LabWindows®/CVI

Instrument Driver Developers Guide

July 1996 Edition

Part Number 320684C-01

© Copyright 1994, 1996 National Instruments Corporation.
All rights reserved.

Internet Support

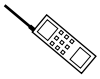
GPIB: gpib.support@natinst.com
DAQ: daq.support@natinst.com
VXI: vxi.support@natinst.com
LabVIEW: lv.support@natinst.com
LabWindows: lw.support@natinst.com
Lookout: lookout.support@natinst.com
HiQ: hiq.support@natinst.com
VISA: visa.support@natinst.com
FTP Site: ftp.natinst.com
Web Address: www.natinst.com

Bulletin Board Support

BBS United States: (512) 794-5422 or (800) 327-3077
BBS United Kingdom: 01635 551422
BBS France: 1 48 65 15 59

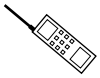
FaxBack Support

(512) 418-1111



Telephone Support (U.S.)

Tel: (512) 795-8248
Fax: (512) 794-5678



International Offices

Australia 03 9 879 9422, Austria 0662 45 79 90 0, Belgium 02 757 00 20,
Canada (Ontario) 519 622 9310, Canada (Québec) 514 694 8521, Denmark 45 76 26 00,
Finland 90 527 2321, France 1 48 14 24 24, Germany 089 741 31 30, Hong Kong 2645 3186,
Italy 02 413091, Japan 03 5472 2970, Korea 02 596 7456, Mexico 95 800 010 0793,
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, U.K. 01635 523545

National Instruments Corporate Headquarters

6504 Bridge Point Parkway Austin, TX 78730-5039 Tel: (512) 794-0100

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

NI-DAQ®, NI-488.2™, and NI-488.2M™ 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.

Table of Contents

Chapter 4
Function Panel Editor

Figures

Tables

About This Manual

The *LabWindows/CVI Instrument Driver Developer Guide* describes developing and adding instrument drivers to the LabWindows/CVI Instrument Library. This guide is for 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 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 LabWindows/CVI Instrument Library and instrument drivers, and explains how to use them. This chapter also gives a historical perspective on the instrument driver library and presents the general models for their structure.
- Chapter 2, *Developing an Instrument Driver*, explains the proper procedure for developing an instrument driver.
- Chapter 3, *The Function Tree Editor*, explains the function tree and the Function Tree Editor, and describes the Function Tree Editor menu bar, menus, and commands.
- Chapter 4, *The Function Panel Editor*, describes how to create and modify instrument driver function panels using the Function Panel Editor.
- Chapter 5, *Adding Help Information*, describes the types of help information available from an instrument driver and how you can create help information.
- Chapter 6, *Programming Guidelines for Instrument Drivers*, gives you guidelines for creating instrument drivers and using them with one another. If you write instrument drivers for general distribution to users, these guidelines ensure portability and proper operation. This chapter tells you how to create an instrument driver from a LabWindows/CVI core instrument driver.

- Chapter 7, *Required Instrument Driver Functions*, describes the implementation of the required instrument driver functions of a LabWindows/CVI instrument driver. For each required instrument driver function, the following information is presented; the C function prototype, a description of the purpose and operation of the function, a table defining each parameter, all possible completion and error codes, and any special implementation requirements.
- Chapter 8, *Instrument Driver Example*, shows you how to create a complete GPIB instrument driver. The example presented in this chapter can serve as a model for your own instrument driver development.
- Appendix A, *Tektronix 2430A Instrument Driver Code Sample*, contains instrument driver code samples for the Tektronix 2430A.
- Appendix B, *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:

bold	Bold text denotes a parameter, menu item, return value, function panel item, or dialog box button or option.
italic	Italic text denotes emphasis, a cross reference, or an introduction to a key concept.
bold italic	Bold italic text denotes a note, caution, or warning.
`monospace`	Text in this font denotes text or characters that you should literally enter from the keyboard. Sections of code, programming examples, and syntax examples also appear in this font. This font also is used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, variables, filenames, and extensions, and for statements and comments taken from program code.
`italic monospace`	Italic text in this font denotes that you must supply the appropriate words or values in the place of these items.
< >	Angle brackets enclose the name of a key. A hyphen between two or more key names enclosed in angle brackets denotes that you

	should simultaneously press the named keys–for example, <Ctrl-Alt-Delete>.
»	The **»** symbol leads you through nested menu items and dialog box options to a final action. The sequence **File»Page Setup»Options»Substitute Fonts** directs you to pull down the **File** menu, select the **Page Setup** item, select **Options**, and finally select the **Substitute Fonts** option from the last dialog box.
paths	Paths in this manual are denoted using backslashes (\) to separate drive names, directories, and files, as in `drivename\dir1name\dir2name\myfile`

Acronyms, abbreviations, metric prefixes, mnemonics, and symbols, and terms are listed in the *Glossary*.

The LabWindows/CVI Documentation Set

For a detailed discussion of the best way to use the LabWindows/CVI documentation set, see the section *Using the LabWindows/CVI Documentation Set* in Chapter 1, *Introduction to LabWindows/CVI* of *Getting Started with LabWindows/CVI*.

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 B, *Customer Communication*, at the end of this manual.

Chapter 1
Instrument Driver Overview

This chapter introduces the LabWindows/CVI Instrument Library and instrument drivers, and explains how to use them. This chapter also gives an historical perspective on the instrument driver library and presents the general models for their structure.

About the Instrument Library and Instrument Drivers

The Instrument Library is a special LabWindows/CVI library that contains a collection of instrument drivers. Instrument drivers free the user from learning the programming protocol of an instrument. The software routines of an instrument driver control an instrument, and a set of data structures represent the instrument driver within LabWindows/CVI. 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 actual physical instrument exists.

In addition to controlling the instrument, an instrument driver formats the data output from the instrument so the data can be easily presented to the user. 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 instrument driver consists of four 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
- An ASCII text file (`.doc`), which contains documentation for the instrument driver

The four 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`, 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 one or more functions that perform instrument-specific actions. The instrument drivers collectively make up the Instrument Library.

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 includes one or more 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. Most developers distribute instrument driver programs in both the `.c` and `.obj` formats.

Purpose and Benefits of Instrument Drivers

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

Instrument drivers are conceptually one layer above the traditional instrument command sets. Rather than requiring a user to include individual I/O statements throughout an application program, an instrument driver includes all the communication details of a particular instrument in high-level software functions that are directly usable by end users as part of their application programs. The instrument driver architecture defined in this manual accommodates traditional message-based instruments, both SCPI and non-SCPI, as well as direct control of VXIbus register-based modules.

All LabWindows/CVI instrument drivers are delivered in source code, whenever possible, and are fully documented. In addition, the drivers are developed using the standard LabWindows/CVI environment. Therefore, users can understand the operation of the driver and modify or enhance the operation of a particular instrument driver to achieve the optimum level of performance and flexibility for their applications.

Historical Evolution of Instrument Drivers

Instrument drivers have become increasingly popular over the last several years, and both users and vendors have taken advantage of the technology. Increased use of instrument drivers has fueled a continuous improvement process that has resulted in high-quality instrument drivers.

The 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.

This manual documents the latest instrument driver technology and specifies rules, guidelines, and requirements for the development of standard instrument drivers using LabWindows/CVI. Differences between the new standard for LabWindows/CVI instrument drivers and the old are minor. The most notable difference between the old and the new LabWindows/CVI instrument driver styles are as follows.

- VISA defined data types are used 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 the VISA (Virtual Instrumentation Software Architecture).
- The initialize function has been made generic to the type of interface (GPIB or VXI) that is used to control the instrument. All instrument addressing information is passed to the initialize function via a string parameter.

Instrument Driver Architecture

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

A VXI*plug&play* instrument driver consists of software modules that control a specific instrument. The software modules that make up an instrument driver must interact with other software in the overall system, both to communicate with the instrument and to communicate with higher-level software and/or end users who use the instrument driver. The first step in creating a standard for instrument drivers, therefore, is to define a model to explain how the instrument driver interacts with the rest of the system.

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

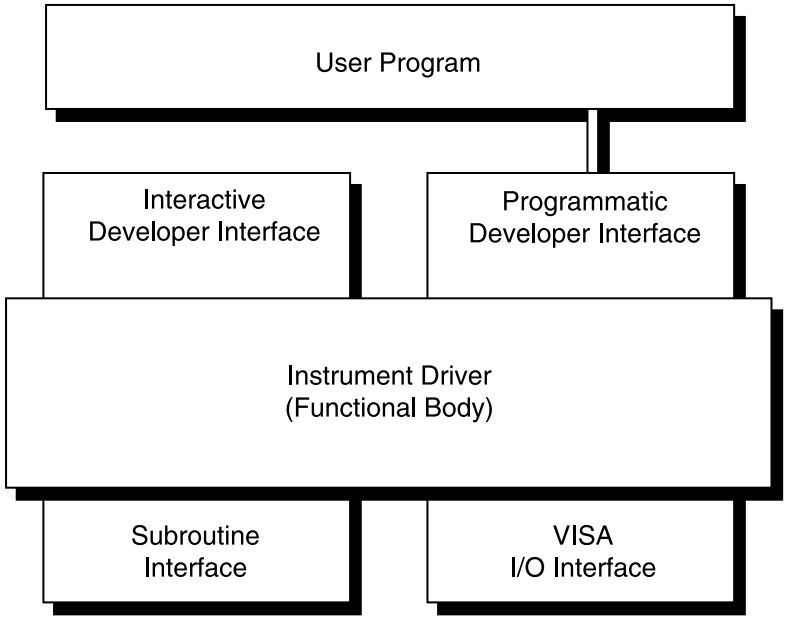


Figure 1-1. Instrument Driver External Interface Model

This general model contains the instrument driver *functional body*, which is the 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 *VISA I/O interface* is the mechanism through which the driver communicates with the instrument hardware. The *subroutine interface* is the mechanism through which the driver may call other software modules it may need to perform its task. These other software modules may include operating system calls or calls to other unique libraries such as formatting and analysis functions.

Functional Body

The functional body of a LabWindows/CVI 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 explained using the instrument driver internal design model. Chapter 6, *Programming Guidelines for Instrument Drivers*, describes the guidelines for creating the instrument driver functional body.

VISA I/O Interface

An important consideration for instrument drivers is how they perform I/O to and from instruments. 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.

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 may do nothing more than perform simple message-based and register-based I/O to and from an instrument, but others may 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 both 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. The instrument driver support functions are commonly used routines for a particular instrument driver. These functions can be either declared and defined within the LabWindows instrument driver source file or supplied in an external module. The instrument driver support functions are not exported from the instrument driver and are not intended to be accessed by the end user.

Programmatic Developer Interface

The programmatic developer interface is the mechanism for using the instrument driver as part of a test program application. LabWindows/CVI instrument drivers consist of component functions and one or more application functions to control the instrument. The programmatic developer interface to these modular software functions is a standard software function call, and the user's program has a single multi-parameter call for each instrument driver function.

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 may want to develop or use. This interface is accomplished through standard software function calls, with no special instrument-driver-specific requirements.

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

Interactive Developer Interface

When a LabWindows/CVI instrument driver is used as an integral part of a higher-level application software development environment, the programmatic developer interface to the instrument driver can be enhanced with graphical function panels. Function panels, referred to as the *interactive developer interface*, are designed to assist the programmer by making it easier to understand how to use the instrument driver. The function panel interface allows the programmer to operate the particular instrument driver function interactively to understand the function and automatically generate the instrument control statements that are used in an application program.

Instrument Driver Internal Design Model

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

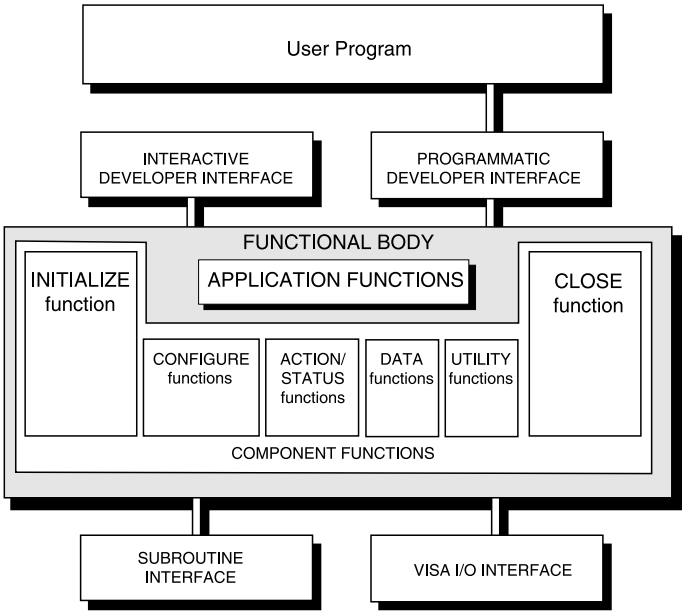


Figure 1-2. Instrument Driver Internal Design Mode

The functional body of a LabWindows/CVI instrument driver consists of two main categories. The first category is a collection of component functions, which are software modules that each control a specific area of the instrument's functionality. The second category is a collection of application functions, that show how to use the component functions together to perform complete test and measurement operations.

The modularity of LabWindows/CVI instrument drivers builds on proven technology. With a modular approach, a user has the granularity needed to control instruments properly in their software application. The user can, for example, initialize all instruments once at the start, configure multiple instruments, and then trigger several instruments simultaneously. As another example, a user can initialize and configure an instrument once, and then trigger and read from the instrument several times.

Component Functions

LabWindows/CVI instrument drivers have component functions, which are divided into six categories: initialize, configuration, action/status, data, utility, and close. Each of these categories, with the exception of the initialize and close functions, will consist 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
- Error Query
- Error Message
- Revision Query

The remainder of instrument driver functions are known as developer-specified functions, and the actual operations performed by those routines are left up to the instrument driver developer. All instruments will have configuration functions, for example, but different instruments may have different numbers of configuration functions depending on the differences in how the instruments can be configured. General guidelines in Section 6, 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 specifications recommend that an instrument driver provide full function control of the instrument. However, it 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.

Initialize Function

The initialize function initializes the software connection to the instrument. The initialize function can optionally perform an instrument identification query and reset operations. In addition, it may perform any necessary actions to place the instrument in its default power-on state or other specific state.

Configuration Functions

The configuration functions are a collection of software routines that configure the instrument to perform the desired operation. There may be numerous configuration functions, depending on the particular instrument.

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 a measurement instrument, functions for downloading waveforms or digital patterns to a source instrument, and so on. The specific routines in this category and the actual operations performed by those routines are left up to the instrument driver developer.

Utility Functions

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

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

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 that are also provided in source code. These examples are instrument driver functions that can be called via their own program interface when a user wants a single, test and measurement oriented, and

high-level function interface to the driver. In most cases, these examples are single functions that configure, start, and read the instrument, all in a single operation.

