists the include files that contain the defined constants for the particular status codes.

Table 11-5. Status Code Ranges

Status Code Type	Numeric Range (in Hex)	Include File
IVI Errors	`BFFA0000` to `BFFA1FFF`	`ivi.h`
IVI Warnings	`3FFA0000` to `3FFA1FFF`	`ivi.h`
Driver Errors	`BFFA4000` to `BFFA5FFF`	`Prefix.h`
Driver Warnings	`3FFA4000` to `3FFA5FFF`	`Prefix.h`
Common Errors	`BFFC0000` to `BFFCFFFF`	`vpptype.h`
Common Warnings	`3FFC0000` to `3FFCFFFF`	`vpptype.h`
VISA Errors	`BFFF0000` to `BFFFFFFF`	`visa.h`
VISA Warnings	`3FFF0000` to `3FFFFFFF`	`visa.h`

The Common Errors and Warnings are values that VXI*plug&play* defines and that specific instrument drivers return. They provide a consistent set of codes for error and warning conditions that are common among all instrument drivers. Each particular instrument driver defines its own set of Driver Errors and Warnings. The status codes values for one driver can overlap the status code values for other drivers.

The IVI Library and instrument driver include files define particular status codes as the unsigned sum of a base value and a decimal integer value. The following are the base values:

Table 11-6. Default Values of Defined Constants

Status Code Type	Defined Constant for Base Value	Value
IVI Errors	`IVI_ERROR_BASE`	`BFFA0000`
IVI Warnings	`IVI_WARN_BASE`	`3FFA0000`
Driver Errors	`IVI_SPECIFIC_ERROR_BASE`	`BFFA4000`
Driver Warnings	`IVI_SPECIFIC_WARN_BASE`	`3FFA4000`

For example, if you pass an invalid attribute ID to an IVI Library function, the function returns `IVI_ERROR_INVALID_ATTRIBUTE`, which `ivi.h` defines as `IVI_ERROR_BASE` + 12, or `0xBFFA000C`.

The following tables contain the IVI Status Codes, the Common Status Codes, and the most commonly used VISA Status Codes.

Table 11-7. IVI Errors and Warnings

Status	Description
0 No error	(the call was successful).
`BFFA0001`	Instrument error. Call *`Prefix`*`_error_query`.
`BFFA0002`	Cannot open file.
`BFFA0003`	Error reading from file.
`BFFA0004`	Error writing to file.
`BFFA0005`	Driver module file not found.
`BFFA0006`	Cannot open driver module file for reading.
`BFFA0007`	Driver module has invalid file format or invalid data.
`BFFA0008`	Driver module contains undefined references.
`BFFA0009`	Cannot find function in driver module.
`BFFA000A`	Failure loading driver module.
`BFFA000B`	Invalid path name.
`BFFA000C`	Invalid attribute.
`BFFA000D`	IVI attribute is not writable.
`BFFA000E`	IVI attribute is not readable.

Table 11-7. IVI Errors and Warnings (Continued)

Status	Description
BFFA000F	Invalid parameter.
BFFA0010	Invalid value.
BFFA0011	Function not supported.
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.
BFFA0017	Specified item already exists.
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.
BFFA001D	Object or item is not initialized.
BFFA001E	Non-interchangeable behavior.
BFFA001F	No channel table has been built for the session.
BFFA0020	Channel name specified is not valid.
BFFA0021	Unable to allocate system resource.
BFFA0022	Permission to access file was denied.
BFFA0023	Too many files are already open.
BFFA0024	Unable to create temporary file in target directory.
BFFA0025	All temporary filenames already used.
BFFA0026	Disk is full.
BFFA0027	Cannot find configuration file on disk.
BFFA0028	Cannot open configuration file.

Table 11-7. IVI Errors and Warnings (Continued)

Status	Description
BFFA0029	Error reading configuration file.
BFFA002A	Invalid ViInt32 value in configuration file.
BFFA002B	Invalid ViReal64 value in configuration file.
BFFA002C	Invalid ViBoolean value in configuration file.
BFFA002D	Entry missing from configuration file.
BFFA002E	Initialization failed in driver DLL.
BFFA002F	Driver module has unresolved external reference.
BFFA0030	Cannot find CVI Run-Time Engine.
BFFA0031	Cannot open CVI Run-Time Engine.
BFFA0032	CVI Run-Time Engine has invalid format.
BFFA0033	CVI Run-Time Engine is missing required function(s).
BFFA0034	CVI Run-Time Engine initialization failed.
BFFA0035	CVI Run-Time Engine has unresolved external reference.
BFFA0036	Failure loading CVI Run-Time Engine.
BFFA0037	Cannot open DLL for read exports.
BFFA0038	DLL file is corrupt.
BFFA0039	No DLL export table in DLL.
BFFA003A	Unknown attribute name in default configuration file.
BFFA003B	Unknown attribute value in default configuration file.
BFFA003C	Memory pointer specified is not known.
BFFA003D	Unable to find any channel strings.
BFFA003E	Duplicate channel string.
BFFA003F	Duplicate virtual channel name.
BFFA0040	Missing virtual channel name.
BFFA0041	Bad virtual channel name.
BFFA0042	Unassigned virtual channel name.

Table 11-7. IVI Errors and Warnings (Continued)

Status	Description
BFFA0043	Bad virtual channel assignment.
BFFA0044	Channel name required.
BFFA0045	Channel name not allowed.
BFFA0046	Attribute not valid for channel.
BFFA0047	Attribute must be channel based.
BFFA0048	Channel already excluded.
BFFA0049	Missing option name (nothing before the '=').
BFFA004A	Missing option value (nothing after the '=').
BFFA004B	Bad option name.
BFFA004C	Bad option value.
BFFA004D	Operation only valid on a class driver session.
BFFA004E	'ivi.ini' filename is reserved.
BFFA004F	Duplicate run-time configuration entry.
BFFA0050	Index parameter is one-based.
BFFA0051	Index parameter is too high.
BFFA0052	Attribute is not cacheable.
BFFA0053	You cannot export a ViAddr attribute to the user.

Table 11-8. Common Errors and Warnings

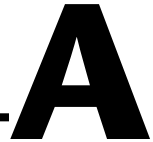
Status	Description
BFFC0001	Parameter 1 out of range, or error trying to set it.
BFFC0002	Parameter 2 out of range, or error trying to set it.
BFFC0003	Parameter 3 out of range, or error trying to set it.
BFFC0004	Parameter 4 out of range, or error trying to set it.
BFFC0005	Parameter 5 out of range, or error trying to set it.
BFFC0006	Parameter 6 out of range, or error trying to set it.

Table 11-8. Common Errors and Warnings (Continued)

Status	Description
BFFC0007	Parameter 7 out of range, or error trying to set it.
BFFC0008	Parameter 8 out of range, or error trying to set it.
BFFC0011	Instrument failed the ID Query.
BFFC0012	Invalid response from instrument.
3FFC0101	Instrument does not have ID Query capability.
3FFC0102	Instrument does not have Reset capability.
3FFC0103	Instrument does not have Self-Test capability.
3FFC0104	Instrument does not have Error Query capability.
3FFC0105	Instrument does not have Revision Query capability.

Table 11-9. Most Often Encountered VISA Errors and Warnings

Status	Description
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.
3FFF0085	The status value you passed is unknown.



Customer Communication

For your convenience, this appendix contains forms to help you gather the information necessary to help us solve your technical problems and a form you can use to comment on the product documentation. When you contact us, we need the information on the Technical Support Form and the configuration form, if your manual contains one, about your system configuration to answer your questions as quickly as possible.

National Instruments has technical assistance through electronic, fax, and telephone systems to quickly provide the information you need. Our electronic services include a bulletin board service, an FTP site, a fax-on-demand system, and e-mail support. If you have a hardware or software problem, first try the electronic support systems. If the information available on these systems does not answer your questions, we offer fax and telephone support through our technical support centers, which are staffed by applications engineers.

Electronic Services

Bulletin Board Support

National Instruments has BBS and FTP sites dedicated for 24-hour support with a collection of files and documents to answer most common customer questions. From these sites, you can also download the latest instrument drivers, updates, and example programs. For recorded instructions on how to use the bulletin board and FTP services and for BBS automated information, call 512 795 6990. You can access these services at:

United States: 512 794 5422
Up to 14,400 baud, 8 data bits, 1 stop bit, no parity

United Kingdom: 01635 551422
Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

France: 01 48 65 15 59
Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

FTP Support

To access our FTP site, log on to our Internet host, `ftp.natinst.com`, as `anonymous` and use your Internet address, such as `joesmith@anywhere.com`, as your password. The support files and documents are located in the `/support` directories.

Fax-on-Demand Support

Fax-on-Demand is a 24-hour information retrieval system containing a library of documents on a wide range of technical information. You can access Fax-on-Demand from a touch-tone telephone at 512 418 1111.

E-Mail Support (Currently USA Only)

You can submit technical support questions to the applications engineering team through e-mail at the Internet address listed below. Remember to include your name, address, and phone number so we can contact you with solutions and suggestions.

`support@natinst.com`

Telephone and Fax Support

National Instruments has branch offices all over the world. Use the list below to find the technical support number for your country. If there is no National Instruments office in your country, contact the source from which you purchased your software to obtain support.

Country	Telephone	Fax
Australia	03 9879 5166	03 9879 6277
Austria	0662 45 79 90 0	0662 45 79 90 19
Belgium	02 757 00 20	02 757 03 11
Brazil	011 288 3336	011 288 8528
Canada (Ontario)	905 785 0085	905 785 0086
Canada (Québec)	514 694 8521	514 694 4399
Denmark	45 76 26 00	45 76 26 02
Finland	09 725 725 11	09 725 725 55
France	01 48 14 24 24	01 48 14 24 14
Germany	089 741 31 30	089 714 60 35
Hong Kong	2645 3186	2686 8505
Israel	03 6120092	03 6120095
Italy	02 413091	02 41309215
Japan	03 5472 2970	03 5472 2977
Korea	02 596 7456	02 596 7455
Mexico	5 520 2635	5 520 3282
Netherlands	0348 433466	0348 430673
Norway	32 84 84 00	32 84 86 00
Singapore	2265886	2265887
Spain	91 640 0085	91 640 0533
Sweden	08 730 49 70	08 730 43 70
Switzerland	056 200 51 51	056 200 51 55
Taiwan	02 377 1200	02 737 4644
United Kingdom	01635 523545	01635 523154
United States	512 795 8248	512 794 5678

Technical Support Form

Photocopy this form and update it each time you make changes to your software or hardware, and use the completed copy of this form as a reference for your current configuration. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

If you are using any National Instruments hardware or software products related to this problem, include the configuration forms from their user manuals. Include additional pages if necessary.

Name ____

Company ____

Address ____

____

Fax (___) _________________Phone (___) ____________________________

Computer brand_____________ Model ____________________Processor____________________

Operating system (include version number) _________________________________

Clock speed ______MHz RAM _____MB Display adapter ________________________

Mouse ___yes ___no Other adapters installed ______________________________

Hard disk capacity _____MB Brand______________________________________

Instruments used ____

____

National Instruments hardware product model _____________ Revision ___________________

Configuration ____

National Instruments software product ___________________ Version ___________________

Configuration ____

The problem is: ____

____

____

____

____

List any error messages: ____

____

____

The following steps reproduce the problem: __________________________________

____

____

____

____

LabWindows/CVI Hardware and Software Configuration Form

Record the settings and revisions of your hardware and software on the line to the right of each item. Complete a new copy of this form each time you revise your software or hardware configuration, and use this form as a reference for your current configuration. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

National Instruments Products

DAQ hardware ______________________________

Interrupt level of hardware ______________________________

DMA channels of hardware ______________________________

Base I/O address of hardware ______________________________

Programming choice ______________________________

NI-DAQ, LabVIEW, or
LabWindows version ______________________________

Other boards in system ______________________________

Base I/O address of other boards ______________________________

DMA channels of other boards ______________________________

Interrupt level of other boards ______________________________

Other Products

Computer make and model ______________________________

Microprocessor ______________________________

Clock frequency or speed ______________________________

Type of video board installed ______________________________

Operating system version ______________________________

Operating system mode ______________________________

Programming language ______________________________

Programming language version ______________________________

Other boards in system ______________________________

Base I/O address of other boards ______________________________

DMA channels of other boards ______________________________

Interrupt level of other boards ______________________________

Documentation Comment Form

National Instruments encourages you to comment on the documentation supplied with our products. This information helps us provide quality products to meet your needs.