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

Note: ***LabWindows/CVI instrument driver application functions do not call the*** `PREFIX_init` ***or*** `PREFIX_close` ***functions.***

Chapter 2
Developing an Instrument Driver

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

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:

- Before creating your instrument driver, define the structure of the driver. A useful instrument driver is more than a group of functions; it is a tool to help users develop application programs. Therefore, design an instrument driver with the user in mind.
- Always base your instrument on one of the core instrument drivers or an existing instrument driver developed from one of the core instrument drivers.
- 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. 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.
3. Create a function tree for the instrument driver, adding help information to the top level of the tree.
4. For each function in the driver:
    a. Define the parameters to the function, including variable types and limits, and error codes.

b. Create the function panel for the function, including help information for the panel and for each control.

c. Write the code to perform the function.

d. Test the source code.

5. Create the include file for the final instrument source code, including function declarations and constant definitions.

6. Operate the completed driver using function panels without loading the source code.

7. Document the driver.

Naming the Driver

When you create instrument drivers, you add routines to the LabWindows/CVI Instrument Library. 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`. 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.***

Defining the Instrument Functions

An instrument driver can feature one or more functions you can use to program the instrument. For a simple instrument, you can use two or three functions through which you can program the instrument. For a more complex instrument, you can use function classes, each of which contains functions specific to that class. In addition, you can break down complex instruments conceptually into independent instrument drivers, where each driver represents one major application of the complex instrument.

Structuring Functions In An Instrument Driver

The three implementations of a single instrument driver in this section show you some options for structuring functions. In this example, the designer 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 may at first seem logical to use the functions `SET_DMM_RANGE` and `SET_DMM_FUNCTION` for setting the range and function of a multimeter. However, a more useful function may be `DMM_CONFIG`, for setting up multiple parameters.

Defining the Hierarchy of Functions

If the instrument has numerous functions that logically fall into classes, you can group them into function classes or treat the groups of functions as independent instruments. 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 LabWindows/CVI. From the application program, functions within an instrument driver are all called the same way, regardless of which function class they are in. Functions that are divided between separate instrument drivers, however, are treated as functions of two distinct instruments.

Defining the Function Parameters

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

Function parameters give input information to the function and variables where the function can store its results, or output. Output parameters can contain values that were read from the instrument and formatted for the user.

Data Types

You must define a data type for each parameter in an instrument driver. All data types used by the instrument driver must be defined for 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 prototypes in the instrument driver header file.

With this information, LabWindows/CVI gives appropriate variable declaration and run-time checking capabilities when users operate a function panel. When you declare a variable from a function panel, LabWindows/CVI presents options based on the data type defined in the `.fp` file. 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* defined by LabWindows/CVI.

Intrinsic C Data Types

The *intrinsic C data types* predefined by LabWindows/CVI 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 open and close 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 will then perform the appropriate variable declaration and runtime checking capabilities when the user operates the function panel.

When you define a `.fp` control with an intrinsic C data type, variables you declare in that `.fp` control using the **Declare Variable** command appear with that data type in the dialog box. You must define the parameter as that data type in the instrument driver function prototype.

Meta Data Types

The *meta data types* combine intrinsic C data types and user-defined data types. The meta data types are `Numeric Array`, `Any Array`, `Any Type`, and `Var Args`. These data types define sets of allowable data types for a parameter. 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 allowable data types.

Numeric Array

Numeric Array specifies a parameter that may 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` and 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 may 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` and is defined as a `void *` in the function prototype shown below.

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

Any Type

Any Type specifies a parameter that may 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 control attribute from the selected panel and control. Although attribute values may be of different data types, the parameter is passed by reference and 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. It is passed by value rather than by reference and thus 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 may 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 is an example 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 function prototypes shown below.

```
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 header file of the driver. Declare user-defined data types with the **Data Types** command box in the Function Panel Editor.

For example, you can define a data type `waveform_var` 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.

Step one for the `waveform_var` data type presented previously is to include the following code in the header file of the instrument driver.

```
typedef struct {
   double waveform_arr [500];
   float t_zero;
   float t_delta;
} waveform_var;
```

Step two is to 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 select **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 was defined as `waveform_var`.

See Chapter 4, *The 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 will be used as arrays. If you want to define a user-defined array data type, brackets `[]` must appear at the end of the type 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 in the instrument driver header file the `typedef` declaration for `waveform.var` that was presented in the previous example.

Examples of incorrect ways to define array user-defined 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

A special set of data types are defined by the VISA I/O library. 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 used for instrument drivers are as follows.

Table 2-1. VISA Data Types.

VISA Type Name	Definition
`ViInt16`	Signed 16-bit integer
`ViInt32`	Signed 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 select **Add VISA Types** from the **Edit Data Type List** dialog.

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

See Chapter 4, *The Function Panel Editor*, for a discussion of the **Edit Data Type List** dialog box.

Input and Output Parameters

Because instrument drivers generally reflect a physical instrument, the input and output function parameters 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 brackets `[]` at the end of the data type name.

   b. For output parameters, select the data type of the value that will be passed, not a 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, in a function `examp_func` that had an output `examp_out` as an integer parameter, you prototype the function in the instrument driver header file as

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

   When you create a function panel for this function, you need to 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 will create a variable of the type specified and automatically put 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 may 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

If your instrument driver is for a GPIB, VXI, or RS-232 instrument, you are required to define several functions. These functions are as follows.

- Initialize
- Close
- Reset
- Self-Test
- Error Query
- Error Message
- Revision Query

Chapter 7, *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 3, *The 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 5, *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 3, *The Function Panel 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 5, *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 6, *Programming Guidelines for Instrument Drivers*.

Operating the Driver

After you have created 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, *The 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 6, *Programming Guidelines for Instrument Drivers*, contains guidelines and suggestions for documenting your instrument driver.

Chapter 3
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 3-1.

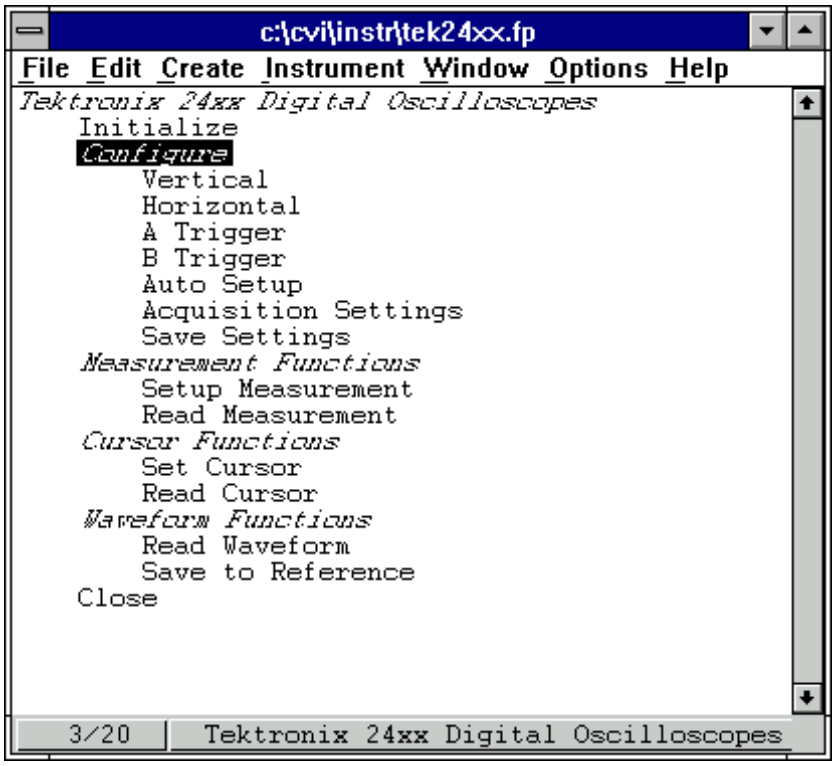


Figure 3-1. A Function Tree

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

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` 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.
- **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` 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 4, *The Project Window,* tells more about the **File** menu.

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 highlighted function or class from the tree and copies it to the Clipboard.
- **Copy** copies the highlighted function or class from the tree to the Clipboard.
- **Paste Above** inserts the contents of the Clipboard into the tree above the highlighted line.
- **Paste Below** inserts the contents of the Clipboard into the tree below the highlighted line.
- 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 5, *Adding Help Information,* to learn how to add help information.
- **Edit Function Panel Window** lets you edit the highlighted function in the function tree editor and display it in the Function Tree Editor. Chapter 4, *Function Panel Editor,* gives you information on using the Function Panel Editor.
- **FP Auto-Load List** allows you to specify other instrument drivers on which the instrument driver you are currently developing is dependent. These instrument drivers are loaded when the current instrument driver is loaded

  – via the **Load** command in the **Instrument** menu

  – in the process of loading a project file in which the current `.fp` file is listed.

  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.

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

  1. It first looks in the directory of the referencing `.fp` file.

2. It then looks for them in the "instrument directories list" which is edited using the **Instrument Directories** command in the **Options** menu of the **Project** window.

3. 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 is not listed 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 within the instrument driver code. 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 line `Create Class or Function Panel Window` 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 which users choose from the **Instrument** menu.

Adding a Class to an Empty Tree or Class

Add a class to an empty tree as follows.

1. Highlight the line containing `Create Class` or `Function Panel Window`.
2. Select **Class** from 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 highlighted 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. Highlight an existing function or class at the level you want to place the new class.
2. Select **Class** from 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 32000 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. Enter in the Name text box the name that you want to appear in the Function Panel Selection dialog box when the instrument is chosen from the **Instrument** menu.

2. Enter in the Function Name text box 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. The prefix was specified when the function tree was created from 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. Highlight the line containing Create Class or Function Panel Window.

2. Select **Function Panel Window** from 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 highlighted line.

Inserting a Function into an Existing Tree

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

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

2. Select **Function Panel Window** from 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 3-2 appears.

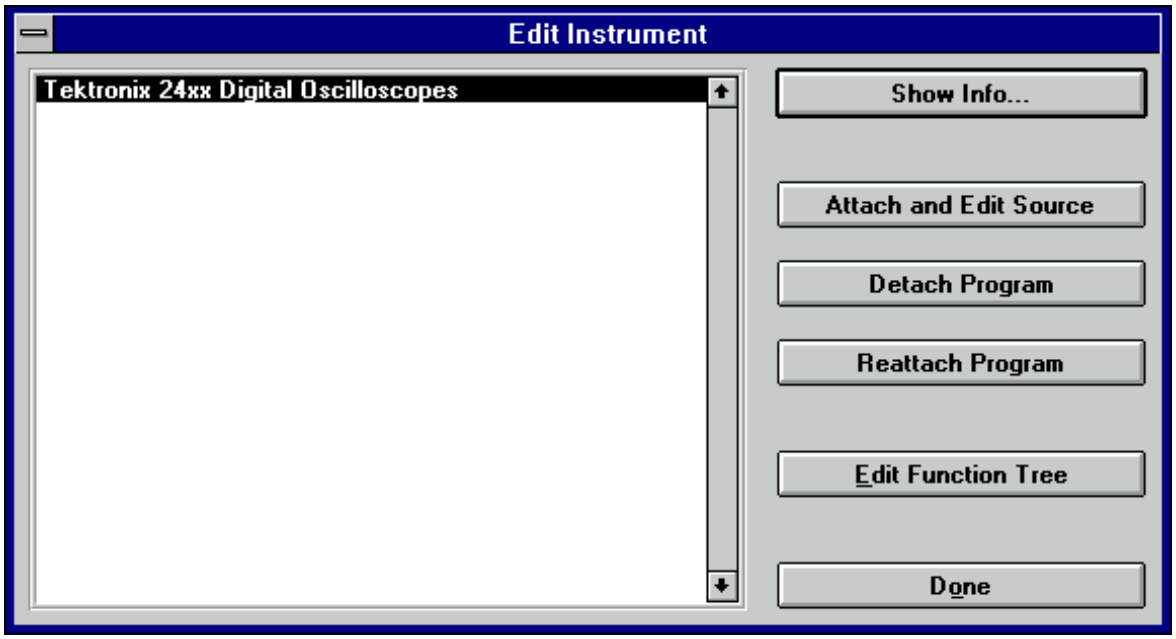


Figure 3-2. The Edit Instrument Dialog Box

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.

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, *The 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 less. 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 will have to change the help text to a Windows-compatible character set.

- **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 2 source files (`.rtf` and `.whh`) that can be used 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.

- The **Create DLL Project** (Windows 95 and Windows 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, *LabWindows/CVI Programmer Reference Manual* in this document.

- **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 *VXIplug&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 *VXIplug&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 Incude** 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 *VXIplug&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. Select **Instrument** from the **Create** menu.
2. Enter the name `Function Tree Examples` as the Name and `tree` as the Prefix. Click on **OK**.
3. Select **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. Select **Class** from the **Create** menu.
6. Enter the name `Class 1` as the Name. Click on **OK**.
7. Position the highlight on the line beneath the name `Class 1`.
8. Select **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. Select **Save .FP File As** from the **File** menu and save the file as `mltcls`.

The new function tree is shown in Figure 3-3.

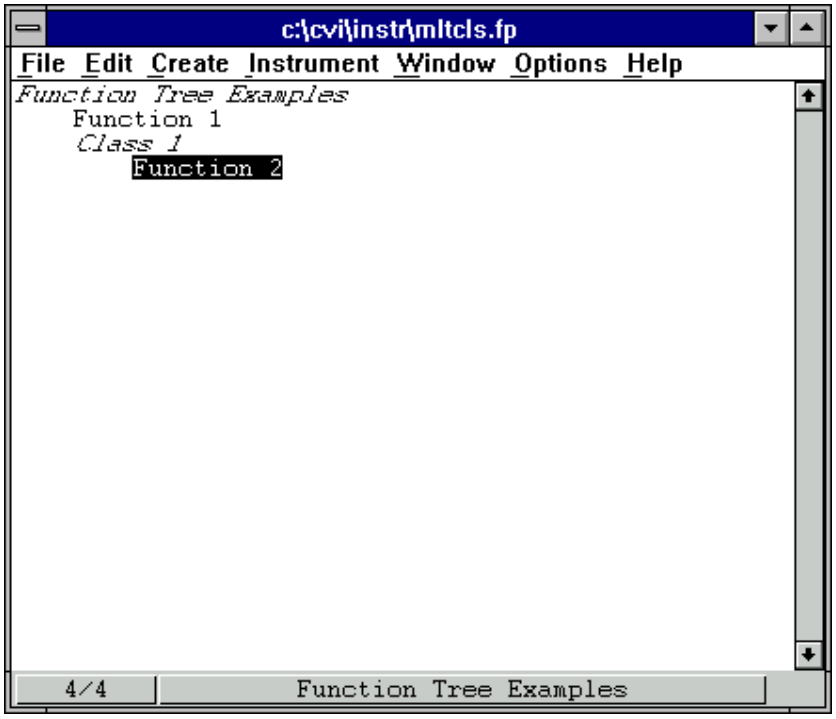


Figure 3-3. A Sample Function Tree

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 highlight on the name `Function 1`.
2. Select **Cut** from the **Edit** menu. The function disappears from the tree and is stored on the Clipboard.
3. Position the highlight on the name `Function 2`.
4. Select **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 highlight 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. Select **New, Function Tree (*.fp)** from the **File** menu. A new blank function tree window appears on the screen.
2. Select **Instrument** from the **Create** menu.
3. Name the instrument `New Instrument` and type `new` in the prefix box. Click on **OK**.
4. Select **Function Tree** from the **Window** menu and select the file called `mltcls`.
5. Position the highlight on the item `Class 1`.
6. Select **Copy** from the **Edit** menu.
7. Return to the `New Instrument` file through the **Window** menu.
8. Position the highlight on the line beneath the name of the instrument.
9. Select **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. Highlight `New Instrument`.
2. Select **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 will 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 will be reflected in the Generated Code Window in each function panel.

Chapter 4
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 a LabWindows/CVI library function panel, 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 4-1 appears.

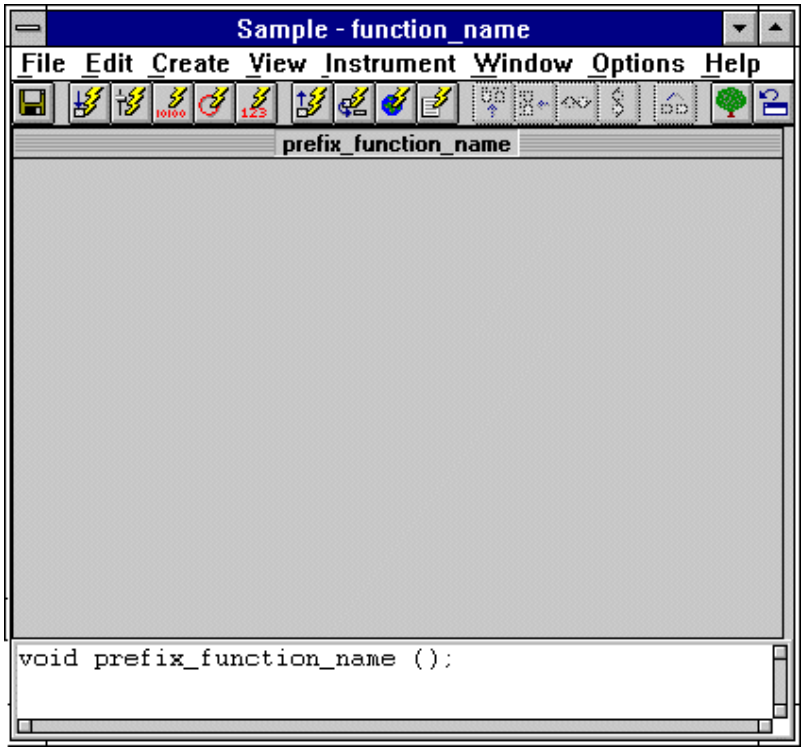


Figure 4-1. The Function Panel Editor

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 Name appear in the title bar of the function panel window.
- The Function Code Name appears in the title bar of the function panel.
- The Function Code 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` 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.