Title: *LabWindows/CVI Instrument Driver Developers Guide*

Edition Date: February 1998

Part Number: 320684D-01

Please comment on the completeness, clarity, and organization of the manual.

____

____

____

____

____

____

____

If you find errors in the manual, please record the page numbers and describe the errors.

____

____

____

____

____

____

____

Thank you for your help.

Name ____

Title ____

Company ____

Address ____

____

E-Mail Address ____

Phone (___)____________________________ Fax (___) ____________________________

Mail to: Technical Publications
National Instruments Corporation
6504 Bridge Point Parkway
Austin, Texas 78730-5039

Fax to: Technical Publications
National Instruments Corporation
512 794 5678

Glossary

Prefix	Meaning	Value
p-	pico-	10^{-12}
n-	nano-	10^{-9}
μ-	micro-	10^{-6}
m-	milli-	10^{-3}
k-	kilo-	10^{3}
M-	mega-	10^{6}

A

ANSI — American National Standards Institute

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 upon whether the control is in the up or down position.

C

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.

E

external module — A `.lib`, `.obj`, or `.dll` file that can be loaded and executed.

F

`.fp` file	A file containing 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 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
Hz	hertz

I

in.	inches
include file	A file that contains function declarations, constant definitions, and external declaration of global variables exported by the instrument driver.
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.

K

ksamples — 1,000 samples

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.

pt — points

pts/s — points per second

R

reporting method	integer function and return the appropriate value.
return value control	A function panel control that displays a value returned from the primary function.
return value error	The method used to declare each instrument driver routine as an
ring control	A control that displays a list of options one option at a time.

S

s	seconds
s/pt	seconds per point
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.
slide control	A function panel control that resembles a mechanical slide switch; it inserts a parameter value depending upon the position of the cross-bar on the slide control.

V

V	volts
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.

Index

A

action/status functions, 1-11
Add Attribute button, Edit Driver Attributes dialog box, 4-5
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, 6-5
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, 4-6
architecture. *See* instrument driver architecture.
array data types, user-defined, 3-15
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, 4-5 to 4-6
    illustration, 4-3
    Instrument Attributes list box, 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.
  creation functions
    function tree, 11-3
    Ivi_AddAttributeViAddr, 11-15 to 11-17
    Ivi_AddAttributeViBoolean, 11-18 to 11-20
    Ivi_AddAttributeViInt32, 11-21 to 11-23
    Ivi_AddAttributeViReal64, 11-24 to 11-27
    Ivi_AddAttributeViSession, 11-28 to 11-30
    Ivi_AddAttributeViString, 11-31 to 11-33
  information functions
    function tree, 11-6 to 11-7
    Ivi_AttributeIsCached, 11-37
    Ivi_AttributeUpdateIsPending, 11-38
    Ivi_DisposeInvalidationList, 11-87
    Ivi_GetAttributeFlags, 11-92
    Ivi_GetAttributeName, 11-93 to 11-94
    Ivi_GetAttributeType, 11-95
    Ivi_GetAttrMinMaxViInt32, 11-102 to 11-103
    Ivi_GetAttrMinMaxViReal64, 11-104 to 11-105

Ivi_GetInvalidationList, 11-111 to 11-112
Ivi_GetNextCoercionInfo, 11-116 to 11-117
Ivi_GetNthAttribute, 11-118
Ivi_GetNumAttributes, 11-123
Ivi_SetAttributeFlags, 11-192 to 11-193
purpose and use, 1-11
attributes. *See also* state-caching, IVI.
class
constant name for ID, 2-9
definition, 2-7
comparison precision, 2-18
creating and declaring, 2-9 to 2-13
flags, 2-10 to 2-13
description of individual flags, 2-11 to 2-13
list of flags (table), 2-10 to 2-11
IDs, 2-9 to 2-10
inherent. *See* inherent attributes.
instrument-specific
constant name for ID, 2-10
definition, 2-7
invalidation
by changing one attribute, 2-20
by changing two attributes, 2-20
purpose and use, 2-7
range tables. *See* range tables.
types of attributes, 2-7
unsupported, in generated driver files, 3-9

B

Binary command, Create menu, 6-12 to 6-13
binary controls
control label, 6-12
Create Binary Control dialog box
available items, 6-12 to 6-13
creating function window (example), 6-26 to 6-27, 6-29
creating, 6-12 to 6-13, 6-26 to 6-27
data type, 6-12
default value, 6-13
definition, 6-12
Edit On/Off Settings dialog box
available items, 6-13
creating function window (example), 6-27
Edit On/Off Settings dialog box, 6-29
parameter position, 6-12
buffered I/O callback
possible values for msg parameter (table), 2-28
purpose and use, 2-27 to 2-28
bulletin board support, A-1

C

caching/status-checking control functions. *See also* state-caching, IVI.
Ivi_InvalidateAllAttributes, 11-158
Ivi_InvalidateAttribute, 11-159 to 11-160
Ivi_NeedToCheckStatus, 11-165 to 11-166
Ivi_SetNeedToCheckStatus, 11-210 to 11-211
callback functions, 2-22 to 2-25
check functions
default, 2-17 to 2-18
Ivi_SetAttrCheckCallbackViAddr, 11-181 to 11-183
Ivi_SetAttrCheckCallbackViBoolean, 11-181 to 11-183
Ivi_SetAttrCheckCallbackViInt32, 11-181 to 11-183
Ivi_SetAttrCheckCallbackViReal64, 11-181 to 11-183
Ivi_SetAttrCheckCallbackViSession, 11-181 to 11-183

Ivi_SetAttrCheckCallbackViString, 11-181 to 11-183
purpose and use, 2-23
coerce functions
default, 2-17 to 2-18
Ivi_SetAttrCoerceCallbackViAddr, 11-184 to 11-186
Ivi_SetAttrCoerceCallback ViBoolean, 11-184 to 11-186
Ivi_SetAttrCoerceCallback ViInt32, 11-184 to 11-186
Ivi_SetAttrCoerceCallback ViReal64, 11-184 to 11-186
Ivi_SetAttrCoerceCallback ViSession, 11-184 to 11-186
Ivi_SetAttrCoerceCallbackViString, 11-184 to 11-186
purpose and use, 2-23 to 2-25
compare functions
Ivi_SetAttrCompareCallback ViAddr, 11-187 to 11-189
Ivi_SetAttrCompareCallback ViBoolean, 11-187 to 11-189
Ivi_SetAttrCompareCallback ViInt32, 11-187 to 11-189
Ivi_SetAttrCompareCallback ViReal64, 11-187 to 11-189
Ivi_SetAttrCompareCallback ViSession, 11-187 to 11-189
Ivi_SetAttrCompareCallback ViString, 11-187 to 11-189
Ivi_SetAttrRangeTableCallback, 11-198 to 11-199
purpose and use, 2-25
comparison precision functions
Ivi_GetAttrComparePrecision, 11-91
Ivi_SetAttrComparePrecision, 11-190 to 11-191
default callbacks
function tree, 11-6
Ivi_DefaultBufferedIOCallback, 11-55
Ivi_DefaultCheckCallbackViInt32, 11-56 to 11-57
Ivi_DefaultCheckCallbackViReal64, 11-58 to 11-59
Ivi_DefaultCoerceCallback ViBoolean, 11-60 to 11-61
Ivi_DefaultCoerceCallbackViInt32, 11-62 to 11-63
Ivi_DefaultCoerceCallback ViReal64, 11-64 to 11-65
Ivi_DefaultCompareCallback ViReal64, 11-66 to 11-68
definition, 1-12
delete attribute function
Ivi_DeleteAttribute, 11-82
function tree, 11-3 to 11-4
overview, 2-8 to 2-9, 2-22
purpose and use, 1-12
range table callback, 2-25
read callback, 2-22 to 2-23
read functions
Ivi_SetAttrReadCallbackViAddr, 11-200 to 11-202
Ivi_SetAttrReadCallbackViBoolean, 11-200 to 11-202
Ivi_SetAttrReadCallbackViInt32, 11-200 to 11-202
Ivi_SetAttrReadCallbackViReal64, 11-200 to 11-202
Ivi_SetAttrReadCallbackViSession, 11-200 to 11-202
Ivi_SetAttrReadCallbackViString, 11-200 to 11-202
purpose and use, 2-22 to 2-23
session functions, 2-26 to 2-28
buffered I/O callback, 2-27 to 2-28
check status, 2-26 to 2-27
definition, 1-12
instruments without error queues, 2-27
operation complete callback, 2-26

string callbacks
Ivi_SetValInStringCallback, 11-217
types of, 1-12
write functions
Ivi_SetAttrWriteCallbackViAddr, 11-203 to 11-205
Ivi_SetAttrWriteCallbackViBoolean, 11-203 to 11-205
Ivi_SetAttrWriteCallbackViInt32, 11-203 to 11-205
Ivi_SetAttrWriteCallbackViReal64, 11-203 to 11-205
Ivi_SetAttrWriteCallbackViSession, 11-203 to 11-205
Ivi_SetAttrWriteCallbackViString, 11-203 to 11-205
purpose and use, 2-23
Change Control Type command, Edit menu, 6-5
Change Input Control Type dialog box, 6-31
channel functions
function tree, 11-3
Ivi_AddToChannelTable, 11-34
Ivi_BuildChannelTable, 11-39 to 11-41
Ivi_CoerceChannelName, 11-51 to 11-52
Ivi_GetNthChannelString, 11-119 to 11-120
Ivi_GetUserChannelName, 11-131 to 11-132
Ivi_RestrictAttrToChannels, 11-179 to 11-180
Ivi_ValidateAttrForChannel, 11-233 to 11-234
channel string, defined, 2-29
channel-based attribute, defined, 2-29
channels, 2-29 to 2-30
coercing and validating channel names, 2-30
overview, 2-29
passing channel names to IVI functions, 2-30
virtual channel names, 2-29
check attribute functions
Ivi_CheckAttributeViAddr, 11-42 to 11-43
Ivi_CheckAttributeViBoolean, 11-42 to 11-43
Ivi_CheckAttributeViInt32, 11-42 to 11-43
Ivi_CheckAttributeViReal64, 11-42 to 11-43
Ivi_CheckAttributeViSession, 11-42 to 11-43
Ivi_CheckAttributeViString, 11-42 to 11-43
check callback functions
default, 2-17 to 2-18
Ivi_SetAttrCheckCallbackViAddr, 11-181 to 11-183
Ivi_SetAttrCheckCallbackViBoolean, 11-181 to 11-183
Ivi_SetAttrCheckCallbackViInt32, 11-181 to 11-183
Ivi_SetAttrCheckCallbackViReal64, 11-181 to 11-183
Ivi_SetAttrCheckCallbackViSession, 11-181 to 11-183
Ivi_SetAttrCheckCallbackViString, 11-181 to 11-183
purpose and use, 2-23
check status callback, 2-26 to 2-27
checkAlloc macro, 11-10
checkErr macro, 11-10
checkWarn macro, 11-10
class attributes
constant name for ID, 2-9
definition, 2-7
Class command, Create menu, 5-7 to 5-8
class definitions for instruments, 1-10
classes. *See* function trees.