- **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` 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*, of the *LabWindows/CVI User Manual*, gives more information about the **File** menu.

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 highlighted controls and copies them to the Clipboard.
- **Copy Controls** copies the highlighted controls to the Clipboard.
- **Paste** inserts the contents of the Clipboard into the highlighted function panel.
- **Cut Panel** deletes the highlighted panel from the function panel window and copies it to the Clipboard.
- **Copy Panel** copies the highlighted 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.
- **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 used 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 used 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 used to create the panel. The *Create* section later in this chapter discusses the proper use of these dialog boxes.

Alignment

Alignment lets you align a set of highlighted 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 a set of highlighted 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.

Control Help

You can add or modify context-sensitive help information for a particular control with **Control Help**. Chapter 5, *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 3, *The Function Tree Editor*, for more information on how to set the help style of the instrument driver. See Chapter 5, *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.

Function panels graphically represent a single function in the function panel window. Function panels may contain any of the nine different control types. A function panel may only have one return value control. The function panel window may 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. 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, we recommend 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 4-2.

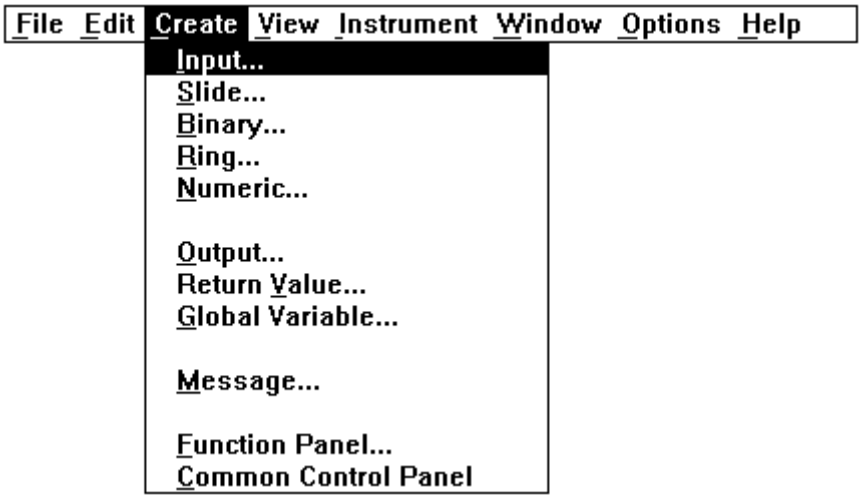


Figure 4-2. Control Types

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 4-3 appears.

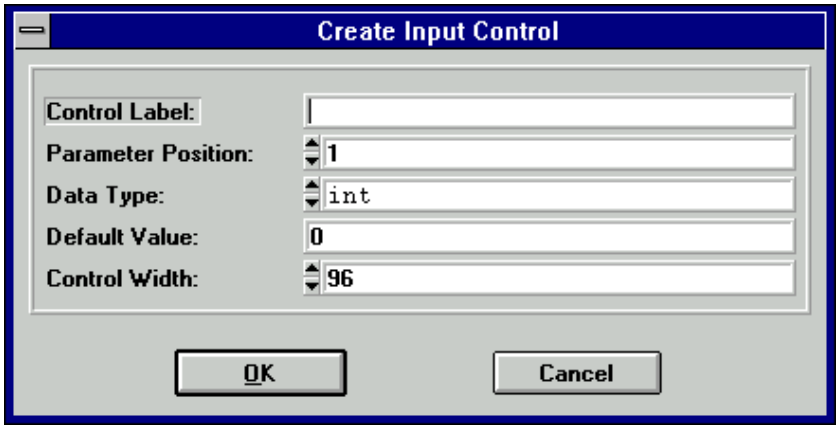


Figure 4-3. The Create Input Control Dialog Box

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 2, *Developing an Instrument Driver*.

- **Default Value** specifies the default for the input control, which should 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 2048.

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 4-4 appears.

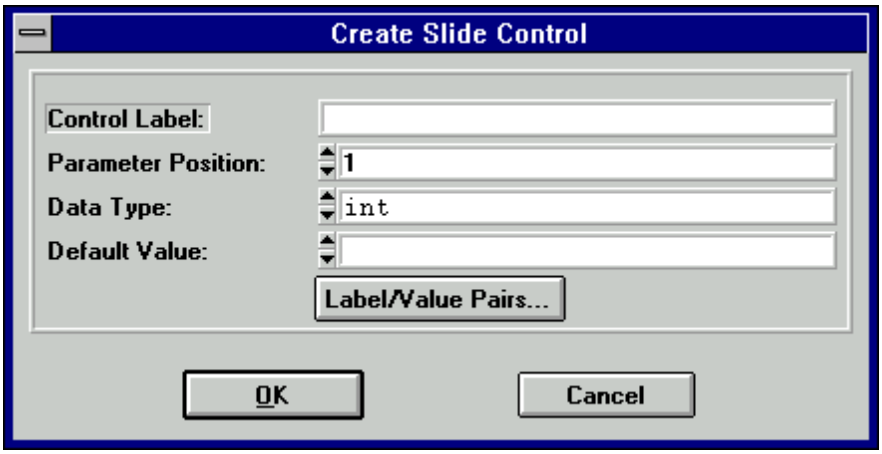


Figure 4-4. The Create Slide Control Dialog Box

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 2, *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 4-5 appears.

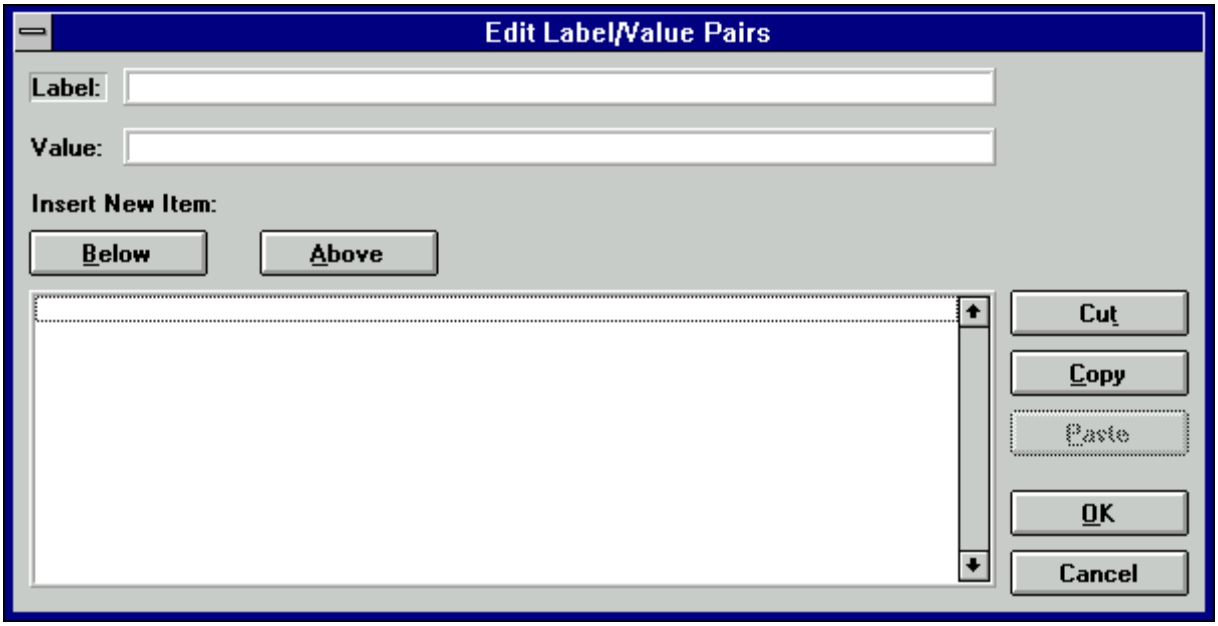


Figure 4-5. The Edit Label/Value Pairs Dialog Box

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 may 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 highlighted 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 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 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 4-6 appears.

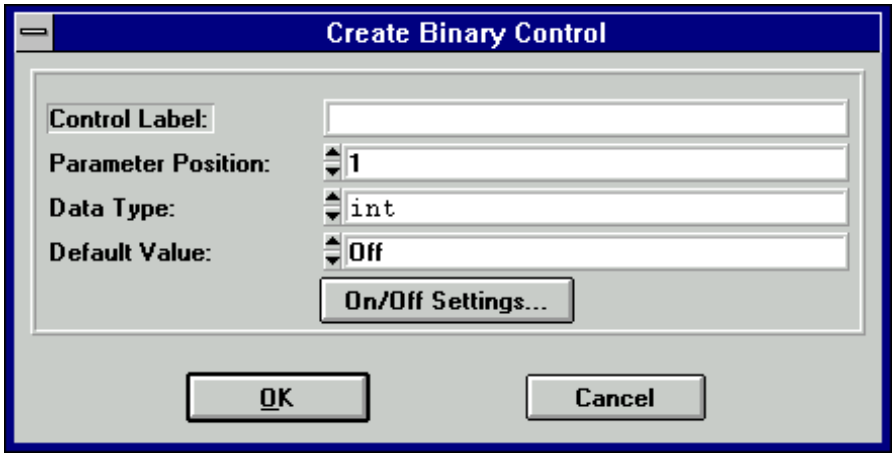


Figure 4-6. The Create Binary Control Dialog Box

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 2, *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 4-7 appears.

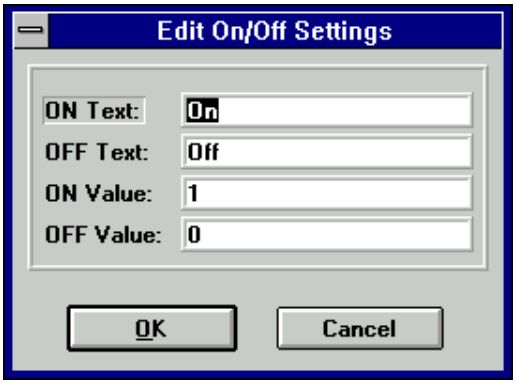


Figure 4-7. The Edit On/Off Settings Dialog Box

- **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 expression associated with the On label.
- **OFF Value** specifies the value, constant name, or expression associated 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 4-8 appears.

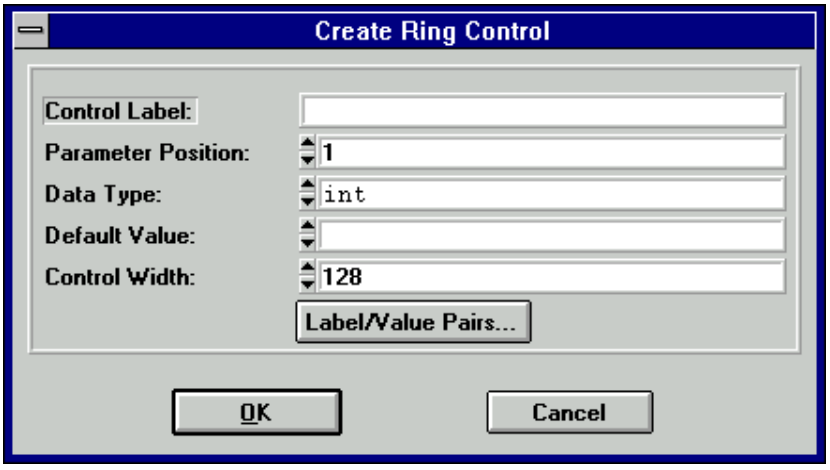


Figure 4-8. The Create Ring Control Dialog Box

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 2, *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 2048.

When you press the **Label/Value Pairs** button the Edit Label/Value Pairs dialog box shown in Figure 4-9 appears.

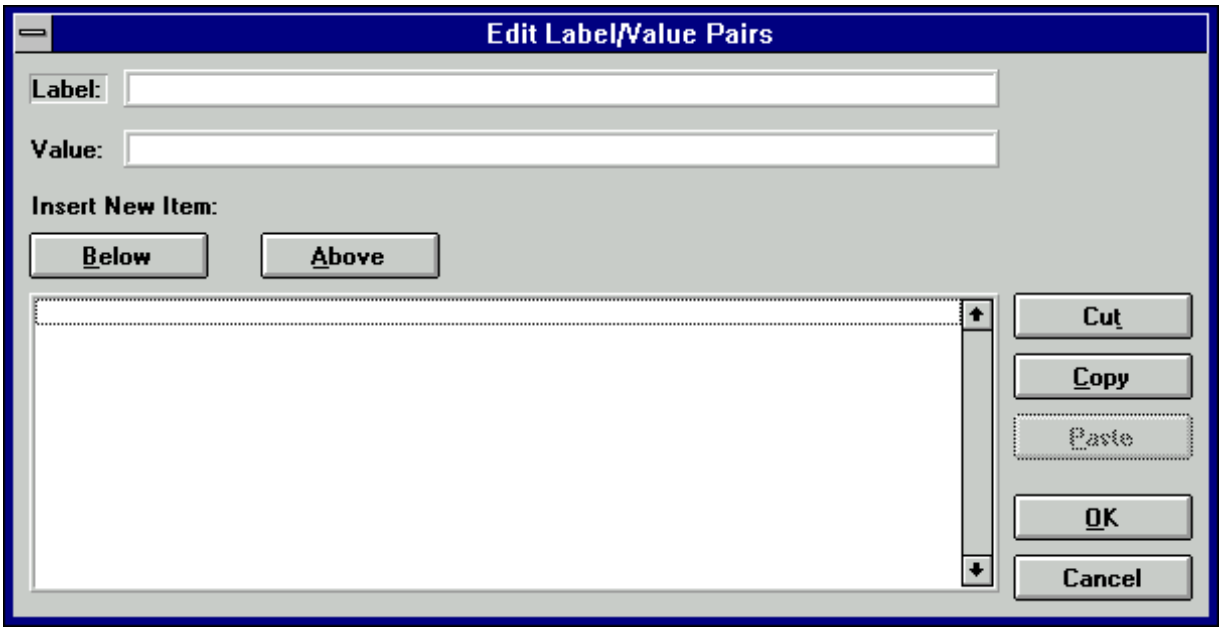


Figure 4-9. The Ring Control Edit Label/Value Pairs Dialog Box

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 32000 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 may 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 highlighted 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 4-10 appears.

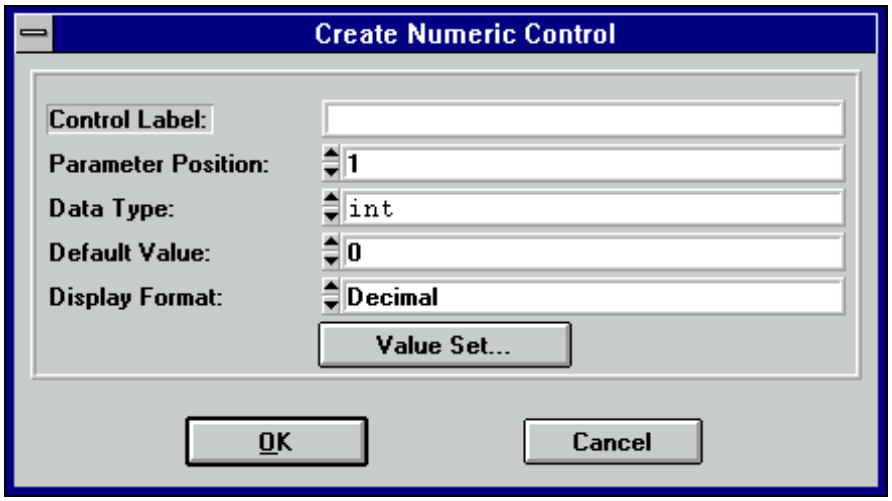


Figure 4-10. The Create Numeric Control Dialog Box

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 4-11 appears.

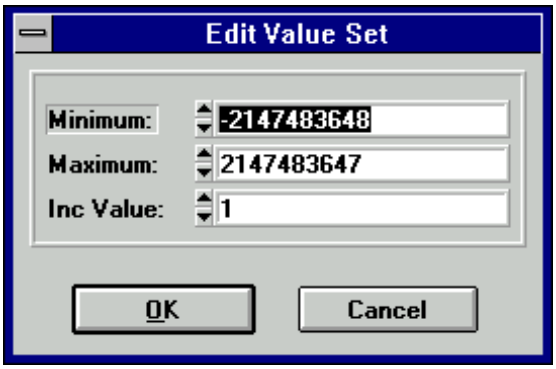


Figure 4-11. The Edit Value Set Dialog Box

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 4-12 appears.

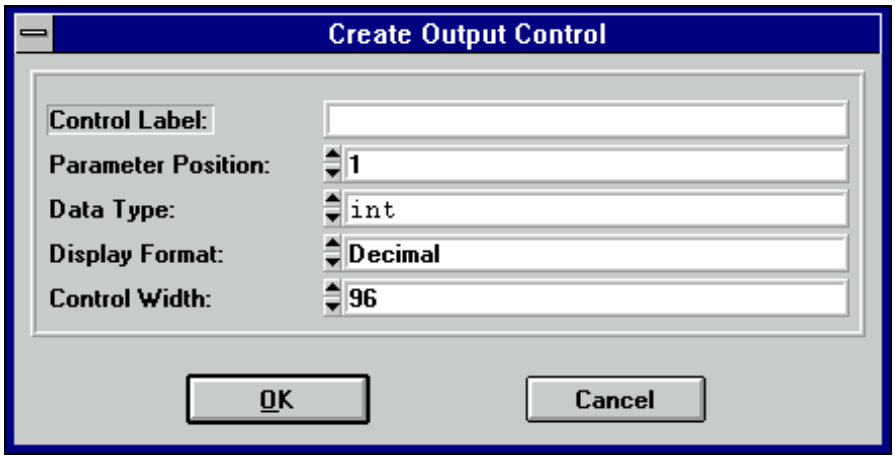


Figure 4-12. The Create Output Control Dialog Box

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 2, *Developing an Instrument Driver*.
- **Display Format** lets you select the format in which values in the output control are displayed. 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 2084.

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 4-13 appears.

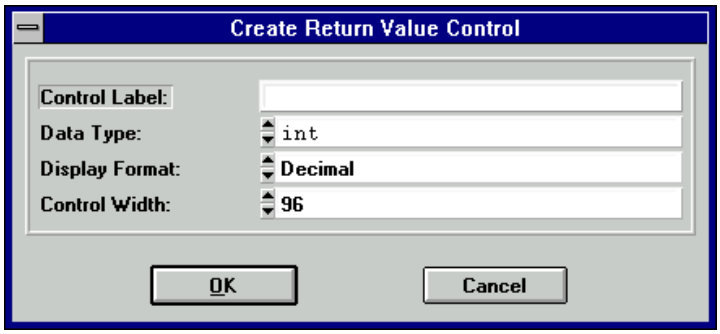


Figure 4-13. The Create Return Value Control Dialog Box

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 values in the return control are displayed. 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 2048.

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 4-14 appears.

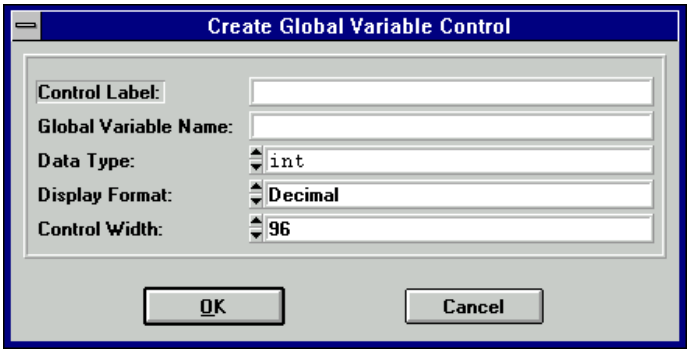


Figure 4-14. The Create Global Variable Control Dialog Box

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 2, *Developing an Instrument Driver*.
- **Display Format** lets you select the format in which values in the global variable control are displayed. 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 2048.

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 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 3, *The Function Tree Editor*, gives more information about the **Instrument** menu.

Window

The **Window** menu lets you select which window to make active. The **Window** menu operates like to the **Window** menu of the Project window. Chapter 3, *The 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: ***You must define the data types specified with the Data Types command in the .h file for the instrument driver.***

When you select **Data Types** from the **Options** menu, the dialog box in Figure 4-15 appears.

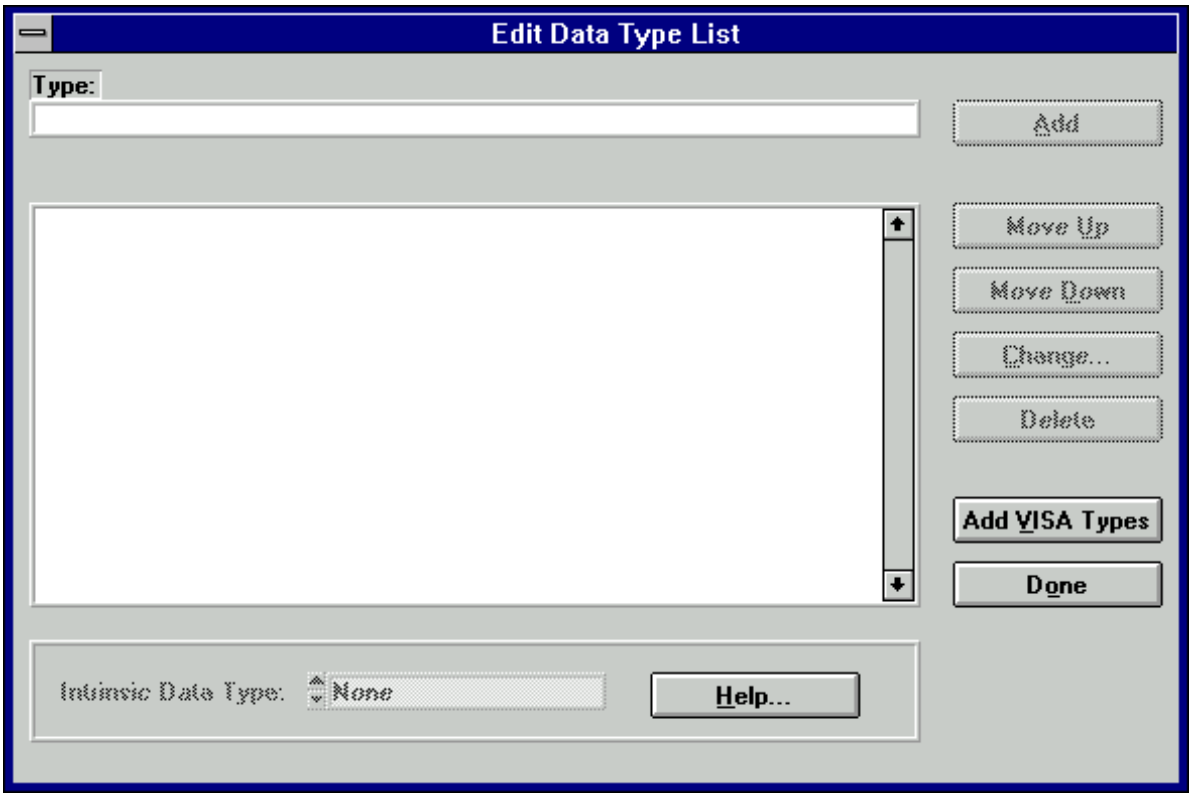


Figure 4-15. The Edit Data Type List Dialog Box

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

- **Type** specifies the name of a user-defined data type.
- **Add** places the name in the Type control in the Data Type list.
- **Move Up** moves the highlight up one entry in the Data Type list.
- **Move Down** moves the highlight down one entry in the Data Type list.
- **Change...** displays a dialog box that prompts you to change the highlighted 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.
- **Intrinsic Data Type** allows you to associate each user defined data type with one of the data types that can be used in a numeric control. If you select an item other than `None`, you will be able to use the user-defined data type as the data type for a numeric control.

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. (They 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. Highlight the desired control.
2. Select **Cut Controls** from the **Edit** menu.
3. Move to the new function panel.
4. Select **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. Select the **Function Tree (*.fp)** option of the **New** command from the **File** menu.
2. Select **Instrument** from the **Create** menu.
3. Enter `Function Panel Examples` as the Name and `panel` as the Prefix. Click on **OK**.
4. Select **Function Panel Window** from the **Create** menu.
5. Enter `Configure` as the Name and `config` as the Function Name. Click on **OK**.
6. Highlight the item `Configure` in the function tree and select **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 4-16.

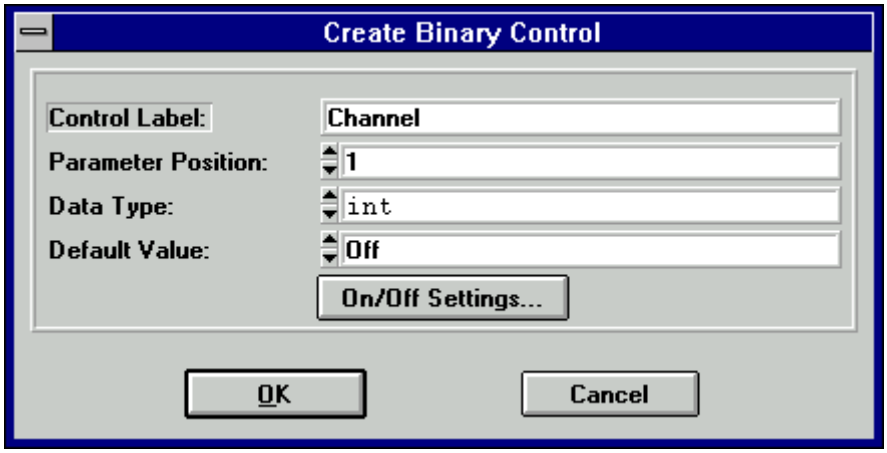


Figure 4-16. The Channel Create Binary Control Dialog Box

9. Press the **On/Off Setting** button and complete the Edit On/Off Settings dialog box as shown in Figure 4-17. Position the control on the panel.

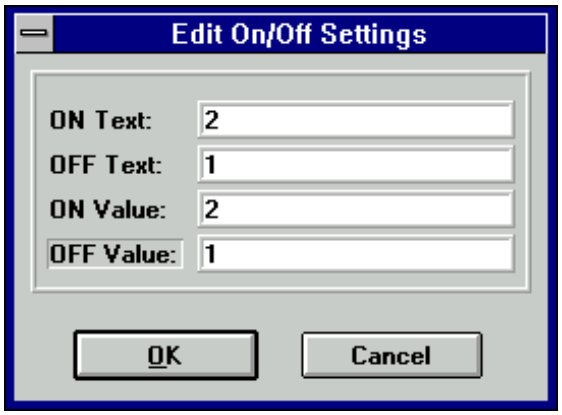


Figure 4-17. The Channel Edit On/Off Settings Dialog Box

10. Select **Input** from the **Create** menu.

11. Complete the Create Input Control dialog box as shown in Figure 4-18, and position the control on the panel.

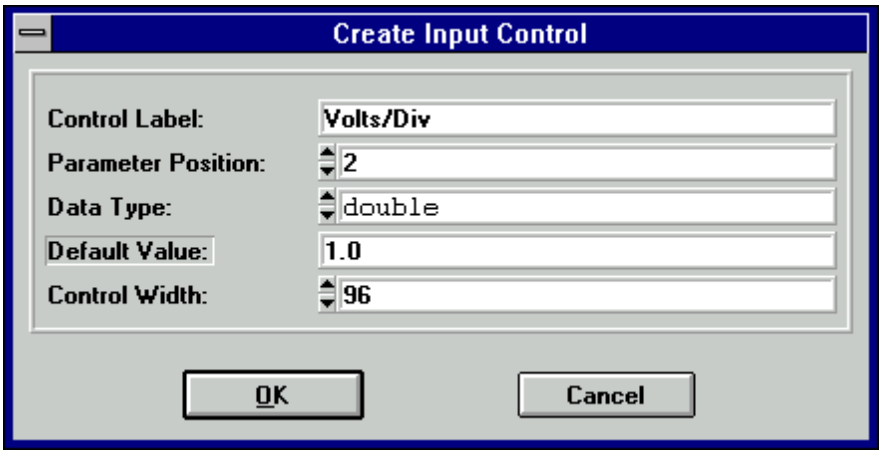


Figure 4-18. The Volts/Div Create Input Control Dialog Box

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

13. Complete the Create Slide Control dialog box as shown in Figure 4-19.

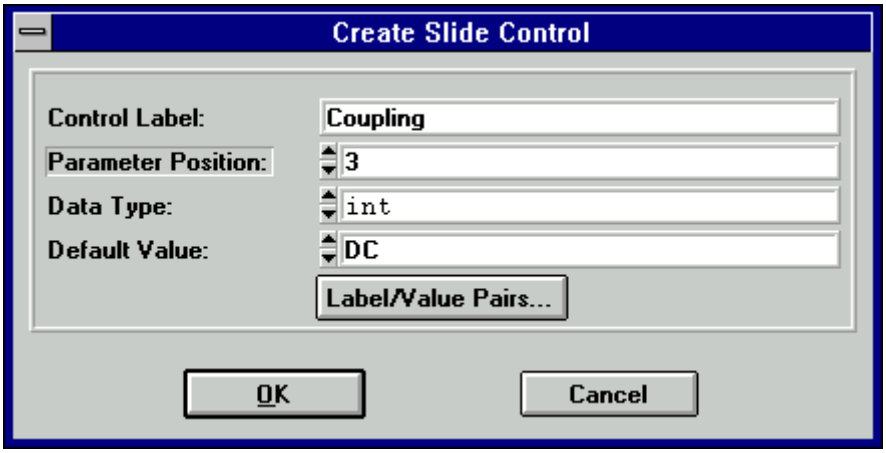


Figure 4-19. The Coupling Create Slide Control Dialog Box

14. Press **Label/Value Pairs**, and complete the Edit Label/Value Pairs dialog box as shown in Figure 4-20, and position the control on the panel.

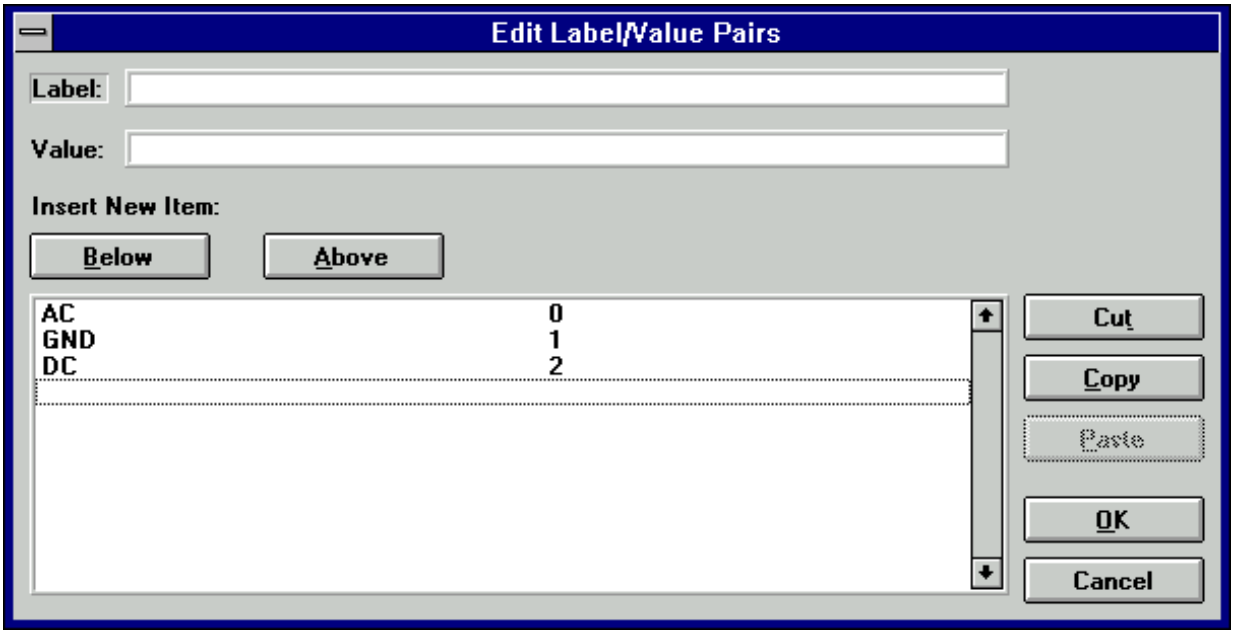


Figure 4-20. The Coupling Edit Label/Value Pairs Dialog Box

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

16. Complete the Create Binary Control dialog box as shown in Figure 4-21.

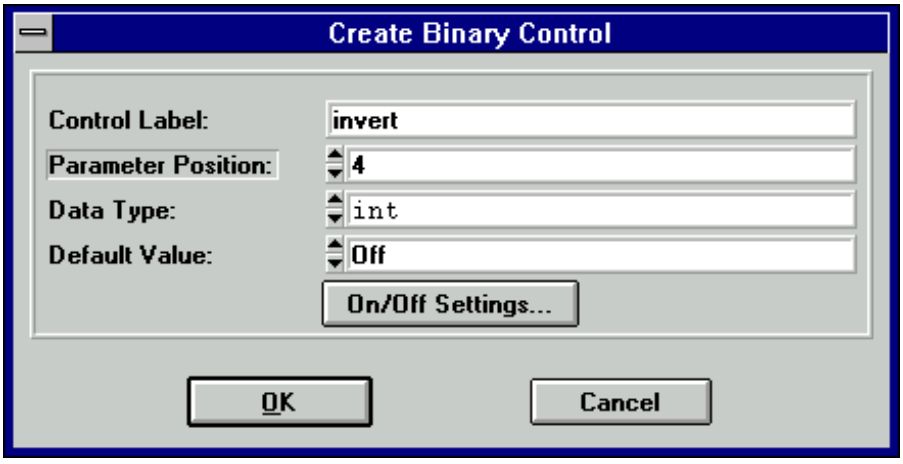


Figure 4-21. The Invert Create Binary Control Dialog Box

17. Press the **On/Off Settings** button and complete the Edit On/Off Settings dialog box as shown in Figure 4-22. Position the control on the panel.

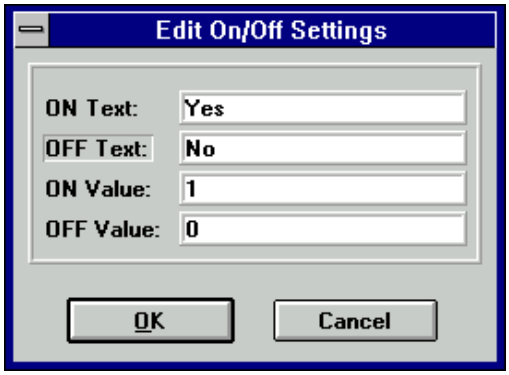


Figure 4-22. The Invert Edit On/Off Settings Dialog Box

You now see the function panel shown in Figure 4-23.

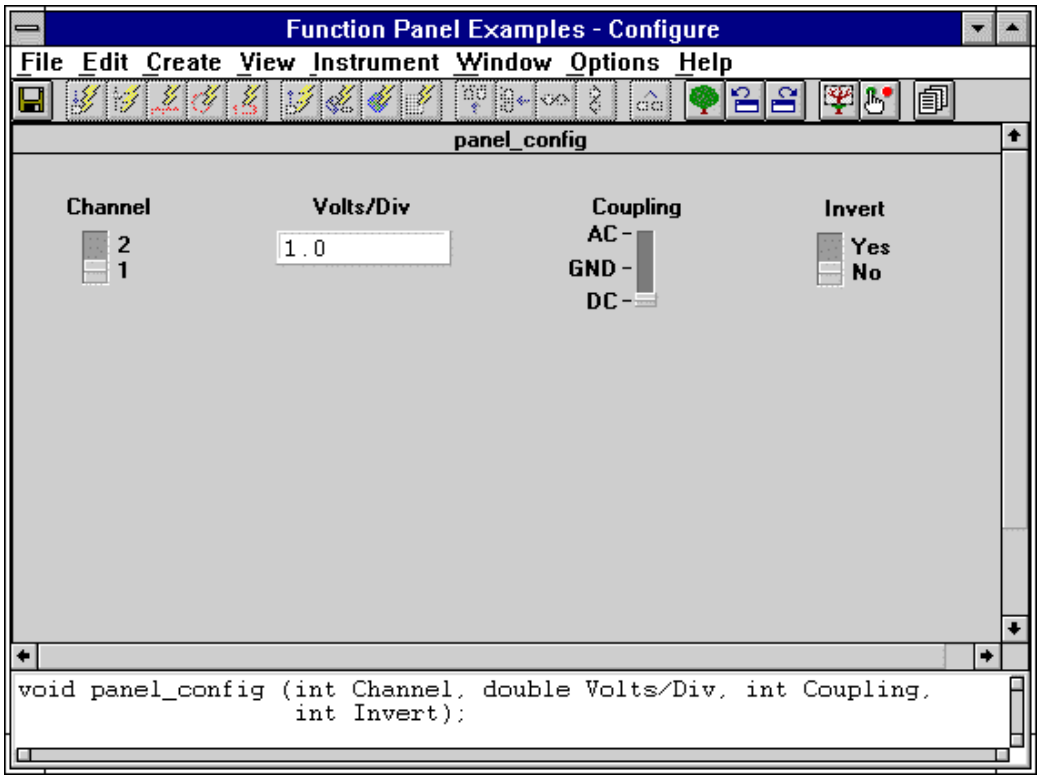


Figure 4-23. A Function Panel Window

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 highlight on the Volts/Div control.

2. Select **Change Control Type** from the **Edit** menu. The dialog box shown in Figure 4-24 appears.

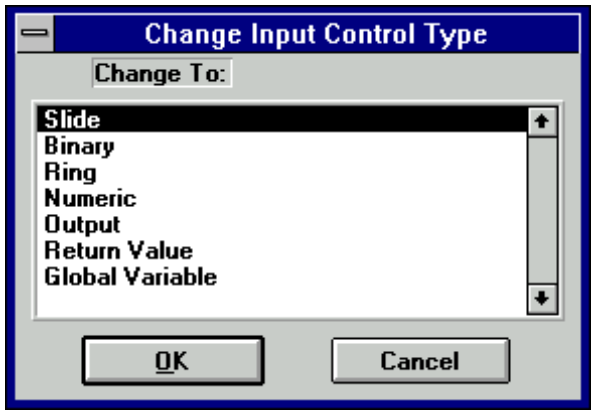


Figure 4-24. The Change Input Control Type Dialog Box

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 4-25.

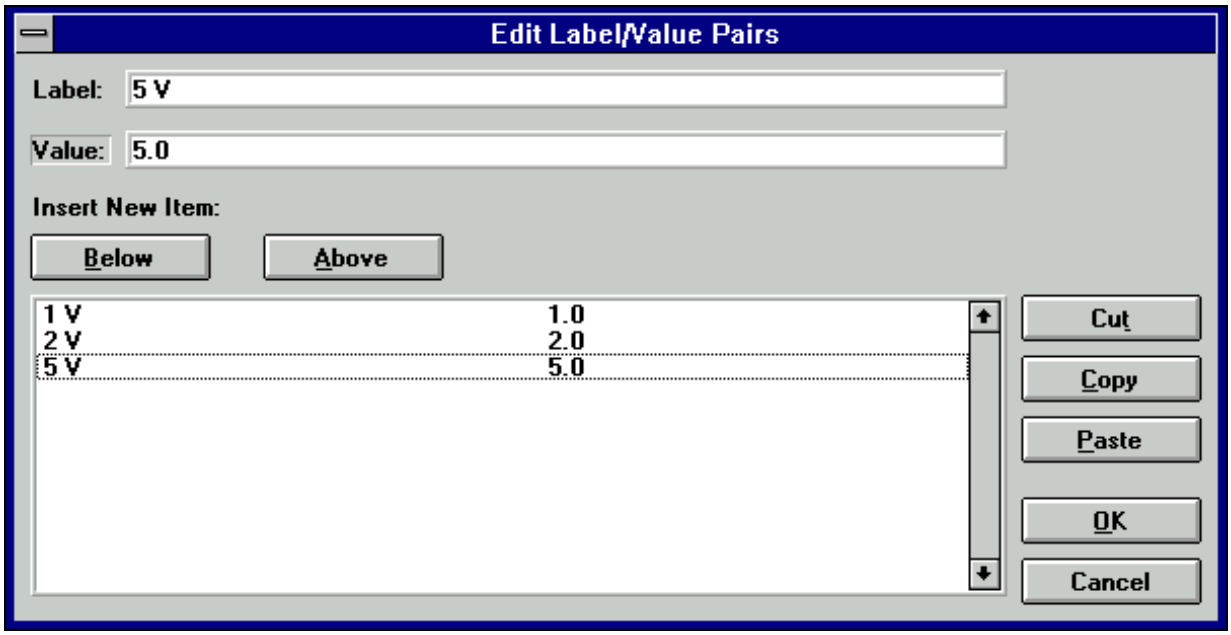


Figure 4-25. The Volts/Div Edit Label/Value Pairs Dialog Box

After you complete the slide control dialog box, select **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 highlight on the Volts/Div control.