close functions for instrument drivers
definition, 1-12
not called by instrument driver application functions, 1-12
*Prefix*_close, 9-10 to 9-11
*Prefix*_IviClose, 9-12
coerce callback functions
default, 2-17 to 2-18
Ivi_SetAttrCoerceCallbackViAddr, 11-184 to 11-186
Ivi_SetAttrCoerceCallbackViBoolean, 11-184 to 11-186
Ivi_SetAttrCoerceCallbackViInt32, 11-184 to 11-186
Ivi_SetAttrCoerceCallbackViReal64, 11-184 to 11-186
Ivi_SetAttrCoerceCallbackViSession, 11-184 to 11-186
Ivi_SetAttrCoerceCallbackViString, 11-184 to 11-186
purpose and use, 2-23 to 2-25
coerced range tables
example, 2-16
IVI_VAL_COERCED, 2-14
coercion
channel names, 2-30
state-caching mechanism, 2-21
common control panel, 6-7
compare callback functions
Ivi_SetAttrCompareCallbackViAddr, 11-187 to 11-189
Ivi_SetAttrCompareCallbackViBoolean, 11-187 to 11-189
Ivi_SetAttrCompareCallbackViInt32, 11-187 to 11-189
Ivi_SetAttrCompareCallbackViReal64, 11-187 to 11-189
Ivi_SetAttrCompareCallbackViSession, 11-187 to 11-189
Ivi_SetAttrCompareCallbackViString, 11-187 to 11-189
Ivi_SetAttrRangeTableCallback, 11-198 to 11-199
purpose and use, 2-25
comparison precision for attributes, 2-18
comparison precision functions
Ivi_GetAttrComparePrecision, 11-91
Ivi_SetAttrComparePrecision, 11-190 to 11-191
component functions, 1-9 to 1-10
developer-specified functions, 1-9 to 1-10
instrument class definitions, 1-10
required functions, 1-9
configuration entries, IVI instrument drivers, 2-35 to 2-36
configuration functions
configuration file
function tree, 11-7
Ivi_GetIviIniDir, 11-113
Ivi_SetIviIniDir, 11-209
function tree, 11-7
logical names
Ivi_DisposeLogicalNamesList, 11-88
Ivi_GetLogicalNamesList, 11-114 to 11-115
Ivi_GetNthLogicalName, 11-121 to 11-122
overview, 1-10
run-time configuration
Ivi_DefineClass, 11-69 to 11-70
Ivi_DefineDriver, 11-71 to 11-73
Ivi_DefineHardware, 11-74 to 11-75
Ivi_DefineLogicalName, 11-76 to 11-77
Ivi_DefineVInstr, 11-78 to 11-81
Ivi_UndefClass, 11-223
Ivi_UndefDriver, 11-224
Ivi_UndefHardware, 11-225
Ivi_UndefLogicalName, 11-226

Ivi_UndefVInstr, 11-227
Ivi_WriteRunTimeDefinesToFile, 11-240
constants, defined, default values of (table), 11-242
context menu, Function Tree Editor, 5-2 to 5-4
available options, 5-3 to 5-4
illustration, 5-3
Control Help command, Edit menu, 6-6
Copy command, Edit menu, 5-5
Copy Controls command, Edit menu, 6-4
Copy Panel command, Edit menu, 6-5
copying and pasting help text, 7-9 to 7-11
Create Binary Control dialog box
available options, 6-12 to 6-13
creating function window (example), 6-26 to 6-27, 6-29
Edit On/Off Settings dialog box, 6-13
illustration, 6-12
Create Distribution Kit dialog box, 5-14
Create Dynamic Link Library dialog box, 5-13
Create Global Variable Control dialog box, 6-21
Create Input Control dialog box
available options, 6-9
creating function window (example), 6-27
illustration, 6-8
Create IVI Instrument Driver command, Tools menu
Function Panel Editor, 6-22
Function Tree Editor, 5-11
starting Instrument Driver Development Wizard, 3-4
Create menu
Function Panel Editor, 6-7 to 6-21. *See also* function panel controls.
available controls (figure), 6-8
Binary command, 6-12 to 6-13
Global Variable command, 6-20 to 6-21
Input command, 6-9 to 6-9
Message command, 6-21
Numeric command, 6-16 to 6-18
Output command, 6-19
Return Value command, 6-20
Ring command, 6-14 to 6-16
Slide command, 6-9 to 6-11
Function Tree Editor
available options, 5-6
Class command, 5-7 to 5-8
Function Panel Window command, 5-8 to 5-9
Instrument command, 5-7
Create Numeric Control dialog box, 6-16 to 6-18
Create Output Control dialog box, 6-19
Create Return Value Control dialog box, 6-20
Create Ring Control dialog box, 6-14 to 6-15
Create Slide Control dialog box
available options, 6-10
creating function window example, 6-28
Edit Label/Value Pairs dialog box, 6-10 to 6-11
illustration, 6-9
customer communication, *xxiv,* A-1 to A-2
Cut command, Edit menu, 5-5
Cut Controls command, Edit menu, 6-4
Cut Panel command, Edit menu, 6-4
cutting and pasting
controls (example), 6-32 to 6-33
functions and panels (example), 5-16 to 5-17

D

data functions, instrument drivers, 1-11
data types, 3-11 to 3-17
defining in header files (note), 6-23
intrinsic C data types, 3-12
meta data types, 3-13 to 3-14
Any Array, 3-13
Any Type, 3-13 to 3-14

definition, 3-13
Numeric Array, 3-13
Var Args, 3-14
predefined data types, 3-12 to 3-14
purpose and use, 3-11
user-defined, 3-14 to 3-15
array data types, 3-15
creating, 3-14 to 3-15
VISA data types
how to use, 3-16
list of types (table), 3-16
purpose and use, 3-16 to 3-17
Data Types command, Options menu, 6-23 to 6-24
.def file, generating for instrument driver, 5-13
default callback functions
default check and coerce callbacks, 2-17 to 2-18
function tree, 11-6
Ivi_DefaultBufferedIOCallback, 11-55
Ivi_DefaultCheckCallbackViInt32, 11-56 to 11-57
Ivi_DefaultCheckCallbackViReal64, 11-58 to 11-59
Ivi_DefaultCoerceCallbackViBoolean, 11-60 to 11-61
Ivi_DefaultCoerceCallbackViInt32, 11-62 to 11-63
Ivi_DefaultCoerceCallbackViReal64, 11-64 to 11-65
Ivi_DefaultCompareCallbackViReal64, 11-66 to 11-68
Default Panel Size command, Options menu, 6-24
deferred updates
buffered I/O callbacks, 2-27 to 2-28
Ivi_Update function, 11-229 to 11-231
purpose and use, 2-34 to 2-35
delete attribute callback function (Ivi_DeleteAttribute), 11-82
Detach Program command, Edit Instrument dialog box, 5-10
developing instrument drivers. *See* instrument driver development.
direct instrument I/O functions
function tree, 11-6
Ivi_ReadInstrData, 11-175 to 11-176
Ivi_ReadToFile, 11-177 to 11-178
Ivi_WriteFromFile, 11-237 to 11-238
Ivi_WriteInstrData, 11-239
discrete range tables
example, 2-15
IVI_VAL_DISCRETE, 2-14
Distribute Vertical Centers command, Edit menu, 6-6
Distribution command, Edit menu, 6-6
DLLs. *See* Windows DLLs.
documentation
conventions used in manual, *xxii-xxiii*
LabWindows/CVI
documentation set, *xxiii*
organization of manual, *xxi-xxii*
documentation for instrument driver, writing, 3-20
Done command, Edit Instrument dialog box, 5-11
dynamic range table functions
function tree, 11-5
Ivi_RangeTableFree, 11-171
Ivi_RangeTableNew, 11-172 to 11-174
Ivi_SetRangeTableEnd, 11-212
Ivi_SetRangeTableEntry, 11-213 to 11-214
purpose and use, 2-17
dynamic range tables, 2-17

E

Edit Attribute dialog box, Attribute Editor, 4-7 to 4-9
- Advanced dialog box (figure), 4-9
- entering information, 4-7 to 4-9
- illustration, 4-7

Edit button, Edit Driver Attributes dialog box, 4-5

Edit command, Instrument menu, 5-10 to 5-11. *See also* Edit Instrument dialog box.

Edit Control command, Edit menu, 6-5

Edit Driver Attributes dialog box, Attribute Editor, 4-3 to 4-6
- command buttons, 4-5 to 4-6
- illustration, 4-3
- Instrument Attributes list box, 4-4
- restrictions on modification to inherent and class attributes, 4-4

Edit Function command, Edit menu, 6-5

Edit Function Panel Window command
- Edit menu, 5-5, 6-1
- Function Tree Editor context menu, 5-5
- Options menu, 6-1

Edit Function Tree command
- Edit Instrument dialog box, 5-10 to 5-11
- Options menu, 6-24

Edit Help command, Edit menu, 5-5

Edit Instrument Attributes command, Tools menu
- Function Panel Editor, 6-22
- Function Tree Editor, 5-11

Edit Instrument dialog box
- available options, 5-10 to 5-11
- illustration, 5-10

Edit Label/Value Pairs dialog box
- adding label and value
  - ring control list, 6-15 to 6-16
  - slide control list, 6-11
- available options, 6-11
- changing control type (example), 6-31
- command buttons
  - ring controls, 6-16
  - slide controls, 6-11
- illustration
  - ring controls, 6-15
  - slide controls, 6-10
- positioning control (example), 6-28

Edit menu
- Find command, 5-5
- .FP Auto-Load List command, 5-6
- Function Panel Editor, 6-3 to 6-7
  - Align Horizontal Centers command, 6-5
  - Alignment command, 6-5
  - available options, 6-3
  - Change Control Type command, 6-5
  - Control Help command, 6-6
  - Copy Controls command, 6-4
  - Copy Panel command, 6-5
  - Cut Controls command, 6-4
  - Cut Panel command, 6-4
  - Distribute Vertical Centers command, 6-6
  - Distribution command, 6-6
  - Edit Control command, 6-5
  - Edit Function command, 6-5
  - Find command, 6-6
  - Function Help command, 6-7
  - Paste command, 6-4
  - Replace command, 6-6
  - Window Help command, 6-7
- Function Tree Editor, 5-5
- Help Editor dialog box, 7-3
- Replace command, 5-5

Edit Node command
- Edit menu, 5-5
- Function Tree Editor context menu, 5-5

Edit On/Off Settings dialog box
- available settings for binary controls, 6-13
- creating function window (example), 6-27, 6-29

Edit Value Set dialog box, 6-18
editing help information, 7-2 to 7-4
electronic support services, A- to A-2
e-mail support, A-2
Enable Auto Replace command, Tools menu
    Function Panel Editor, 6-22
    Function Tree Editor, 5-11
error codes, IVI Library
    common errors and warnings (table), 11-245 to 11-246
    IVI errors and warnings (table), 11-242 to 11-245
    VISA errors and warnings (table), 11-246
error info attributes, 2-37
error information functions
    function tree, 11-5
    get/clear error info functions, 1-12
    Ivi_ClearErrorInfo, 11-47 to 11-48
    Ivi_ClearInstrSpecificErrorQueue, 11-49
    Ivi_DequeueInstrSpecificError, 11-84 to 11-85
    Ivi_GetErrorInfo, 11-108 to 11-109
    Ivi_GetErrorMessage, 11-110
    Ivi_GetSpecificDriverStatusDesc, 11-125 to 11-126
    Ivi_InstrSpecificErrorQueueSize, 11-155 to 11-156
    Ivi_QueueInstrSpecificError, 11-168 to 11-169
    Ivi_SetErrorInfo, 11-206 to 11-208
error macros, 11-9 to 11-12
    checkAlloc, 11-10
    checkErr, 11-10
    checkWarn, 11-10
    examples, 11-12
    viCheckAlloc, 11-10
    viCheckErr, 11-10
    viCheckErrElab, 11-10
    viCheckParm, 11-11
    when to use viCheck macros, 11-11
error message function
    definition, 1-11
    *Prefix*_error_message, 9-20 to 9-21
error query function
    definition, 1-11
    *Prefix*_error_query, 9-17 to 9-19
error queue
    checking with check status callback, 2-26 to 2-27
    instruments without error queues, 2-27
error reporting, 11-8 to 11-12
    attributes for reporting, 11-8
    error macros, 11-9 to 11-12
    functions for accessing attribute values, 11-9
external interface model. *See* instrument driver architecture.

F

fax and telephone support numbers, A-2
Fax-on-Demand support, A-2
File menu
    Function Panel Editor, 6-3
    Function Tree Editor, 5-4
    Help Editor dialog box, 7-3
files for instrument drivers, 1-3
Find command, Edit menu
    Function Panel Editor, 6-6
    Function Tree Editor, 5-5
flags for attributes, 2-10 to 2-13
    description of individual flags, 2-11 to 2-13
    list of flags (table), 2-10 to 2-11
floating point numbers, precision comparison, 2-18
.FP Auto-Load List command, Edit menu, Function Tree Editor, 5-5, 5-6
.fp files, 1-3
FTP support, A-1
function classes. *See* function trees.

Function Help command, Edit menu, 6-7
function panel controls
adding help, 6-6
alignment commands
Align Horizontal Centers command, 6-5
Alignment command, 6-5
binary, 6-12 to 6-13, 6-26 to 6-27
changing control type
Change Control Type command, 6-5
Change Input Control Type dialog box, 6-31
example, 6-30 to 6-32
common control panel, 6-7
copying, 6-5
cutting and pasting (example), 6-32 to 6-33
distribution commands
Distribute Vertical Centers command, 6-6
Distribution command, 6-6
global variable, 6-20 to 6-21
help information, 7-6
input, 6-9 to 6-9, 6-27
message, 6-21
moving, 6-25
numeric, 6-16 to 6-18
output, 6-19
removing (cutting), 6-4
return value, 6-20
ring, 6-14 to 6-16
slide, 6-9 to 6-11, 6-28
types of controls (figure), 6-8
Function Panel Editor, 6-1 to 6-33
adding help information, 7-9
available menus, 6-3
Create menu, 6-7 to 6-21
Edit menu, 6-3 to 6-7
examples
changing control type, 6-30 to 6-32
creating function window, 6-26 to 6-30
cutting and pasting controls, 6-32 to 6-33
File menu, 6-3
illustration, 6-2
Instrument menu, 6-22
invoking, 6-1
items in Function Panel Editor, 6-2
Options menu, 6-23 to 6-24
Tools menu, 6-22
View menu, 6-21
Window menu, 6-22
function panel files (.fp), 1-3
Function Panel Window command, Create menu, 5-8 to 5-9
Function Panel windows
creating (example), 6-26 to 6-30
definition, 6-7
illustration, 6-30
function panels. *See also* function panel controls; interactive developer interface.
building for instrument drivers, 3-19
common control panel, 6-7
copying, 6-5
creating function window (example), 6-26 to 6-30
cutting and pasting (example), 5-16 to 5-17
definition, 6-7
determining movability, 6-24
generated by Instrument Driver Development Wizard, 3-6
help information
adding, 6-7
example, 7-7 to 7-8
converting old style to new style, 5-12
new style help only, 7-5

old style help only, 7-5 to 7-6
selecting style, 5-12
invoking Function Panel Editor, 6-1
IVI Library function panels, 11-1 to 11-8
moving controls, 6-25
operating, 6-24
removing (cutting), 6-4
setting default size, 6-24
toggling scroll bars, 6-24
Function Tree Editor, 5-4 to 5-17
adding help information (example), 7-7 to 7-8
available menus, 5-4
context menu, 5-2 to 5-4
Create menu, 5-6 to 5-9
Edit menu, 5-5
examples
cutting and pasting functions and panels, 5-16 to 5-17
editing items in function tree, 5-17
multiple classes in function tree, 5-15 to 5-16
File menu, 5-4
Function Tree Editor window (figure), 5-2
Instrument menu, 5-9 to 5-11
invoking, 5-1
invoking Function Panel Editor, 6-1
Options menu, 5-12 to 5-14
Tools menu, 5-11
Window menu, 5-12
Function Tree (*.fp) option, 5-15
Function Tree option, New command or Open command, 5-1
function trees
adding help information (example), 7-7 to 7-8
adding new functions, 5-8
building for instrument drivers, 3-19, 5-7
classes
adding new classes, 5-7
creating multiple classes (example), 5-15 to 5-16
help information, 7-5
inserting into existing tree, 5-8
number of functions and classes allowed (note), 5-8
cutting and pasting functions and panels (example), 5-16 to 5-17
definition, 5-1
IVI Library function tree, 11-1 to 11-8
class or panel, 11-3 to 11-8
structure, 11-2
number of functions and classes allowed (note), 5-8
functional body
definition, 1-5
purpose and use, 1-6
functions. *See* instrument driver functions; IVI Library.

G

Generate DEF File command, Options menu, 5-13
Generate DLL Make File command, Options menu, 5-12
Generate Documentation command, Options menu, 5-12
Generate Function Prototypes command, Options menu, 5-12
Generate IVI C++ Wrapper command, Tools menu, Function Tree Editor, 5-11
Generate ODL File command, Options menu, 5-13
Generate Source for Function Node command, Function Tree Editor context menu, 5-3
Generate Source for Function Tree command, Tools menu
Function Panel Editor, 6-22
Function Tree Editor, 5-11

Generate Windows Help command,
Options menu, 5-12
generated driver files, reviewing, 3-6 to 3-9
extended functions and attributes, 3-9
function panels, 3-6
include file, 3-9
source file, 3-8
.sub file, 3-7 to 3-8
using Attribute Editor, 3-9
get attribute functions
Ivi_GetAttributeViAddr, 11-96 to 11-98
Ivi_GetAttributeViBoolean,
11-96 to 11-98
Ivi_GetAttributeViInt32, 11-96 to 11-98
Ivi_GetAttributeViReal64,
11-96 to 11-98
Ivi_GetAttributeViSession,
11-96 to 11-98
Ivi_GetAttributeViString,
11-99 to 11-101
get/clear error info functions, 1-12
Global Variable command, Create menu,
6-20 to 6-21
global variable controls, 6-20 to 6-21
control label, 6-21
control width, 6-21
Create Global Variable Control
dialog box, 6-21
data type, 6-21
definition, 6-20
display format, 6-21
global variable name, 6-21
Go to Callback Source button,
Edit Driver Attributes dialog box, 4-5
Go to Declaration command
Function Tree Editor context menu, 5-4
Tools menu
Function Panel Editor, 6-22
Function Tree Editor, 5-11
Go to Definition command,
Function Tree Editor context menu, 5-4
Go to Range Table Source button,
Edit Driver Attributes dialog box, 4-5

H

Help Editor dialog box, 7-3 to 7-4
Edit menu, 7-4
File menu, 7-3
illustration, 7-3
Tools menu, 7-4
Window menu, 7-4
help information, 7-1 to 7-11
controls, 6-6, 7-6
editing, 7-2 to 7-4
examples
adding help in Function Panel
Editor, 7-9
adding help in Function Tree
Editor, 7-7 to 7-8
copying and pasting help text,
7-9 to 7-11
function classes, 7-5
function panels
new style help only, 7-5
old style help only, 7-5 to 7-6
selecting help style, 5-12
selecting old style or new style
help, 6-7
generating files for Windows Help
Compiler, 5-12
Help Editor dialog box, 7-3
instrument drivers
adding, 7-4
new style *vs.* old style help, 5-12, 7-1
types of help (table), 7-2
Help Style command, Options menu, 5-12

helper functions
attribute information
Ivi_AttributeIsCached, 11-37
Ivi_AttributeUpdateIsPending, 11-38
Ivi_DisposeInvalidationList, 11-87
Ivi_GetAttributeFlags, 11-92
Ivi_GetAttributeName, 11-93 to 11-94
Ivi_GetAttributeType, 11-95
Ivi_GetAttrMinMaxViInt32, 11-102 to 11-103
Ivi_GetAttrMinMaxViReal64, 11-104 to 11-105
Ivi_GetInvalidationList, 11-111 to 11-112
Ivi_GetNextCoercionInfo, 11-116 to 11-117
Ivi_GetNthAttribute, 11-118
Ivi_GetNumAttributes, 11-123
Ivi_SetAttributeFlags, 11-192 to 11-193
default callbacks
Ivi_DefaultBufferedIOCallback, 11-55
Ivi_DefaultCheckCallbackViInt32, 11-56 to 11-57
Ivi_DefaultCheckCallbackViReal64, 11-58 to 11-59
Ivi_DefaultCoerceCallback ViBoolean, 11-60 to 11-61
Ivi_DefaultCoerceCallbackViInt32, 11-62 to 11-63
Ivi_DefaultCoerceCallback ViReal64, 11-64 to 11-65
Ivi_DefaultCompareCallback ViReal64, 11-66 to 11-68
direct instrument I/O
Ivi_ReadInstrData, 11-175 to 11-176
Ivi_ReadToFile, 11-177 to 11-178
Ivi_WriteFromFile, 11-237 to 11-238
Ivi_WriteInstrData, 11-239
function tree, 11-6 to 11-7
inherent attribute accessors
Ivi_InterchangeCheck, 11-157
Ivi_IOSession, 11-161
Ivi_QueryInstrStatus, 11-167
Ivi_RangeChecking, 11-170
Ivi_Simulating, 11-218
Ivi_Spying, 11-222
Ivi_UseSpecificSimulation, 11-232
string callbacks
Ivi_SetValInStringCallback, 11-217
string/value tables
Ivi_GetStringFromTable, 11-129 to 11-130
Ivi_GetValueFromTable, 11-133 to 11-134
value manipulation
Ivi_CheckBooleanRange, 11-44
Ivi_CheckNumericRange, 11-45 to 11-46
Ivi_CoerceBoolean, 11-50
Ivi_CompareWithPrecision, 11-53 to 11-54

I

include file
contents of, 1-3
generated by Instrument Driver Development Wizard, reviewing, 3-9
inherent attribute accessor functions
function tree, 11-6
Ivi_InterchangeCheck, 11-157
Ivi_IOSession, 11-161
Ivi_QueryInstrStatus, 11-167
Ivi_RangeChecking, 11-170
Ivi_Simulating, 11-218

Ivi_Spying, 11-222
Ivi_UseSpecificSimulation, 11-232
inherent attribute reference
IVI_ATTR_ATTRIBUTE_CAPABILITIES, 2-38
IVI_ATTR_ATTR_SPY, 2-48 to 2-49
IVI_ATTR_BUFFERED_IO_CALLBACK, 2-38
IVI_ATTR_CACHE, 2-38 to 2-39
IVI_ATTR_CHECK_STATUS_CALLBACK, 2-39
IVI_ATTR_CLASS_MAJOR_VERSION, 2-39
IVI_ATTR_CLASS_MINOR_VERSION, 2-39
IVI_ATTR_CLASS_PREFIX, 2-40
IVI_ATTR_CLASS_REVISION, 2-40
IVI_ATTR_DEFER_UPDATE, 2-40 to 2-41
IVI_ATTR_DRIVER_MAJOR_VERSION, 2-41
IVI_ATTR_DRIVER_MINOR_VERSION, 2-41
IVI_ATTR_DRIVER_REVISION, 2-41
IVI_ATTR_DRIVER_SETUP, 2-41
IVI_ATTR_ENGINE_MAJOR_VERSION, 2-42
IVI_ATTR_ENGINE_MINOR_VERSION, 2-42
IVI_ATTR_ENGINE_REVISION, 2-42
IVI_ATTR_ERROR_ELABORATION, 2-42
IVI_ATTR_FUNCTION_CAPABILITIES, 2-42
IVI_ATTR_GROUP_CAPABILITIES, 2-43
IVI_ATTR_INTERCHANGE_CHECK, 2-43
IVI_ATTR_I/O_SESSION, 2-43
IVI_ATTR_LOGICAL_NAME, 2-44
IVI_ATTR_MODULE_PATHNAME, 2-44
IVI_ATTR_NUM_CHANNELS, 2-44
IVI_ATTR_OPC_CALLBACK, 2-45
IVI_ATTR_PRIMARY_ERROR, 2-45
IVI_ATTR_QUERY_INSTR_STATUS, 2-45 to 2-46
IVI_ATTR_RANGE_CHECK, 2-46
IVI_ATTR_RECORD_COERCIONS, 2-46 to 2-47
IVI_ATTR_RESOURCE_DESCRIPTOR, 2-47
IVI_ATTR_RETURN_DEFERRED_VALUES, 2-47
IVI_ATTR_SECONDARY_ERROR, 2-47
IVI_ATTR_SIMULATE, 2-48
IVI_ATTR_SPECIFIC_PREFIX, 2-48
IVI_ATTR_SUPPORTS_WR_BUF_OPER_MODE, 2-49
IVI_ATTR_UPDATING_VALUES, 2-49
IVI_ATTR_USE_SPECIFIC_SIMULATION, 2-49 to 2-50
IVI_ATTR_VISA_RM_SESSION, 2-50
inherent attributes
categories, 2-36 to 2-37
constant name for ID, 2-9
definition, 2-7
error info, 2-37
instrument capabilities, 2-37
session info, 2-36
session I/O, 2-36
user options, 2-36
version info, 2-37
initialize functions for instrument drivers
definition, 1-10
not called by instrument driver application functions (note), 1-12
*Prefix*_init, 2-5
*Prefix*_init, 9-3 to 9-5
*Prefix*_initWithOptions, 2-5
*Prefix*_initWithOptions, 9-6 to 9-7