2. Select **Change Control Type** from the **Edit** menu.
3. Select **Ring**. The Edit Ring dialog box appears.
4. Select **Label/Value Pairs**, leaving all items unchanged**.** The Edit Label/Value Pairs dialog box appears. Notice that the slide control label value pairs remain.
5. Select **OK**.

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

Example—Cutting and Pasting Controls

You will frequently 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 highlight on the Volts/Div control.
2. Select **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. Select **Save .FP File** and then select **Close** from the **File** menu of the Help Editor dialog box.
5. With the highlight still on the Volts/Div control, select **Copy Controls** from the **Edit** menu.
6. Select **Paste** from the **Edit** menu.
7. With the highlight on the new control, select **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. Select **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 highlight on the function name `Configure`.
4. Select **Edit Function Panel Window** from the **Edit** menu to return to the Configure panel.
5. Position the highlight on the control Volts/Div 2.
6. Select **Cut Controls** from the **Edit** menu.
7. Press <Ctrl-Page Down> to move to the New Panel function panel.
8. Select **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.

Chapter 5
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.

We recommend that you use the New help style for all help information for instrument drivers created in LabWindows/CVI. Chapter 3, *The 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 5-1. 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. Help information should always be added in the new style.

Add help information as follows.

1. From either the Function Tree Editor or the Function Panel Editor, place the highlight on the item that you want to enter help information.

2. Select **Edit Help, Function Help**, or **Control Help** from the **Edit** menu in the menu bar. The dialog box shown in Figure 5-1 appears.

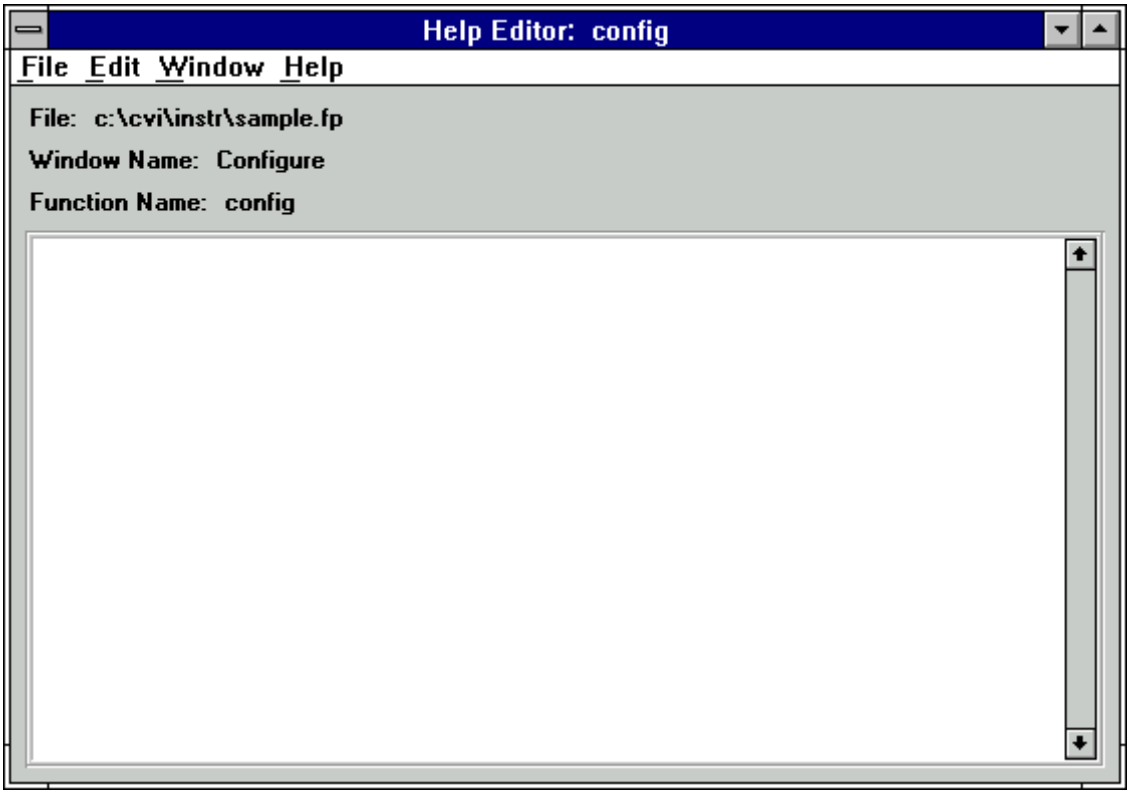


Figure 5-1. The Help Editor Dialog Box

The Help Editor dialog box contains a scrollable text box. Enter help text in the dialog box as you do in the Program or Interactive window. You can scroll the displayed text using the arrow keys or the scroll bars.

You see the following items on the Help Editor dialog box menu bar.

- **File** lets you load, save, and manipulate files.
- **Edit** lets you edit the help text entered in the dialog box.
- **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` 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.

Edit

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

- **Cut** deletes the highlighted text in the dialog box and copies the text to the Clipboard.
- **Copy** copies the highlighted text in the dialog box to the Clipboard without deleting the highlighted text.
- **Paste** inserts the contents of the Clipboard into the dialog box at the location of the cursor.
- **Delete** discards the highlighted text in the dialog box without copying it to the Clipboard.
- **Revert** returns the most recently saved version of help text to the dialog box.

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, *The Project Window*, in the *LabWindows/CVI User Manual*, gives more information about the **Window** menu.

Instrument Help

You can select the **Instrument Help** button to see help information about an instrument when you are viewing help information for a function class or function panel window.

You can add instrument help information in the Function Tree Editor. Follow these steps to enter the help information for the instrument:

1. From the Function Tree Editor, highlight the instrument name at the top of the function tree.
2. Select **Edit Help** from the **Edit** menu. The Help Editor dialog box appears. Alternatively, you can click on the instrument name with the right mouse button to display the Help Editor dialog box.
3. Enter the desired help text into the Help dialog box.

Function Class Help

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

You enter function class help information from the Function Tree Editor. Follow these steps to add help information.

1. Highlight the desired class.

2. Select **Edit Help** from the **Edit** menu in the Function Tree Editor menu bar. The Help Editor dialog box appears. Alternatively, you can click on the desired control with the right mouse button to display the Help Editor dialog box.

3. Enter the desired help text into the Help dialog box.

Function Help (New Style Help Only)

When you are in the New help style mode, you can display help information pertaining to a specific function panel by selecting **Function** from the **Help** menu in the Function Panel menu bar. Alternatively, you can click on the background of the desired function panel with the right mouse button to display the function panel help.

When you are in the New help style mode, you enter function panel help information from the Function Panel Editor. Follow these steps to add function panel help:

1. Activate the desired function panel.

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

3. Type appropriate help text into the Help dialog box.

Chapter 3, *The Function Tree Editor*, gives more information on changing between the New and Old style help modes.

Function Panel Window Help (Old Style Help Only)

In the Old help style mode, 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.

In the Old help style mode, you enter function panel window help information from the Function Panel Editor. Follow these steps to add function panel window help.

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

2. Type appropriate help text into the Help dialog box.

Chapter 3, *The 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 highlighting the control and selecting **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 Function help.

You enter Function help information from the Function Tree Editor or the Function Panel Editor.

Add help information for a function panel control as follows.

1. Highlight the desired control.
2. Select **Control Help** from the **Edit** menu in the Function Panel Editor menu bar. The Help Editor dialog box appears. Alternatively, you can click on the desired control with the right mouse button to display the Help Editor dialog box.
3. Type appropriate help text into the Help dialog box.

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. Select the **Function Tree (*.fp)** option of the **New** command from the **File** menu.
2. Select **Instrument** from the **Create** menu.
3. Type `Help Information Examples` as the Name and `help` as the Prefix. Click on **OK**.
4. Select **Class** from the **Create** menu.
5. Enter `Class 1` as the Name. Click on **OK**.

6. Highlight the line beneath the name `Class 1`.
7. Select **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 5-2.

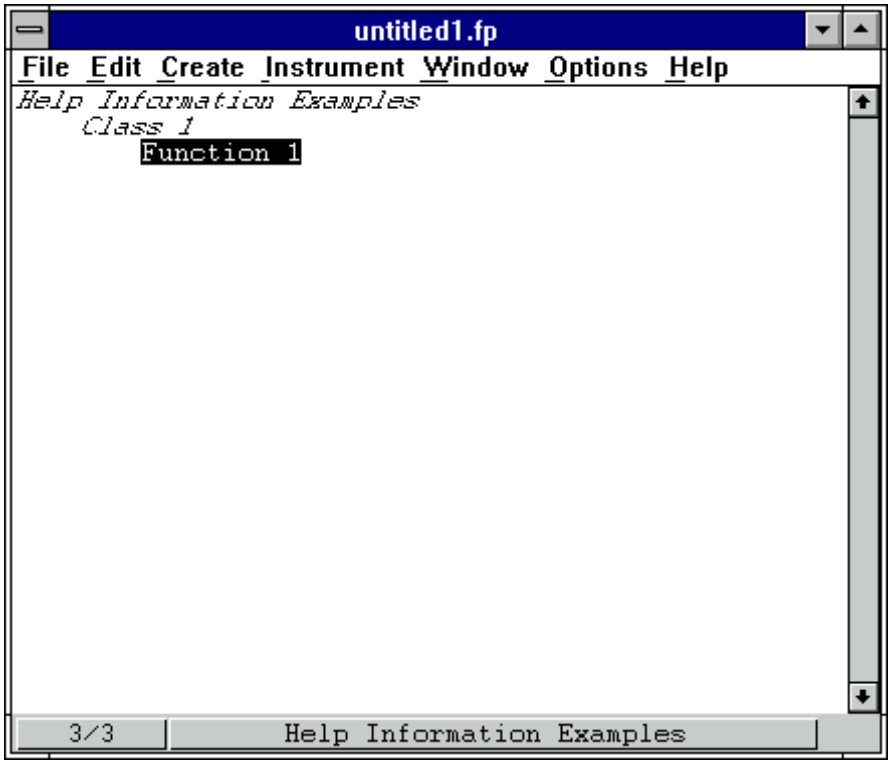


Figure 5-2. A Sample Function Tree

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. Position the highlight on the name `Help Information Examples`.
2. Select **Edit Help** from the **Edit** menu, or click on the instrument name with the right mouse button. The Help Editor dialog box appears.
3. Enter the following help information.

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

4. Select **Save .FP File** and then select **Close** from the **File** menu to save the text and remove the Help Editor dialog box.

Add help information to `Class 1` as follows.

1. Position the highlight on the name `Class 1`.
2. Select **Edit Help** from the **Edit** menu or click on the class name with the right mouse button. The Help Editor dialog box appears.
3. Enter the following help information.

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

4. Select **Save .FP File** and then select **Close** from the **File** menu to save the text and remove the Help Editor dialog box.

View the help information as follows.

1. Select `Help Information Examples` from the **Instrument** menu. The Select Function Panel dialog box appears.
2. Highlight `Class 1` and press Help to display the Class Help dialog box.
3. Press **Instrument Help** to display the Instrument Help dialog box.
4. Press **Done** to exit the Instrument Help dialog box.
5. Press **Done** to exit the Class Help dialog box.
6. Press **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. Select **Function Help** from the **Edit** menu. The Help Editor dialog box appears.
2. Enter the following help information.

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

3. Select **Save .FP File** and then select **Close** from the **File** menu to save the text and remove the Help Editor dialog box.

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

Add a control to the current panel as follows.

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

Now add help information to the control.

1. Highlight the control and select **Control Help** from the **Edit** menu. Alternatively, click the right mouse button on the control. The Help Editor dialog box appears.
2. Enter the following text in the Help Editor dialog box:

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

3. Select Save .FP File and then select Close from the File menu to save the text and remove the Help Editor dialog box.

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. Select **Save .FP file** and then select **Close** from the **File** menu to save the text.

7. With the highlight positioned on the Status control, select **Copy Controls** from the **Edit** menu.
8. Press <Ctrl-Page Down> to display the `Function 2` function panel.
9. Select **Paste** from the **Edit** menu. The Status control appears on the function panel.
10. Select **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. Select **Edit Function Panel** Window from the **Options** menu.
2. Select **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. Position the highlight on the Status control.
5. Select **Control Help** from the **Edit** menu or click the right mouse button on the control.
6. Highlight all of the text in the dialog box.
7. Select **Copy** from the **Edit** menu.
8. Select **Close** from the **File** menu.