*Prefix*_IviInit, 9-8 to 9-9
programming considerations, 2-5
input and output parameters for instrument drivers, 3-17
Input command, Create menu, 6-9 to 6-9
input controls
control label, 6-9
control width, 6-9
Create Input Control dialog box, 6-9 to 6-9, 6-27
creating, 6-9 to 6-9, 6-27
data type, 6-9
default value, 6-9
definition, 6-8
parameter position, 6-9
instrument capabilities attributes, 2-37
instrument class definitions
benefits, 1-10
purpose, 1-10
Instrument command, Create menu, 5-7
instrument driver architecture, 1-4 to 1-12
external interface model, 1-4 to 1-7
functional body, 1-6
general model (figure), 1-5
interactive developer interface, 1-7
IVI engine, 1-6
programmatic developer interface, 1-7
subroutine interface, 1-6 to 1-7
VISA I/O interface, 1-6
internal design model, 1-8 to 1-12
action/status functions, 1-11
application functions, 1-12
close function, 1-12
component functions, 1-9 to 1-10
configuration functions, 1-10
data functions, 1-11
illustration, 1-8
initialize function, 1-10
utility functions, 1-11 to 1-12
instrument driver development, 3-1 to 3-20. *See also* data types; function panels; Instrument Driver Development Wizard.
building function panels, 3-19
documenting the driver, 3-20
example, 10-1
function parameters
defining, 3-11
input and output parameters, 3-17
function tree
adding new classes, 5-7
adding new functions, 5-8
building, 3-19, 5-7
grouping functions hierarchically, 3-11
functions
defining, 3-9 to 3-11
grouping hierarchically, 3-11
return values, 3-17
structuring, 3-10 to 3-11
writing function code, 3-19
general guidelines, 3-1 to 3-2
IVI instrument drivers, 2-5
naming drivers, 3-3, 5-8
programming guidelines, 8-1
steps for programming, 3-2
testing instrument drivers, 3-20
Instrument Driver Development Wizard, 3-3 to 3-9
reviewing generated driver files, 3-6
extended functions and attributes, 3-9
function panels, 3-6
include file, 3-9
source file, 3-8
.sub file, 3-7 to 3-8
using Attribute Editor, 3-9
running preliminary I/O tests, 3-6
selecting template, 3-5 to 3-6
Selection Panel (figure), 3-5

starting, 3-4
worksheet for necessary information, 3-3 to 3-4
instrument driver functions. *See also* IVI Library.
action/status, 1-11
adding to function tree, 5-8 to 5-9
creating multiple classes (example), 5-15 to 5-16
empty tree or class, 5-8
existing tree, 5-9
application functions, 1-12
callbacks, 2-8 to 2-9
close, 1-12
configuration, 1-10
cutting and pasting functions and panels (example), 5-16 to 5-17
data, 1-11
extended functions, in generated driver files, 3-9
get/set/check functions, 2-7 to 2-8
high-level functions
implementation of, 2-6
IVI driver, 2-6
overview, 2-30
VXI*plug&play* drivers, 2-6
initialize, 1-10
naming, 5-8
required. *See* required functions for instrument drivers.
typesafe functions, 2-7
utility, 1-11 to 1-12
instrument driver session functions, IVI
function tree, 11-3
Ivi_Dispose, 11-86
Ivi_LockSession, 11-162 to 11-164
Ivi_SpecificDriverNew, 11-219 to 11-221
Ivi_UnlockSession, 11-228
Ivi_ValidateSession, 11-236
Instrument Driver Support Only command, Build menu, 5-13
instrument drivers. *See also* IVI instrument drivers.
definition, 2-2
files for instrument drivers, 1-3
help information, 7-4
historical evolution, 1-1 to 1-2
operation of, 1-3, 3-19
overview, 1-1
purpose and use, 1-2 to 1-3
Instrument menu
Function Panel Editor, 6-22
Function Tree Editor, 5-9 to 5-11
available options, 5-9
Edit command, 5-10 to 5-11
Load command, 5-9
Unload command, 5-9 to 5-10
instrument simulation. *See* simulation.
instrument state caching, state-caching
instrument-specific attributes
constant name for ID, 2-10
definition, 2-7
Intelligent Virtual Instrument drivers. *See* IVI instrument drivers.
interactive developer interface
definition, 1-5
purpose and use, 1-7
internal design model. *See under* instrument driver architecture.
intrinsic C data types, 3-12
invalidation list functions
function tree, 11-3
Ivi_AddAttributeInvalidation, 11-13 to 11-14
Ivi_DeleteAttributeInvalidation, 11-83
I/O tests, running from Wizard, 3-6

IVI engine
definition, 1-5
interaction with IVI instrument drivers, 2-3 to 2-4
purpose and use, 1-6
IVI instrument drivers, 2-1 to 2-36. *See also* attributes; *instrument driver* entries; IVI Library.
attribute callback functions, 2-22 to 2-25
channels, 2-29 to 2-30
comparison precision, 2-18
configuration entries, 2-35 to 2-36
creating and declaring attributes, 2-9 to 2-13
deferred updates, 2-34
definition, 2-1, 2-2
features, 1-2, 2-1 to 2-2
functions and attribute model, 2-6 to 2-9
high-level driver functions, 2-30
inherent IVI attributes, 2-36 to 2-50
interaction with IVI engine, 2-3 to 2-4
multithread safety, 2-33 to 2-34
operation, 2-3 to 2-4
operation diagram, 2-4
overview, 2-2 to 2-3
programming, 2-5. *See also* instrument driver development.
range checking, 2-31
range tables, 2-13 to 2-18
session callback functions, 2-26 to 2-28
simulation, 2-32 to 2-33
state-caching mechanism, 2-19 to 2-22
status checking, 2-31 to 2-32
IVI Library, 11-1 to 11-246. *See also* instrument driver functions.
attribute creation functions, 11-3
attribute information functions, 11-6 to 11-7
caching/status-checking control functions, 11-5
callback functions, 11-3 to 11-4
channel functions, 11-3
check attribute functions, 11-4 to 11-5
configuration file functions, 11-7
configuration functions, 11-7
default callback functions, 11-6
direct instrument I/O functions, 11-6
dynamic range table functions, 11-5
error information functions, 11-5
error reporting, 11-8 to 11-12
attributes for reporting, 11-8
error macros, 11-9 to 11-12
functions for accessing attribute values, 11-9
function reference
Ivi_AddAttributeInvalidation, 11-13 to 11-14
Ivi_AddAttributeViAddr, 11-15 to 11-17
Ivi_AddAttributeViBoolean, 11-18 to 11-20
Ivi_AddAttributeViInt32, 11-21 to 11-23
Ivi_AddAttributeViReal64, 11-24 to 11-27
Ivi_AddAttributeViSession, 11-28 to 11-30
Ivi_AddAttributeViString, 11-31 to 11-33
Ivi_AddToChannelTable, 11-34
Ivi_Alloc, 11-35 to 11-36
Ivi_AttributeIsCached, 11-37
Ivi_AttributeUpdateIsPending, 11-38
Ivi_BuildChannelTable, 11-39 to 11-41
Ivi_CheckAttributeViAddr, 11-42 to 11-43
Ivi_CheckAttributeViBoolean, 11-42 to 11-43
Ivi_CheckAttributeViInt32, 11-42 to 11-43

Ivi_CheckAttributeViReal64, 11-42 to 11-43
Ivi_CheckAttributeViSession, 11-42 to 11-43
Ivi_CheckAttributeViString, 11-42 to 11-43
Ivi_CheckBooleanRange, 11-44
Ivi_CheckNumericRange, 11-45 to 11-46
Ivi_ClearErrorInfo, 11-47 to 11-48
Ivi_ClearInstrSpecificErrorQueue, 11-49
Ivi_CoerceBoolean, 11-50
Ivi_CoerceChannelName, 11-51 to 11-52
Ivi_CompareWithPrecision, 11-53 to 11-54
Ivi_DefaultBufferedIOCallback, 11-55
Ivi_DefaultCheckCallbackViInt32, 11-56 to 11-57
Ivi_DefaultCheckCallbackViReal64, 11-58 to 11-59
Ivi_DefaultCoerceCallback ViBoolean, 11-60 to 11-61
Ivi_DefaultCoerceCallbackViInt32, 11-62 to 11-63
Ivi_DefaultCoerceCallback ViReal64, 11-64 to 11-65
Ivi_DefaultCompareCallback ViReal64, 11-66 to 11-68
Ivi_DefineClass, 11-69 to 11-70
Ivi_DefineDriver, 11-71 to 11-73
Ivi_DefineHardware, 11-74 to 11-75
Ivi_DefineLogicalName, 11-76 to 11-77
Ivi_DefineVInstr, 11-78 to 11-81
Ivi_DeleteAttribute, 11-82
Ivi_DeleteAttributeInvalidation, 11-83
Ivi_DequeueInstrSpecificError, 11-84 to 11-85
Ivi_Dispose, 11-86
Ivi_DisposeInvalidationList, 11-87
Ivi_DisposeLogicalNamesList, 11-88
Ivi_Free, 11-89
Ivi_FreeAll, 11-90
Ivi_GetAttrComparePrecision, 11-91
Ivi_GetAttributeFlags, 11-92
Ivi_GetAttributeName, 11-93 to 11-94
Ivi_GetAttributeType, 11-95
Ivi_GetAttributeViAddr, 11-96 to 11-98
Ivi_GetAttributeViBoolean, 11-96 to 11-98
Ivi_GetAttributeViInt32, 11-96 to 11-98
Ivi_GetAttributeViReal64, 11-96 to 11-98
Ivi_GetAttributeViSession, 11-96 to 11-98
Ivi_GetAttributeViString, 11-99 to 11-101
Ivi_GetAttrMinMaxViInt32, 11-102 to 11-103
Ivi_GetAttrMinMaxViReal64, 11-104 to 11-105
Ivi_GetAttrRangeTable, 11-106 to 11-107
Ivi_GetErrorInfo, 11-108 to 11-109
Ivi_GetErrorMessage, 11-110
Ivi_GetInvalidationList, 11-111 to 11-112
Ivi_GetIviIniDir, 11-113
Ivi_GetLogicalNamesList, 11-114 to 11-115
Ivi_GetNextCoercionInfo, 11-116 to 11-117
Ivi_GetNthAttribute, 11-118
Ivi_GetNthChannelString, 11-119 to 11-120

Ivi_GetNthLogicalName, 11-121 to 11-122
Ivi_GetNumAttributes, 11-123
Ivi_GetRangeTableNumEntries, 11-124
Ivi_GetSpecificDriverStatusDesc, 11-125 to 11-126
Ivi_GetStoredRangeTablePtr, 11-127 to 11-128
Ivi_GetStringFromTable, 11-129 to 11-130
Ivi_GetUserChannelName, 11-131 to 11-132
Ivi_GetValueFromTable, 11-133 to 11-134
Ivi_GetViInt32EntryFromCmdValue, 11-135 to 11-136
Ivi_GetViInt32EntryFromCoercedVal, 11-137 to 11-138
Ivi_GetViInt32EntryFromIndex, 11-139 to 11-140
Ivi_GetViInt32EntryFromString, 11-141 to 11-142
Ivi_GetViInt32EntryFromValue, 11-143 to 11-144
Ivi_GetViReal64EntryFromCmdValue, 11-145 to 11-146
Ivi_GetViReal64EntryFromCoercedVal, 11-147 to 11-148
Ivi_GetViReal64EntryFromIndex, 11-149 to 11-150
Ivi_GetViReal64EntryFromString, 11-151 to 11-152
Ivi_GetViReal64EntryFromValue, 11-153 to 11-154
Ivi_InstrSpecificErrorQueueSize, 11-155 to 11-156
Ivi_InterchangeCheck, 11-157
Ivi_InvalidateAllAttributes, 11-158
Ivi_InvalidateAttribute, 11-159 to 11-160
Ivi_IOSession, 11-161
Ivi_LockSession, 11-162 to 11-164
Ivi_NeedToCheckStatus, 11-165 to 11-166
Ivi_QueryInstrStatus, 11-167
Ivi_QueueInstrSpecificError, 11-168 to 11-169
Ivi_RangeChecking, 11-170
Ivi_RangeTableFree, 11-171
Ivi_RangeTableNew, 11-172 to 11-174
Ivi_ReadInstrData, 11-175 to 11-176
Ivi_ReadToFile, 11-177 to 11-178
Ivi_RestrictAttrToChannels, 11-179 to 11-180
Ivi_SetAttrCheckCallbackViAddr, 11-181 to 11-183
Ivi_SetAttrCheckCallbackViBoolean, 11-181 to 11-183
Ivi_SetAttrCheckCallbackViInt32, 11-181 to 11-183
Ivi_SetAttrCheckCallbackViReal64, 11-181 to 11-183
Ivi_SetAttrCheckCallbackViSession, 11-181 to 11-183
Ivi_SetAttrCheckCallbackViString, 11-181 to 11-183
Ivi_SetAttrCoerceCallbackViAddr, 11-184 to 11-186
Ivi_SetAttrCoerceCallbackViBoolean, 11-184 to 11-186
Ivi_SetAttrCoerceCallbackViInt32, 11-184 to 11-186
Ivi_SetAttrCoerceCallbackViReal64, 11-184 to 11-186
Ivi_SetAttrCoerceCallbackViSession, 11-184 to 11-186
Ivi_SetAttrCoerceCallbackViString, 11-184 to 11-186
Ivi_SetAttrCompareCallbackViAddr, 11-187 to 11-189
Ivi_SetAttrCompareCallbackViBoolean, 11-187 to 11-189

Ivi_SetAttrCompareCallback ViInt32, 11-187 to 11-189
Ivi_SetAttrCompareCallback ViReal64, 11-187 to 11-189
Ivi_SetAttrCompareCallback ViSession, 11-187 to 11-189
Ivi_SetAttrCompareCallback ViString, 11-187 to 11-189
Ivi_SetAttrComparePrecision, 11-190 to 11-191
Ivi_SetAttributeFlags, 11-192 to 11-193
Ivi_SetAttributeViAddr, 11-194 to 11-197
Ivi_SetAttributeViBoolean, 11-194 to 11-197
Ivi_SetAttributeViInt32, 11-194 to 11-197
Ivi_SetAttributeViReal64, 11-194 to 11-197
Ivi_SetAttributeViSession, 11-194 to 11-197
Ivi_SetAttributeViString, 11-194 to 11-197
Ivi_SetAttrRangeTableCallback, 11-198 to 11-199
Ivi_SetAttrReadCallbackViAddr, 11-200 to 11-202
Ivi_SetAttrReadCallbackViBoolean, 11-200 to 11-202
Ivi_SetAttrReadCallbackViInt32, 11-200 to 11-202
Ivi_SetAttrReadCallbackViReal64, 11-200 to 11-202
Ivi_SetAttrReadCallbackViSession, 11-200 to 11-202
Ivi_SetAttrReadCallbackViString, 11-200 to 11-202
Ivi_SetAttrWriteCallbackViAddr, 11-203 to 11-205
Ivi_SetAttrWriteCallbackViBoolean, 11-203 to 11-205
Ivi_SetAttrWriteCallbackViInt32, 11-203 to 11-205
Ivi_SetAttrWriteCallbackViReal64, 11-203 to 11-205
Ivi_SetAttrWriteCallbackViSession, 11-203 to 11-205
Ivi_SetAttrWriteCallbackViString, 11-203 to 11-205
Ivi_SetErrorInfo, 11-206 to 11-208
Ivi_SetIviIniDir, 11-209
Ivi_SetNeedToCheckStatus, 11-210 to 11-211
Ivi_SetRangeTableEnd, 11-212
Ivi_SetRangeTableEntry, 11-213 to 11-214
Ivi_SetStoredRangeTablePtr, 11-215 to 11-216
Ivi_SetValInStringCallback, 11-217
Ivi_Simulating, 11-218
Ivi_SpecificDriverNew, 11-219 to 11-221
Ivi_Spying, 11-222
Ivi_UndefClass, 11-223
Ivi_UndefDriver, 11-224
Ivi_UndefHardware, 11-225
Ivi_UndefLogicalName, 11-226
Ivi_UndefVInstr, 11-227
Ivi_UnlockSession, 11-228
Ivi_Update, 11-229 to 11-231
Ivi_UseSpecificSimulation, 11-232
Ivi_ValidateAttrForChannel, 11-233 to 11-234
Ivi_ValidateRangeTable, 11-235
Ivi_ValidateSession, 11-236
Ivi_WriteFromFile, 11-237 to 11-238
Ivi_WriteInstrData, 11-239
Ivi_WriteRunTimeDefinesToFile, 11-240

function tree
class or panel, 11-3 to 11-8
description of top-level classes, 11-7 to 11-8
structure, 11-2
get attribute functions, 11-4
helper functions, 11-6 to 11-7
inherent attribute accessor functions, 11-6
instrument driver session functions, 11-3
invalidation list functions, 11-3
logical names functions, 11-7
memory allocation functions, 11-6
range table functions, 11-5
range table pointer functions, 11-5
run-time configuration functions, 11-7
set attribute functions, 11-4
set/get/check attribute functions, 11-4 to 11-7
status codes, 11-240 to 11-246
common errors and warnings (table), 11-245 to 11-246
default values of defined constants (table), 11-242
IVI errors and warnings (table), 11-242 to 11-245
ranges (table), 11-241
VISA errors and warnings (table), 11-246
string callback functions, 11-6
string/value tables functions, 11-6
value manipulation functions, 11-6
Ivi_AddAttributeInvalidation function, 11-13 to 11-14
Ivi_AddAttributeViAddr function, 11-15 to 11-17
Ivi_AddAttributeViBoolean function, 11-18 to 11-20
Ivi_AddAttributeViInt32 function, 11-21 to 11-23
Ivi_AddAttributeViReal64 function, 11-24 to 11-27
Ivi_AddAttributeViSession function, 11-28 to 11-30
Ivi_AddAttributeViString function, 11-31 to 11-33
Ivi_AddToChannelTable function, 11-34
Ivi_Alloc function, 11-35 to 11-36
IVI_ATTR_ATTRIBUTE_CAPABILITIES, 2-38
IVI_ATTR_ATTR_SPY, 2-48 to 2-49
IVI_ATTR_BUFFERED_IO_CALLBACK, 2-38
IVI_ATTR_CACHE, 2-38 to 2-39
IVI_ATTR_CHECK_STATUS_CALLBACK, 2-39
IVI_ATTR_CLASS_MAJOR_VERSION, 2-39
IVI_ATTR_CLASS_MINOR_VERSION, 2-39
IVI_ATTR_CLASS_PREFIX, 2-40
IVI_ATTR_CLASS_REVISION, 2-40
IVI_ATTR_DEFER_UPDATE, 2-40 to 2-41
IVI_ATTR_DRIVER_MAJOR_VERSION, 2-41
IVI_ATTR_DRIVER_MINOR_VERSION, 2-41
IVI_ATTR_DRIVER_REVISION, 2-41
IVI_ATTR_DRIVER_SETUP, 2-41
IVI_ATTR_ENGINE_MAJOR_VERSION, 2-42
IVI_ATTR_ENGINE_MINOR_VERSION, 2-42
IVI_ATTR_ENGINE_REVISION, 2-42
IVI_ATTR_ERROR_ELABORATION, 2-42
IVI_ATTR_FUNCTION_CAPABILITIES, 2-42
IVI_ATTR_GROUP_CAPABILITIES, 2-43
IVI_ATTR_INTERCHANGE_CHECK, 2-43
IVI_ATTR_I/O_SESSION, 2-43
IVI_ATTR_LOGICAL_NAME, 2-44
IVI_ATTR_MODULE_PATHNAME, 2-44
IVI_ATTR_NUM_CHANNELS, 2-44

IVI_ATTR_OPC_CALLBACK, 2-45
IVI_ATTR_PRIMARY_ERROR, 2-45
IVI_ATTR_QUERY_INSTR_STATUS, 2-45 to 2-46
IVI_ATTR_RANGE_CHECK, 2-46
IVI_ATTR_RECORD_COERCIONS, 2-46 to 2-47
IVI_ATTR_RESOURCE_DESCRIPTOR, 2-47
IVI_ATTR_RETURN_DEFERRED_VALUES, 2-47
IVI_ATTR_SECONDARY_ERROR, 2-47
IVI_ATTR_SIMULATE, 2-48
IVI_ATTR_SPECIFIC_PREFIX, 2-48
IVI_ATTR_SUPPORTS_WR_BUF_OPER_MODE, 2-49
IVI_ATTR_UPDATING_VALUES, 2-49
IVI_ATTR_USE_SPECIFIC_SIMULATION, 2-49 to 2-50
IVI_ATTR_VISA_RM_SESSION, 2-50
Ivi_AttributeIsCached function, 11-37
Ivi_AttributeUpdateIsPending function, 11-38
Ivi_BuildChannelTable function, 11-39 to 11-41
Ivi_CheckAttribute functions
  optionFlags parameter, 2-8
  purpose and use, 2-7
Ivi_CheckAttributeViAddr function, 11-42 to 11-43
Ivi_CheckAttributeViBoolean function, 11-42 to 11-43
Ivi_CheckAttributeViInt32 function, 11-42 to 11-43
Ivi_CheckAttributeViReal64 function, 11-42 to 11-43
Ivi_CheckAttributeViSession function, 11-42 to 11-43
Ivi_CheckAttributeViString function, 11-42 to 11-43
Ivi_CheckBooleanRange function, 11-44
Ivi_CheckNumericRange function, 11-45 to 11-46
Ivi_ClearErrorInfo function, 11-47 to 11-48
Ivi_ClearInstrSpecificErrorQueue function, 11-49
Ivi_CoerceBoolean function, 11-50
Ivi_CoerceChannelName function, 11-51 to 11-52
Ivi_CompareWithPrecision function, 11-53 to 11-54
Ivi_DefaultBufferedIOCallback function, 11-55
Ivi_DefaultCheckCallbackViInt32 function, 11-56 to 11-57
Ivi_DefaultCheckCallbackViReal64 function, 11-58 to 11-59
Ivi_DefaultCoerceCallbackViBoolean function, 11-60 to 11-61
Ivi_DefaultCoerceCallbackViInt32 function, 11-62 to 11-63
Ivi_DefaultCoerceCallbackViReal64 function, 11-64 to 11-65
Ivi_DefaultCompareCallbackViReal64 function, 11-66 to 11-68
Ivi_DefineClass function, 11-69 to 11-70
Ivi_DefineDriver function, 11-71 to 11-73
Ivi_DefineHardware function, 11-74 to 11-75
Ivi_DefineLogicalName function, 11-76 to 11-77
Ivi_DefineVInstr function, 11-78 to 11-81
Ivi_DeleteAttribute function, 11-82
Ivi_DeleteAttributeInvalidation function, 11-83
Ivi_DequeueInstrSpecificError function, 11-84 to 11-85
Ivi_Dispose function, 11-86
Ivi_DisposeInvalidationList function, 11-87
Ivi_DisposeLogicalNamesList function, 11-88
Ivi_Free function, 11-89
Ivi_FreeAll function, 11-90
Ivi_GetAttrComparePrecision function, 11-91

Ivi_GetAttribute functions
not used in application programs, 2-5
optionFlags parameter, 2-8
Ivi_GetAttributeFlags function, 11-92
Ivi_GetAttributeName function, 11-93 to 11-94
Ivi_GetAttributeType function, 11-95
Ivi_GetAttributeViAddr function, 11-96 to 11-98
Ivi_GetAttributeViBoolean function, 11-96 to 11-98
Ivi_GetAttributeViInt32 function, 11-96 to 11-98
Ivi_GetAttributeViReal64 function, 11-96 to 11-98
Ivi_GetAttributeViSession function, 11-96 to 11-98
Ivi_GetAttributeViString function, 11-99 to 11-101
Ivi_GetAttrMinMaxViInt32 function, 11-102 to 11-103
Ivi_GetAttrMinMaxViReal64 function, 11-104 to 11-105
Ivi_GetAttrRangeTable function, 11-106 to 11-107
Ivi_GetErrorInfo function, 11-108 to 11-109
Ivi_GetErrorMessage function, 11-110
Ivi_GetInvalidationList function, 11-111 to 11-112
Ivi_GetIviIniDir function, 11-113
Ivi_GetLogicalNamesList function, 11-114 to 11-115
Ivi_GetNextCoercionInfo function, 11-116 to 11-117
Ivi_GetNthAttribute function, 11-118
Ivi_GetNthChannelString function, 11-119 to 11-120
Ivi_GetNthLogicalName function, 11-121 to 11-122
Ivi_GetNumAttributes function, 11-123
Ivi_GetRangeTableNumEntries function, 11-124
Ivi_GetSpecificDriverStatusDesc function, 11-125 to 11-126
Ivi_GetStoredRangeTablePtr function, 11-127 to 11-128
Ivi_GetStringFromTable function, 11-129 to 11-130
Ivi_GetUserChannelName function, 11-131 to 11-132
Ivi_GetValueFromTable function, 11-133 to 11-134
Ivi_GetViInt32EntryFromCmdValue function, 11-135 to 11-136
Ivi_GetViInt32EntryFromCoercedVal function, 11-137 to 11-138
Ivi_GetViInt32EntryFromIndex function, 11-139 to 11-140
Ivi_GetViInt32EntryFromString function, 11-141 to 11-142
Ivi_GetViInt32EntryFromValue function, 11-143 to 11-144
Ivi_GetViReal64EntryFromCmdValue function, 11-145 to 11-146
Ivi_GetViReal64EntryFromCoercedVal function, 11-147 to 11-148
Ivi_GetViReal64EntryFromIndex function, 11-149 to 11-150
Ivi_GetViReal64EntryFromString function, 11-151 to 11-152
Ivi_GetViReal64EntryFromValue function, 11-153 to 11-154
ivi.ini file, 2-35
Ivi_InstrSpecificErrorQueueSize function, 11-155 to 11-156
Ivi_InterchangeCheck function, 11-157
Ivi_InvalidateAllAttributes function, 11-158
Ivi_InvalidateAttribute function, 11-159 to 11-160
Ivi_IOSession function, 11-161
Ivi_LockSession function, 11-162 to 11-164
Ivi_NeedToCheckStatus function, 11-165 to 11-166
Ivi_QueryInstrStatus function, 11-167