9. Position the highlight on the Error control.
10. Select **Control Help** from the **Edit** menu or click the right mouse button on the control.
11. Select **Paste** from the **Edit** menu. The help information appears in the dialog box.
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.

Chapter 6
Programming Guidelines for Instrument Drivers

This chapter gives you guidelines for creating instrument drivers and using them with one another. If you write instrument drivers for general distribution to users, these guidelines ensure portability and proper operation. This chapter tells you how to create an instrument driver from a LabWindows/CVI core instrument driver.

Note: ***All instrument drivers in the LabWindows/CVI Instrument Library are based upon a core instrument driver. Each of the core drivers adheres to the programming guidelines outlined in this chapter.***

General Programming Guidelines

The following guidelines relate to general programming practices.

- Base your instrument driver on an existing instrument driver or one of the core instrument drivers.
- 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 and is specified when you create an instrument function tree.
- Use only the VISA (Virtual Instrument Software Architecture) I/O library to perform instrument I/O.
- Use only the VISA data types.
- 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 exporting global variables to the user. If you need to do so, define the global variables in the `.c` file and declare them as `extern` in the `.h` file.
- Declare `static` variables that are global to the instrument driver, but not needed by any other drivers.
- Avoid declaring large arrays within instrument drivers, because arrays use large amounts of memory.

- Use `FmtOut`, `printf`, and User Interface functions only in exception conditions. Ideally, no screen I/O should occur within an instrument driver.
- Avoid using Advanced Analysis Library functions in instrument drivers. Users would need to have the Advanced Analysis Library loaded on their computer.
- 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.c`, and `tek2430a.h`.
- Test all of the instrument drivers you create in LabWindows/CVI as standalone applications.
- 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 all user callable function prototypes with the macro `_VI_FUNC` before the function name. Declare all array and output parameters in user callable function with the macro `_VI_FAR` before the parameter name.

The Core Instrument Driver

To develop instrument drivers for GPIB, VXI, and RS-232, modify a LabWindows/CVI core instrument driver or modify an existing driver that is based upon a core instrument driver. You create a new instrument driver by changing a core instrument driver to match the requirements of your instrument. LabWindows/CVI gives a core instrument driver for each type of instrument. Develop your instrument driver using a core instrument driver as a foundation. Each core instrument contains the following files.

- The `.fp` file contains a function tree and function panels.
- The `.c` file contains the source code for developing the instrument program.
- The `.h` file contains function declarations and defined constants.

The core driver has a simple, flexible structure and a common set of functions to help you develop all types of instrument drivers. Also, the core driver has template functions for all of the required instrument driver operations. These template functions are based on IEEE 488.2 common commands. So if your instrument is IEEE 488.2 compliant, the core will require few modifications to create a baseline driver for your instrument. The core also includes modification instructions that make it easy to modify for use with non-IEEE 488.2 devices. The required instrument driver functions are described in detail in Chapter 7, *Required Instrument Driver Functions*.

The core also includes many useful functions, called *utility functions*, to create user callable functions and to implement a simple error handling scheme. These functions can check parameter ranges, and perform instrument I/O to and from a file. The utility functions also detect errors and update error information. The LabWindows/CVI core instrument driver includes the following utility functions.

- Check for Valid `ViInt16` Parameter. The `fl45_invalidViInt16Range` function determines whether a `ViInt16` parameter lies within an acceptable range, thus preventing the user from sending illegal commands to an instrument. The function accepts four parameters: the `ViInt16` value, the minimum, the maximum, and an error code. The return value indicates whether the parameter lies within the valid range.
- Check for Valid `ViInt32` Parameter. The `fl45_invalidViInt32Range` function determines whether a `ViInt32` parameter lies within an acceptable range, thus preventing the user from sending illegal commands to an instrument. The function accepts four parameters: the `ViInt32` value, the minimum, the maximum, and an error code. The return value indicates whether the parameter lies within the valid range.
- Check for Valid `ViReal64` Parameter. The `fl45_invalidViReal64Range` function determines whether a `ViReal64` parameter lies within an acceptable range, thus preventing the user from sending illegal commands to an instrument. The function accepts four parameters: the real value, the minimum, the maximum, and an error code. The return value indicates whether the parameter lies within the valid range.
- Check for Valid `ViBoolean` Parameter. The `fl45_invalidViBooleanRange` function determines whether a boolean parameter lies within an acceptable range, thus preventing the user from sending illegal commands to an instrument. The return value indicates whether the parameter lies within the valid range.

Note: ***In the preceding list, the functions named are from the Fluke 45 Digital Multimeter instrument driver. The utility functions used by the Fluke 45 instrument driver are declared at the beginning of the program listing and the code for the utility functions is at the end of the listing. The source code for some of the utility functions was omitted because it is not needed to implement the Fluke 45 instrument driver.***

Modifying the Core Driver

Your first step in developing the code for an instrument driver is to modify the core driver to represent your instrument. Most of the modifications concern the instrument prefix. All instrument driver functions have a prefix that identifies the instrument. For example, the instrument prefix for the Fluke 45 instrument driver is `fl45`. The instrument prefix is also the prefix in the names of all files associated with the instrument driver. As a default, the core instrument driver uses `PREFIX` as the instrument prefix which you must change to a prefix unique to your instrument driver.

Modify the core driver, also given in the source code of the core instrument driver, as follows:

1. Edit the core instrument driver source code, `core_gpb.c`. Change all occurrences of `PREFIX` to the prefix of your instrument.

2. The device-dependent commands in this file are marked with the comment `CHANGE`. Search for occurrences of this comment and make appropriate changes.

3. Save the modified source file as `instr_prefix.c`.

4. Edit the core instrument driver header file, `core_gpb.h`. Change all occurrences of the word `PREFIX` to the prefix of your instrument.

5. Save the modified header file as `instr_prefix.h`.

6. Edit the core instrument driver function panel file. In the Function Tree Editor window, edit the instrument descriptor as follows. Change the current name to the name of the instrument and change the prefix to `INSTR_PREFIX`.

7. Save the modified function panel file as `instr_prefix.fp`.

By following this procedure, you create a good framework from which you can develop your instrument driver.

The core instrument driver files are located in the subdirectory `INSTR`. Their names are shown in Table 6-1.

Table 6-1. Core Instrument Driver Files

Instrument Type	Filenames	Description

GPIB	`core_gpb.c`
	`core_gpb.h`
	`core_gpb.fp`
		C source code file
		Include file for `core_gpb.c`
		Instrument function panel file

VXI	`core_vxi.c`
	`core_vxi.h`
	`core_vxi.fp`
		C source code file
		Include file for `core_vxi.c`
		Instrument function panel file

RS-232	`core_232.c`
	`core_232.h`
	`core_232.fp`
		C source code file
		Include file for `core_232.c`
		Instrument function panel file

Adding User Callable Functions

Add user callable functions to your instrument driver to control the instrument operations that you wish to make available to users. All user callable functions have a function panel interface, and return error and status information. Before you write any code, develop the function panels for all the user callable functions. We recommend that you define the structure of the driver and each of the functions before you develop any code.

To add your new functions.

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

Inserting the function code in the source code is the most difficult step when you are adding a function to the instrument driver. To make this step easier, define a function code programming structure. You can divide code writing for instrument driver functions into the following programming steps.

1. Check input ranges of all parameters. (*Utility Function*)
2. Create the command string and write it to the instrument. (*VISA Function*)
3. Read and parse the data string from the instrument if the instrument has a response (*VISA Function*)

Copy and Paste

By copying and pasting utility routines and VISA routines, you can program your driver more efficiently. The following examples, which are based on the Fluke 45 instrument driver, illustrate how to paste the utility and VISA functions into your user callable functions.

1. A utility function to check input parameter ranges

   a. For integer parameters paste in the following code.

   ```
   if (fl45_invalidViInt16Range (val, min, max))
         return VI_ERROR_PARAMETER2;
   ```

   b. For real parameters paste in the following code.

   ```
   if (fl45_invalidViReal64Range (val, min, max))
         return VI_ERROR_PARAMETER3;
   ```

2. A VISA function to write the command string to the instrument

```
if ((fl45_status = viPrintf(instrHandle, ":SYST:ERR?")) < 0)
   return fl45_status;
```

3. A VISA function to read the data string from the instrument, if the instrument has a response

```
if ((fl45_status = viScanf (instrHandle, "%d,\"%[^\"]",
errCode, errMessage)) < 0)

    return fl45_status;
```

You can use the remaining VISA and utility functions in the same manner. These VISA and utility functions are the building blocks from which you build user callable functions. The Fluke 45 instrument driver is a good example of this programming style. Most of the code in an instrument driver consists of combinations of the previously described programming statements. Using previously described programming statements makes instrument driver functions easier to develop, and guarantees that your programming style matches the style of other drivers in the library.

Note: ***You may Delete any unused utility functions when you complete instrument driver development.***

Tips for Creating an Instrument Driver

An important step in developing a user callable instrument driver function is selecting between command strings to send to the instrument based on an input parameter. In the following example, the commands AC, DC, and GND set the vertical coupling of an imaginary oscilloscope. An integer input parameter selects from these different command strings. The following source code configures the vertical coupling of this instrument.

```
/*============================================================*/
/* This function configures the instrument                    */
/*============================================================*/
ViStatus _VI_FUNC scope_config_coup (ViSession instrHandle, ViInt16 func)
{
      if (scope_invalidViInt16Range (func, 0, 2) != 0)
            return VI_ERROR_PARAMETER2;
      switch (func) {
      case 0:
            scope_status = viPrintf (instrHandle, “AC”);
            break;
      case 1:
            scope_status = viPrintf (instrHandle, “DC”);
            break;
      case 2:
            scope_status = viPrintf (instrHandle, “GND”);
            break;
      }
      return scope_status;
}
```

Notice the large case statement needed to select the different commands. The more command options you need, the larger the case statement grows. The next example shows how you can

discard the case statement approach by defining a string array with the three different instrument commands. The configure function then uses the string array to build the command string.

```
/*============================================================*/
/* This function configures the instrument.                   */
/*============================================================*/
ViStatus _VI_FUNC scope_config_coup (ViSession instrHandle, ViInt16 func)
{
      static ViString cmd_arr[] = {"AC", "DC", "GND"};

      if (scope_invalidViInt16Range (func, 0, 2) != 0)
            return VI_ERROR_PARAMETER2;
      if ((scope_status = viPrintf (instrHandle, “%s”, cmd_arr[func])) < 0)
            return scope_status;
      return scope_status;
}
```

The first example requires a case structure with three different format statements, whereas the second example requires a single format statement that selects the appropriate command from the `cmd_arr` string array. By defining a string array with the instrument commands, you use a single format command in your user callable functions to keep the instrument driver compact and readable.

Developing Portable Instrument Drivers

An important consideration in developing an instrument driver is making the driver accessible by other compilers and operating systems. There are established guidelines for the development of portable instrument driver code. The main concerns in developing portable instrument driver code are given to data types, the declaration of user-callable functions and their output and array parameters, and the use of Scan and Formatting functions.

Instrument Driver Data Types

A subset of the VISA data types has been defined for use in the development of LabWindows/CVI instrument drivers and is accessible as user-defined data types. These special data types are used to define all of the parameters of instrument driver functions. 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.

Table 6-2. VISA Data Types

VISA Type Name	Definition
`ViInt16`	Signed 16-bit integer
`ViInt32`	Signed 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 and Array and Output Parameters

The VISA I/O library also defines macros that are useful for prototyping the user callable functions of an instrument driver. These macros are listed below.

Table 6-3. 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`	
`_VI_FAR`		`_far`		

The macros have been designed to resolve the differences on Windows between using instrument driver functions in a LabWindows/CVI environment as opposed to using them with an external compiler. The `_VI_FUNC` macro is used to define the calling conventions of a function.

The `_VI_FAR` macro is used to define all output and array parameters used in a user-callable function.

An example of an instrument driver function prototype using the above datatypes and macros is shown below.

```
ViStatus _VI_FUNC tek2430a_read_waveform (ViSession instrSession,
                                          ViReal64 _VI_FAR wvfm[],
                                          ViReal64 _VI_FAR *xin,
                                          ViReal64 _VI_FAR *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.

1. 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]`

2. 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.

3. 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 code example below shows the correct way to scan binary data read from an instrument. In the code example, the `viRead` function is used to transfer the binary waveform information from the instrument to the buffer `cmd`. Then a `Scan` function is used to parse the binary information and place it in the `ViInt16` array `wavefrm`. Notice that the `o` modifier is on the side of the `Scan` statement that represents the binary data that was received from the instrument and that a `b` modifier is used on both sides of the `Scan` function to represent the size of the binary data and the element size of the array where the data will be placed.

```
ViInt16 wavefrm[4000];
ViSession instrSession;
ViInt16 NumPoints;
ViUInt32 retCnt;
ViStatus scope_status;

if ((scope_status = viRead (instrSession, cmd, 1027, &retCnt)) < 0)
      return scope_status;
Scan (cmd, "%*d[zb2o10]>%*d[b2]", NumPoints, NumPoints, wavefrm);
```

Error Reporting Guidelines

One of the most important operations performed in an instrument driver is reporting the status of each operation. The core driver also gives you an error/warning handling scheme. Each user callable routine is a function with a return value of the type `ViStatus` which is used to return the appropriate error or warning value.

Table 6-2 presents a scheme for determining error values. It lists predefined error codes for instrument drivers.

Table 6-4. 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.

Table 6-5. Instrument Driver Completion and Warning Codes

Completion Code	Description	Error Number
VI_SUCCESS	No error: the call was successful	
VI_WARN_NSUP_ID_QUERY	Identification query not supported	0x3FFC0101L
VI_WARN_NSUP_RESET	Reset not supported	0x3FFC0102L
VI_WARN_NSUP_SELF_TEST	Self-test not supported	0x3FFC0103L
VI_WARN_NSUP_ERROR_QUERY	Error query not supported	0x3FFC0104L
VI_WARN_NSUP_REV_QUERY	Revision query not supported	0x3FFC0105L
	Instrument-specific warnings	0x3FFC0800 to 0x3FFC0FFF

Table 6-6. 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 Table 6-3 and 6-4 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.

An important error situation is error `VI_ERROR_INV_RESPONSE` (*Error in interpreting an instrument response).* This error is detected when a `Scan` statement tries to parse data from an erroneous instrument response. The user-callable function `tek2430a_read_waveform` in the Tektronix 2430a example instrument driver shown in Appendix A, gives a good example of this type of error detection and reporting feature.

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

In the previous code, error `VI_ERROR_INV_RESPONSE` is returned if the scan does not place data in the variables, `header` and `wvfm`.

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 end 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 error return control in the lower right corner of every function panel. Place the instrument ID control in the lower left corner. 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. Multimeter drivers, oscilloscopes, and function generators should resemble each other.

For example, the Fluke 45 instrument driver function tree is shown in Figure 6-1.

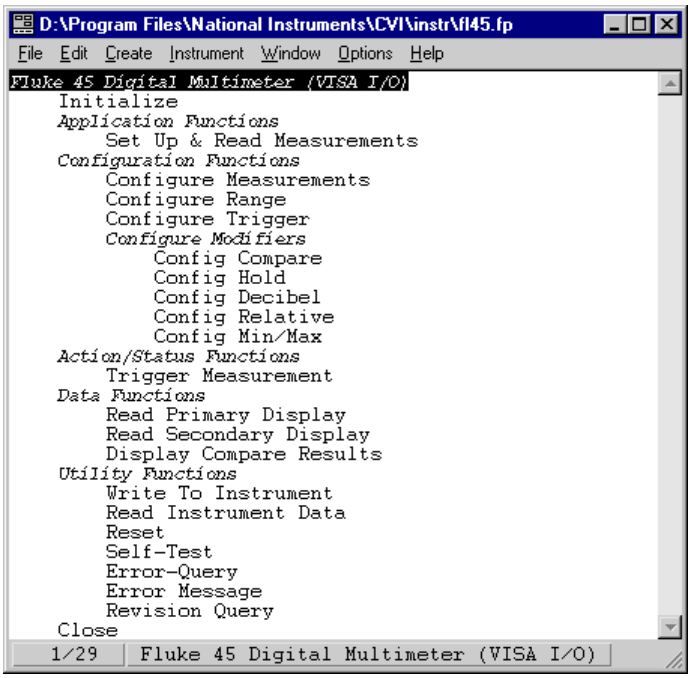


Figure 6-1. The Fluke 45 Digital Multimeter Function Tree

The functions are easy to understand and immediately incorporate into an application program. Be sure to develop your function tree and function panels *before* you develop any code for your instrument driver. 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 to understand the instrument driver and its functions. Instrument driver documentation should consist of the following.

- Online help from within LabWindows/CVI function trees and function panels
- A `.doc` file distributed on the disk with the instrument driver files

Online Help

Users consult the online help of an instrument driver most 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* appears in dialog boxes when a user views help for a function panel window or function class. This type of help describes the instrument driver and lists the functions and classes of functions in the driver. Figure 6-2, shows instrument help for the Fluke 45 instrument driver.

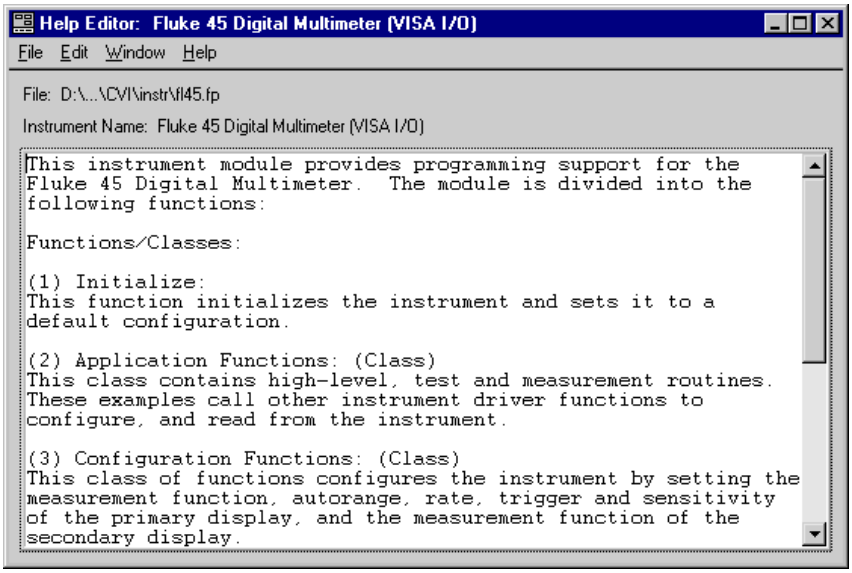


Figure 6-2. The Fluke 45 Instrument Help

- *Function class help* is available from the instrument driver pull-down menu after the function class has been selected. Function class help briefly describes all the functions and subclasses beneath the selected function class. Figure 6-3, shows function class help from the Fluke 45 instrument driver.