Ivi_QueueInstrSpecificError function, 11-168 to 11-169
Ivi_RangeChecking function, 11-170
Ivi_RangeTableFree function, 11-171
Ivi_RangeTableNew function, 11-172 to 11-174
Ivi_ReadInstrData function, 11-175 to 11-176
Ivi_ReadToFile function, 11-177 to 11-178
Ivi_RestrictAttrToChannels function, 11-179 to 11-180
Ivi_SetAttrCheckCallbackViAddr function, 11-181 to 11-183
Ivi_SetAttrCheckCallbackViBoolean function, 11-181 to 11-183
Ivi_SetAttrCheckCallbackViInt32 function, 11-181 to 11-183
Ivi_SetAttrCheckCallbackViReal64 function, 11-181 to 11-183
Ivi_SetAttrCheckCallbackViSession function, 11-181 to 11-183
Ivi_SetAttrCheckCallbackViString function, 11-181 to 11-183
Ivi_SetAttrCoerceCallbackViAddr function, 11-184 to 11-186
Ivi_SetAttrCoerceCallbackViBoolean function, 11-184 to 11-186
Ivi_SetAttrCoerceCallbackViInt32 function, 11-184 to 11-186
Ivi_SetAttrCoerceCallbackViReal64 function, 11-184 to 11-186
Ivi_SetAttrCoerceCallbackViSession function, 11-184 to 11-186
Ivi_SetAttrCoerceCallbackViString function, 11-184 to 11-186
Ivi_SetAttrCompareCallbackViAddr function, 11-187 to 11-189
Ivi_SetAttrCompareCallbackViBoolean function, 11-187 to 11-189
Ivi_SetAttrCompareCallbackViInt32 function, 11-187 to 11-189
Ivi_SetAttrCompareCallbackViReal64 function, 11-187 to 11-189
Ivi_SetAttrCompareCallbackViSession function, 11-187 to 11-189
Ivi_SetAttrCompareCallbackViString function, 11-187 to 11-189
Ivi_SetAttrComparePrecision function, 11-190 to 11-191
Ivi_SetAttribute functions
  not used in application programs, 2-5
  optionFlags parameter, 2-8
Ivi_SetAttributeFlags function, 11-192 to 11-193
Ivi_SetAttributeViAddr function, 11-194 to 11-197
Ivi_SetAttributeViBoolean function, 11-194 to 11-197
Ivi_SetAttributeViInt32 function, 11-194 to 11-197
Ivi_SetAttributeViReal64 function, 11-194 to 11-197
Ivi_SetAttributeViSession function, 11-194 to 11-197
Ivi_SetAttributeViString function, 11-194 to 11-197
Ivi_SetAttrRangeTableCallback function, 11-198 to 11-199
Ivi_SetAttrReadCallbackViAddr function, 11-200 to 11-202
Ivi_SetAttrReadCallbackViBoolean function, 11-200 to 11-202
Ivi_SetAttrReadCallbackViInt32 function, 11-200 to 11-202
Ivi_SetAttrReadCallbackViReal64 function, 11-200 to 11-202
Ivi_SetAttrReadCallbackViSession function, 11-200 to 11-202
Ivi_SetAttrReadCallbackViString function, 11-200 to 11-202
Ivi_SetAttrWriteCallbackViAddr function, 11-203 to 11-205
Ivi_SetAttrWriteCallbackViBoolean function, 11-203 to 11-205

Ivi_SetAttrWriteCallbackViInt32 function, 11-203 to 11-205
Ivi_SetAttrWriteCallbackViReal64 function, 11-203 to 11-205
Ivi_SetAttrWriteCallbackViSession function, 11-203 to 11-205
Ivi_SetAttrWriteCallbackViString function, 11-203 to 11-205
Ivi_SetErrorInfo function, 11-206 to 11-208
Ivi_SetIviIniDir function, 11-209
Ivi_SetNeedToCheckStatus function, 11-210 to 11-211
Ivi_SetRangeTableEnd function, 11-212
Ivi_SetRangeTableEntry function, 11-213 to 11-214
Ivi_SetStoredRangeTablePtr function, 11-215 to 11-216
Ivi_SetValInStringCallback function, 11-217
Ivi_Simulating function, 11-218
Ivi_SpecificDriverNew function, 11-219 to 11-221
Ivi_Spying function, 11-222
Ivi_UndefClass function, 11-223
Ivi_UndefDriver function, 11-224
Ivi_UndefHardware function, 11-225
Ivi_UndefLogicalName function, 11-226
Ivi_UndefVInstr function, 11-227
Ivi_UnlockSession function, 11-228
Ivi_Update function, 11-229 to 11-231
Ivi_UseSpecificSimulation, 11-232
Ivi_UseSpecificSimulation function, 11-232
IVI_VAL_ALWAYS_CACHE flag, 2-11
IVI_VAL_COERCEABLE_ONLY_BY_INSTR flag, 2-12
IVI_VAL_COERCED range tables, 2-14
IVI_VAL_DISCRETE range tables, 2-14
IVI_VAL_DONT_CHECK_STATUS flag, 2-13
IVI_VAL_DONT_RETURN_DEFERRED_VALUE flag, 2-12
IVI_VAL_FLUSH_ON_WRITE flag, 2-12
IVI_VAL_HIDDEN flag, 2-11
IVI_VAL_MULTI_CHANNEL flag, 2-12
IVI_VAL_NEVER_CACHE flag, 2-11
IVI_VAL_NO_DEFERRED_UPDATE flag, 2-11
IVI_VAL_NOT_READABLE flag, 2-11
IVI_VAL_NOT_SUPPORTED flag, 2-11
IVI_VAL_NOT_USER_READABLE flag, 2-11
IVI_VAL_NOT_USER_WRITABLE flag, 2-11
IVI_VAL_NOT_WRITABLE flag, 2-11
IVI_VAL_RANGED range tables, 2-14
IVI_VAL_USER_CALLBACKS_FOR_SIMULATION flag, 2-12
IVI_VAL_WAIT_FOR_OPC_AFTER_WRITES flag, 2-12
IVI_VAL_WAIT_FOR_OPC_BEFORE_READS flag, 2-12
Ivi_ValidateAttrForChannel function, 11-233 to 11-234
Ivi_ValidateRangeTable function, 11-235
Ivi_ValidateSession function, 11-236
Ivi_WriteFromFile function, 11-237 to 11-238
Ivi_WriteInstrData function, 11-239
Ivi_WriteRunTimeDefinesToFile function, 11-240

L

Load command, Instrument menu, 5-9
lock/unlock session functions
    Ivi_LockSession, 11-162 to 11-164
    Ivi_UnlockSession, 11-228
    overview, 1-12
    programming considerations, 2-5
logical name functions
    function tree, 11-7
    Ivi_DisposeLogicalNamesList, 11-88

Ivi_GetLogicalNamesList, 11-114 to 11-115
Ivi_GetNthLogicalName, 11-121 to 11-122

M

manual. *See* documentation.
memory allocation functions
function tree, 11-6
Ivi_Alloc, 11-35 to 11-36
Ivi_Free, 11-89
Ivi_FreeAll, 11-90
Message command, Create menu, 6-21
message controls, 6-21
meta data types, 3-13 to 3-14
Any Array, 3-13
Any Type, 3-13 to 3-14
definition, 3-13
Numeric Array, 3-13
Var Args, 3-14
Microsoft Windows DLLs. *See* Windows DLLs.
models for instrument drivers. *See* instrument driver architecture.
Move buttons, Edit Driver Attributes dialog box, 4-5
moving controls, 6-25
multithread safety, IVI instrument drivers, 2-33 to 2-34

N

names
files for instrument drivers, 1-3
functions for instrument drivers, 5-8
instrument drivers, 3-3, 5-8
New command, File menu, 5-1
Numeric Array data type, 3-13
Numeric command, Create menu, 6-16 to 6-18
numeric controls
control label, 6-17
Create Numeric Control dialog box, 6-16 to 6-18
creating, 6-16 to 6-18
data type, 6-17
default value, 6-17
definition, 6-16
display format, 6-18
Edit Value Set dialog box, 6-18
increment and decrement values, 6-18
maximum value, 6-18
minimum value, 6-18
parameter position, 6-17

O

Object Description Language (.odl) file, generating, 5-13
ODL file, generating, 5-13
online help. *See* help information.
Open command, File menu, 5-1
Operate Function Panel command, Options menu, 6-24
operation complete callback, 2-26
Options menu
Function Panel Editor, 6-23 to 6-24
Data Types command, 6-23 to 6-24
Default Panel Size command, 6-24
Edit Data Type List dialog box, 6-23 to 6-24
Edit Function Tree command, 6-24
Operate Function Panel command, 6-24
Panels Movable command, 6-24
Toggle Scroll Bars command, 6-24
Toolbar command, 6-24
Function Tree Editor, 5-12 to 5-14
Create DLL Project command, 5-13
Generate DLL Make File command, 5-12

Generate Documentation command, 5-12
Generate Function Prototypes command, 5-12
Generate ODL File command, 5-13
Generate Windows Help command, 5-12
Help Style command, 5-12
Transfer Window Help to Function Help command, 5-12
VXIplug&playStyle command, 5-13 to 5-14
Output command, Create menu, 6-19
output controls, 6-19
control label, 6-19
control width, 6-19
Create Output Control dialog box, 6-19
data type, 6-19
definition, 6-19
display format, 6-19
parameter position, 6-19

P

Panels Movable command, Options menu, 6-24
parameters for instrument drivers. *See also* data types.
defining, 3-11
input and output parameters, 3-17
Paste Above command, Edit menu, 5-5
Paste Below command, Edit menu, 5-5
Paste command, Edit menu, 6-4
pasting
controls (example), 6-32 to 6-33
functions and panels (example), 5-16 to 5-17
help text, 7-9 to 7-11
precision of floating point numbers, 2-18. *See also* comparison precision functions.
prefix for instrument driver names, 3-3
*Prefix*_CheckAttribute functions, 2-7
*Prefix*_close function, 9-10 to 9-11
*Prefix*_error_message function, 9-20 to 9-21
*Prefix*_error_query function, 9-17 to 9-19
*Prefix*_GetAttribute functions, 2-7
*Prefix*_init function
programming considerations, 2-5
purpose and use, 9-3 to 9-5
*Prefix*_initWithOptions function
programming considerations, 2-5
purpose and use, 9-6 to 9-7
*Prefix*_IviClose function, 9-12
*Prefix*_IviInit function, 9-8 to 9-9
*Prefix*_LockSession function, 2-5
*Prefix*_reset function, 9-13 to 9-14
*Prefix*_revision_query function, 9-22 to 9-23
*Prefix*_self_test function, 9-15 to 9-16
*Prefix*_SetAttribute functions, 2-7
*Prefix*_UnLockSession function, 2-5
programmatic developer interface
definition, 1-5
purpose and use, 1-7
programming instrument drivers. *See* instrument driver development.

R

range checking
definition, 2-3
purpose and use, 2-31
range table functions
function tree, 11-5
Ivi_GetAttrRangeTable, 11-106 to 11-107
Ivi_GetRangeTableNumEntries, 11-124
Ivi_GetViInt32EntryFromCmdValue, 11-135 to 11-136
Ivi_GetViInt32EntryFromCoercedVal, 11-137 to 11-138
Ivi_GetViInt32EntryFromIndex, 11-139 to 11-140

Ivi_GetViInt32EntryFromString, 11-141 to 11-142
Ivi_GetViInt32EntryFromValue, 11-143 to 11-144
Ivi_GetViReal64EntryFromCmdValue, 11-145 to 11-146
Ivi_GetViReal64EntryFromCoercedVal, 11-147 to 11-148
Ivi_GetViReal64EntryFromIndex, 11-149 to 11-150
Ivi_GetViReal64EntryFromString, 11-151 to 11-152
Ivi_GetViReal64EntryFromValue, 11-153 to 11-154
Ivi_ValidateRangeTable, 11-235
purpose and use, 2-25
range table pointer functions. *See also* dynamic range table functions.
Ivi_GetStoredRangeTablePtr, 11-127 to 11-128
Ivi_SetStoredRangeTablePtr, 11-215 to 11-216
range tables, 2-13 to 2-18
coerced range table example, 2-16
default check and coerce callbacks, 2-17 to 2-18
discrete range table example, 2-15
IVI_VAL_COERCED, 2-14
IVI_VAL_DISCRETE, 2-14
IVI_VAL_RANGED, 2-14
ranged range table example, 2-16
static and dynamic, 2-17
structures, 2-13 to 2-15
Range Tables button, Edit Driver Attributes dialog box, 4-5
Range Tables dialog box, Attribute Editor, 4-10 to 4-13
Edit Range Table dialog box, 4-11 to 4-13
illustration, 4-10
ranged range tables
example, 2-16
IVI_VAL_RANGED, 2-14
read callback functions
Ivi_SetAttrReadCallbackViAddr, 11-200 to 11-202
Ivi_SetAttrReadCallbackViBoolean, 11-200 to 11-202
Ivi_SetAttrReadCallbackViInt32, 11-200 to 11-202
Ivi_SetAttrReadCallbackViReal64, 11-200 to 11-202
Ivi_SetAttrReadCallbackViSession, 11-200 to 11-202
Ivi_SetAttrReadCallbackViString, 11-200 to 11-202
purpose and use, 2-22 to 2-23
Reattach Program command, Edit Instrument dialog box, 5-10 to 5-11
Replace command, Edit menu
Function Panel Editor, 6-6
Function Tree Editor, 5-5
required functions for instrument drivers. *See also* specific *Prefix* functions.
categories of functions, 9-1 to 9-2
list of functions, 2-6, 3-18
overview, 9-2
reset function
definition, 1-11
*Prefix*_reset, 9-13 to 9-14
Return Value command, Create menu, 6-20
return value controls, 6-20
control label, 6-20
control width, 6-20
Create Return Value Control dialog box, 6-20
data type, 6-20
definition, 6-20
display format, 6-20
return values, instrument driver functions, 3-17

reviewing generated driver files. *See* generated driver files, reviewing.
revision query function
  definition, 1-11
  *Prefix*_revision_query, 9-22 to 9-23
Ring command, Create menu, 6-14 to 6-16
ring controls. *See also* Edit Label/Value Pairs dialog box.
  adding label and value to ring control list, 6-15 to 6-16
  control label, 6-14
  control width, 6-15
  Create Ring Control dialog box, 6-14 to 6-15
  creating, 6-14 to 6-16
  data type, 6-14
  default value, 6-14
  definition, 6-14
  Edit Label/Value Pairs dialog box, 6-15 to 6-16
  parameter position, 6-14
run-time configuration functions
  function tree, 11-7
  Ivi_DefineClass, 11-69 to 11-70
  Ivi_DefineDriver, 11-71 to 11-73
  Ivi_DefineHardware, 11-74 to 11-75
  Ivi_DefineLogicalName, 11-76 to 11-77
  Ivi_DefineVInstr, 11-78 to 11-81
  Ivi_UndefClass, 11-223
  Ivi_UndefDriver, 11-224
  Ivi_UndefHardware, 11-225
  Ivi_UndefLogicalName, 11-226
  Ivi_UndefVInstr, 11-227
  Ivi_WriteRunTimeDefinesToFile, 11-240

S

Select Attribute Constant dialog box, 3-7
self-test function (*Prefix*_self_test function), 9-15 to 9-16
session callback functions, 2-26 to 2-28
  buffered I/O callback, 2-27 to 2-28
  check status, 2-26 to 2-27
  definition, 1-12
  instruments without error queues, 2-27
  operation complete callback, 2-26
session info attributes, 2-36
session I/O attributes, 2-36
sessions, initializing, 2-5
set attribute functions
  function tree, 11-4
  Ivi_SetAttributeViAddr, 11-194 to 11-197
  Ivi_SetAttributeViBoolean, 11-194 to 11-197
  Ivi_SetAttributeViInt32, 11-194 to 11-197
  Ivi_SetAttributeViReal64, 11-194 to 11-197
  Ivi_SetAttributeViSession, 11-194 to 11-197
  Ivi_SetAttributeViString, 11-194 to 11-197
set check callback functions. *See* check callback functions.
set coerce callback functions. *See* coerce callback functions.
set compare callback functions. *See* compare callback functions.
set read callback functions. *See* callback functions.
set write callback functions. *See* callback functions.
set/get/check attribute functions, IVI, 11-4 to 11-7
Show Info command, Edit Instrument dialog box, 5-10
simulation
  definition, 2-3
  overview, 1-2

preventing instrument I/O during (note), 2-33
purpose and use, 2-32 to 2-33
Slide command, Create menu, 6-9 to 6-11
slide controls
adding labels and values to slide control list, 6-11
control label, 6-10
Create Slide Control dialog box, 6-9 to 6-10, 6-28
creating, 6-9 to 6-11, 6-28
data type, 6-10
default value, 6-10
definition, 6-9
Edit Label/Value Pairs dialog box, 6-10 to 6-11, 6-28, 6-31
parameter position, 6-10
source file, generated by Instrument Driver Development Wizard
categories of functions in, 3-8
reviewing, 3-8
state-caching, IVI, 2-19 to 2-22. *See also* caching/status-checking control functions.
definition, 2-3
enabling and disabling, 2-21 to 2-22
initial instrument state, 2-19
instrument coerce values, 2-21
invalidation of attributes
by changing one attribute, 2-20
by changing two attributes, 2-20
overview, 1-2, 2-19
setting/getting values of two attributes with one command, 2-20 to 2-21
special cases, 2-19
static range tables, 2-17
status checking
check status callback, 2-26 to 2-27
purpose and use, 2-31 to 2-32
status checking functions, 1-11. *See also* caching/status-checking control functions.
status codes, IVI Library, 11-240 to 11-246
common errors and warnings (table), 11-245 to 11-246
default values of defined constants (table), 11-242
IVI errors and warnings (table), 11-242 to 11-245
ranges (table), 11-241
VISA errors and warnings (table), 11-246
status query, definition, 2-3
string callback function (Ivi_SetValInStringCallback), 11-217
string/value table functions
function tree, 11-6
Ivi_GetStringFromTable, 11-129 to 11-130
Ivi_GetValueFromTable, 11-133 to 11-134
structures, for range tables, 2-13 to 2-15
.sub file
definition, 1-3
generated by Instrument Driver Development Wizard, reviewing, 3-7 to 3-8
subroutine interface
definition, 1-5
purpose and use, 1-6 to 1-7

T

technical support, A-1 to A-2
telephone and fax support numbers, A-2
template for instrument driver, selecting, 3-5 to 3-6
testing instrument drivers
procedure, 3-20
running preliminary I/O tests from Wizard, 3-6
To Definition command, Tools menu
Function Panel Editor, 6-22
Function Tree Editor, 5-11

Toggle Scroll Bars command, Options menu, 6-24
Toolbar command, Options menu, 6-24
Tools menu
- Function Panel Editor
  - Create IVI Instrument Driver command, 3-4, 6-22
  - Edit Instrument Attributes command, 6-22
  - Enable Auto Replace command, 6-22
  - Generate Source for Function Tree command, 6-22
  - Go to Declaration command, 6-22
  - Go to Definition command, 6-22
- Function Tree Editor
  - Create IVI Instrument Driver command, 5-11
  - Edit Instrument Attributes command, 5-11
  - Enable Auto Replace command, 5-11
  - Generate IVI C++ Wrapper command, 5-11
  - Generate Source for Function Tree command, 5-11
  - Go to Declaration command, 5-11
  - Go to Definition command, 5-11
- Help Editor dialog box, 7-4

Transfer Window Help to Function Help command, Options menu, 5-12
typesafe functions, 2-7

U

Unload command, Instrument menu, 5-9 to 5-10
unlock session functions. *See* lock/unlock session functions.
user options attributes, 2-36
user-defined data types, 3-14 to 3-15
- array data types, 3-15
- creating, 3-14 to 3-15
- VISA data types, 3-16 to 3-17

utility functions
- *Prefix*_error_message, 9-20 to 9-21
- *Prefix*_error_query, 9-17 to 9-19
- *Prefix*_reset, 9-13 to 9-14
- *Prefix*_revision_query, 9-22 to 9-23
- *Prefix*_self_test, 9-15 to 9-16
- types of, 1-11 to 1-12

V

value manipulation functions
- function tree, 11-6
- Ivi_CheckBooleanRange, 11-44
- Ivi_CheckNumericRange, 11-45 to 11-46
- Ivi_CoerceBoolean, 11-50
- Ivi_CompareWithPrecision, 11-53 to 11-54

value table functions. *See* string/value table functions.
Var Args data type, 3-14
version info attributes, 2-37
VIBoolean[ ] data type (table), 3-16
VIBoolean data type (table), 3-16
VIChar[ ] data type (table), 3-16
viCheckAlloc macro, 11-10
viCheckErr macro, 11-10
viCheckErrElab macro, 11-10
viCheckParm macro, 11-11
View menu, Function Panel Editor, 6-21
VIInt16[ ] data type (table), 3-16
VIInt16 data type (table), 3-16
VIInt32[ ] data type (table), 3-16
VIInt32 data type (table), 3-16
VIReal64[ ] data type (table), 3-16
VIReal64 data type (table), 3-16
VIRsrc data type (table), 3-16
virtual channel names, 2-29

Virtual Instrumentation Software Architecture. *See* VISA I/O interface.
VISA data types
how to use, 3-16
list of types (table), 3-16
purpose and use, 3-16 to 3-17
VISA I/O interface
definition, 1-5
errors and warnings (table), 11-246
purpose and use, 1-6
VISession data type (table), 3-16
VIStatus data type (table), 3-16
VXI*plug&play* instrument driver
high-level functions, 2-6
historical evolution of instrument drivers, 1-1 to 1-2
VXIplug&playStyle command, Options menu, 5-13 to 5-14
default settings
Advanced dialog box, 5-14
Create Distribution Kit dialog box, 5-14
Create Dynamic Link Library dialog box, 5-13
Instrument Driver Support Only command, 5-13
effects on DLL project, 5-13

W

Window Help command, Edit menu, 6-7
Window menu
Function Panel Editor, 6-22
Function Tree Editor, 5-12
Help Editor dialog box, 7-4
Windows DLLs
Create DLL Project command, Options menu, 5-13
Generate DLL Make Files command, Options menu, 5-12
VXIplug&playStyle command, Options menu, 5-13 to 5-14
wrappers for IVI Library functions, 9-2
write callback functions
Ivi_SetAttrWriteCallbackViAddr, 11-203 to 11-205
Ivi_SetAttrWriteCallbackViBoolean, 11-203 to 11-205
Ivi_SetAttrWriteCallbackViInt32, 11-203 to 11-205
Ivi_SetAttrWriteCallbackViReal64, 11-203 to 11-205
Ivi_SetAttrWriteCallbackViSession, 11-203 to 11-205
Ivi_SetAttrWriteCallbackViString, 11-203 to 11-205
purpose and use, 2-23
write/read instrument data functions, 1-12
writing instrument drivers. *See* instrument driver development.