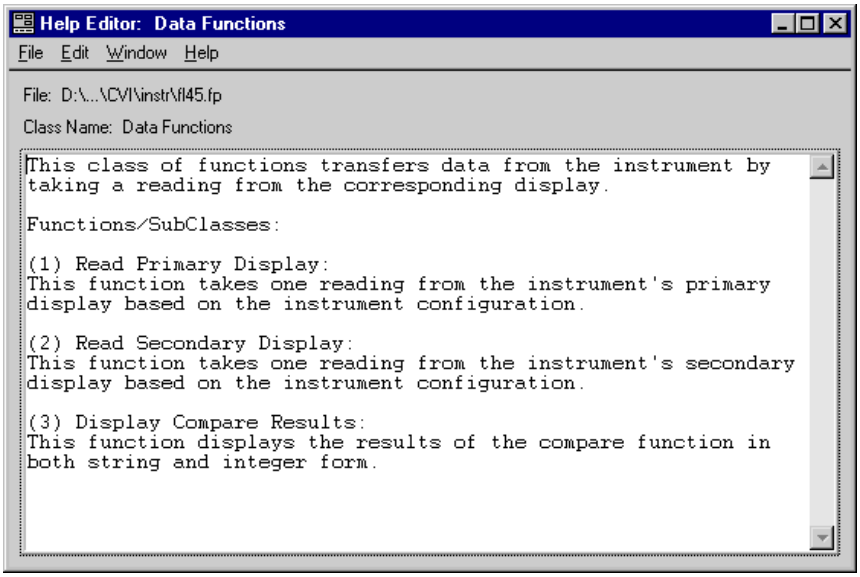


Figure 6-3. The Fluke 45 Function Class Help

- *Function panel help* is available from the **Help** menu in the Function Panel menu bar. Function panel help describes the function call. Figure 6-4, shows the function panel help from the Fluke 45 instrument driver.

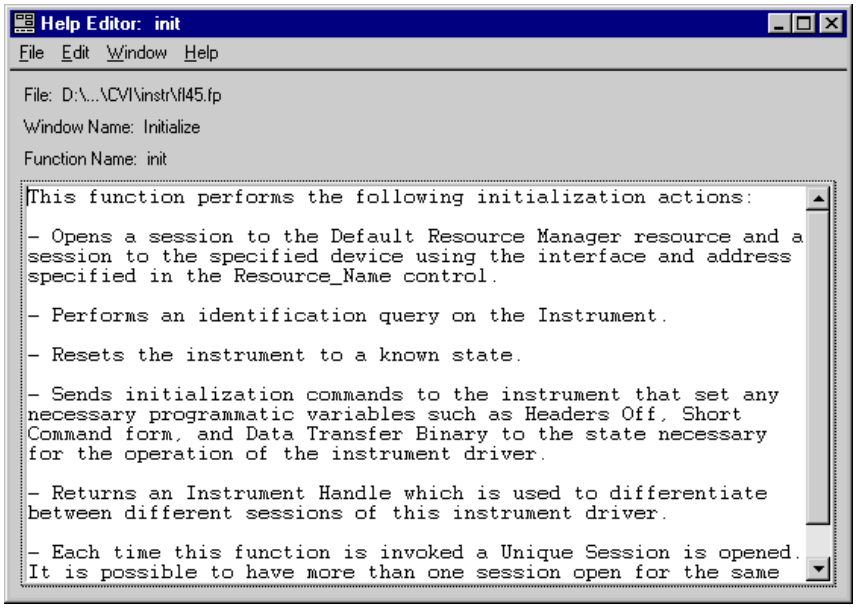


Figure 6-4. The Fluke 45 Function Panel Help

- *Control help* is available from the **Help** menu in the Function Panel. Control help contains a description of the parameter, the valid range, and the default value. Figure 6-5, shows an example of function panel control help from the Fluke 45 instrument driver.

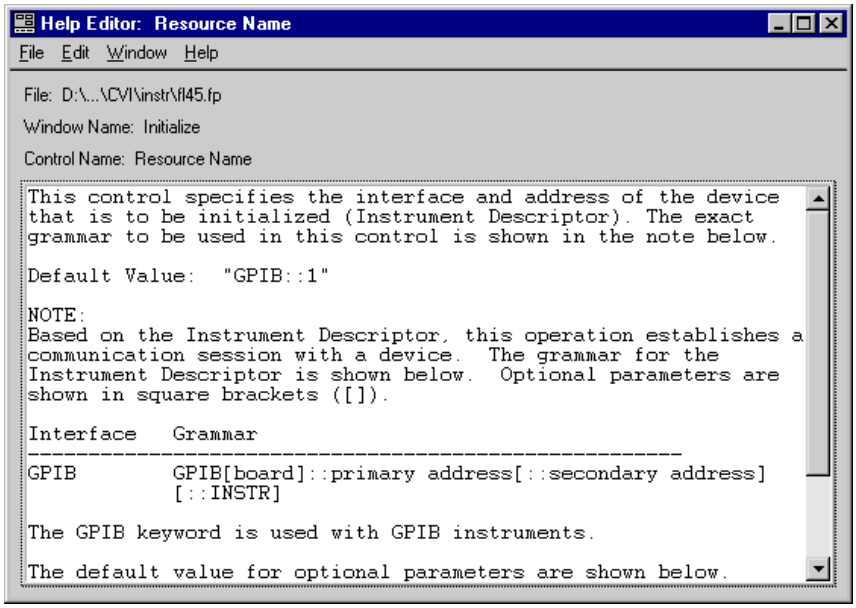


Figure 6-5. The Fluke 45 Function Panel Control Help

- *Status help* is available from the **Help** menu in the Function Panel. Error help contains a description of the parameter and the possible error values. Figure 6-6, shows an example of status control help from the Fluke 45 instrument driver.

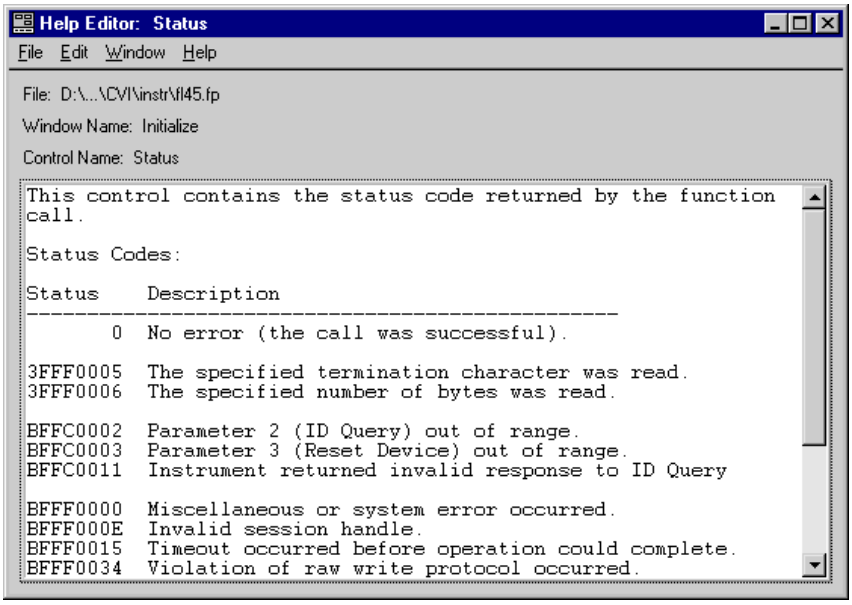


Figure 6-6. The Fluke 45 Function Panel Error Control Help

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 global variables and constant names declared in the driver
- 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

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 RS-232 Instruments

Initialization Routine

The initialization routine for an RS-232 instrument driver must open a `com` port using the `OpenComConfig` function to specify such parameters as baud rate and parity, and set the timeout value appropriately.

In addition, the initialization routine should perform the following steps.

1. Clear the instrument.
2. Query current operating status.
3. Send commands to place the instrument in a known state.

The initialize routine in the instrument driver core file `core_232.c` handles these operations and can be modified for your instrument driver.

Close Routine

The close routine for an RS-232 instrument driver closes the `com` port used by the driver. The routine should set the appropriate variables to zero to indicate that the instrument driver has been closed. The instrument driver core file `core_232.c` implements a close routine that can be modified for your instrument driver.

Utility Routines

The instrument driver core file `core_232.c` contains a number of utility functions. These functions perform operations common to most RS-232 instruments, including reading and writing to an RS-232 instrument.

Programming Guidelines for VXI Instruments

Use the VXI core instrument driver to develop drivers for all message-based and register-based VXI devices. Drivers developed with the VXI core instrument driver can control a VXI device from either a GPIB-VXI translator, a MXI-equipped computer interface, or an embedded VXI controller.

Instrument Driver Checklist

All instrument drivers you add to the LabWindows/CVI Instrument Library must conform to our 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. The structure is logical and follows the instrument driver internal design model.

____ B. All required instrument driver functions are implemented (initialize, reset, self-test, revision, error query, and error message).

____ C. Help exists for the instrument and all functions and classes.

II. Function Panels

____ A. The controls are neatly organized.

____ B. The instrument error control is in the lower right corner.

____ C. The instrument ID control is in the lower left corner.

____ D. The proper defaults are set for each control.

____ E. The return value is consistently used for error reporting.

____ F. Notes, if any, are understandable.

____ G. The proper display format is used, such as hexadecimal for status registers, and so on.

____ H. Help:

- ____ 1. Exists for all controls.
- ____ 2. Is well formatted and includes:
  - ____ a. Description
  - ____ b. Default value and valid range
  - ____ c. All needed error codes

III. Source Code

____ A. Standard instrument driver header comments are used:

- ____ 1. Author
- ____ 2. Original language
- ____ 3. Modifications history

____ B. Only the utility functions are declared.

____ C. All instrument driver functions make proper use of the utility functions:
- ____ 1. All parameter ranges are checked (`_invalidViInt16Range` and `_invalidViReal64Range`).
- ____ 2. All scans check for and report errors correctly.

____ D. Errors are reported and correct error codes are used.

____ E. Complete and descriptive comments are included.

____ F. Reference tables are properly used.

____ G. `visa.h` is included.

____ H. Binary instrument data is scanned or formatted correctly for multi-platform use.

____ I. Prototypes for user callable functions are correctly formatted:
- ____ 1. All function prototypes include the macro `_VI_FUNC` before the function name.
- ____ 2. Use only VISA datatypes for function parameters.
- ____ 3. All array parameters include the macro `_VI_FAR` before the parameter name.

IV. Include File

____ A. Only user callable instrument driver functions are declared.

____ B. The file `vpptype.h` is included.

____ C. Prototypes correctly formatted:
- ____ 1. All function prototypes include the macro `_VI_FUNC` before the function name.
- ____ 2. Use only VISA datatypes for function parameters.
- ____ 3. All array parameters include the macro `_VI_FAR` before the parameter name.

V. Document File

____ A. The LabWindows/CVI-generated document file is properly edited.

____ B. The document file contains no redundant information such as variable name and variable type.

Chapter 7
Required Instrument Driver Functions

This chapter describes the implementation of the required instrument driver functions of a LabWindows/CVI instrument driver. For each required instrument driver function, the following information is presented; the C function prototype, a description of the purpose and operation of the function, a table defining each parameter, all possible completion and error codes, and any special implementation requirements.

The required instrument driver functions are as follows.

- Initialize
- Close
- Reset
- Self-Test
- Error Query
- Error Message
- Revision Query

For each function, the following information is given.

- **Prototype**—The C function prototype
- **Description**—A desription of the purpose and general operation of the function
- **Parameters**—A table defining each parameter. Information includes the name of each parameter, the direction (input or output), the data type, and a description
- **Return Values**—Lists possible completion and error codes. For each a definition is specified along with a description and who is responsible for setting this code (either the instrument driver or the VISA I/O library)
- **Implementation Requirements**—Any special implementation requirements that should be considered when creating this function for a particular instrument driver

PREFIX_init

```
ViStatus status = _VI_FUNC  PREFIX_init (ViRsrc rsrcName, ViBoolean id_query,
                                         ViBoolean reset,
                                         ViSession _VI_FAR *vi);
```

Purpose

The `PREFIX_init` function is the first function called when you access an instrument driver. It configures the communications interface and sends a default setup command string to the instrument. Typically, the default setup configures the instrument's operation for the rest of the driver (such as turning headers on or off or using long or short form for queries). Upon successful operation, the initialize function returns a session that is used to address the instrument in all subsequent instrument driver functions.

The `PREFIX_init` function has an instrument descriptor string as an input. Based on the syntax of this input, it configures the I/O interface and generates an instrument session that is used by all other instrument driver functions. The grammar for the instrument descriptor is shown below. Optional parameters are shown 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]

The GPIB keyword is used with GPIB instruments. The VXI keyword is used for either embedded or MXIbus controllers. The GPIB-VXI keyword is used for a GPIB-VXI controller.

Additionally, the `PREFIX_init` function can perform selectable ID query and reset operations. It is helpful if the ID query and reset operations are user selectable, because a user can disable the ID query when he or she attempts to use the driver with a similar instrument but does not need to modify the driver source code. Also, a user can enable or disable the reset operation, and this action is useful for debugging when resetting would take the instrument out of the state the user was trying to test.

Parameters

Input	**rsrcName**	ViRsrc	Instrument Description
			Examples:
			`VXI::5`
			`GPIB-VXI::128::INSTR`

	id_query	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	**vi**	ViSession	Unique logical identifier reference to a session

Return Values

Type 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	id_query 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

Verifying the identity can be accomplished by checking the manufacturer ID and model number in the instrument's VXI register set by using the `*IDN` query for IEEE 488.2 compatible instruments, or by other means. If your instrument cannot perform an identification query or be programmatically reset to a known state, their corresponding parameters must still be provided in the `PREFIX_init` function, but they can be ignored.

If the `PREFIX_init` function encounters an error, the value of the **vi** output parameter should be `VI_NULL` and any valid sessions obtained from `viOpen` should be closed.

PREFIX_close

`ViStatus` **status** = _VI_FUNC **PREFIX_close** (`ViSession` **vi**);

Purpose

All LabWindows/CVI instrument drivers include a Close function that terminates the software connection to the instrument and deallocates system resources. Additionally, the developer may select to place the instrument in an idle state. For example, the developer of a switch driver may disconnect all switches when he or she closes the instrument driver.

Parameter

Input	vi	ViSession	Unique logical identifier to a session with an instrument

Return Values

Type 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

PREFIX_reset

`ViStatus` **status** = _VI_FUNC **PREFIX_reset** (`ViSession` **vi**);

Purpose

The `PREFIX_reset` function programmatically 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. You can either call the `PREFIX_reset` function separately, or you can select it to be called from the `PREFIX_init` function.

Parameter

Input	**vi**	ViSession	Unique logical identifier to a session with an instrument

Return Values

Type 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
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

The default state that the Reset function places the instrument in should be documented in the help information for the Reset function.

PREFIX_self_test

```
Vistatus status = _VI_FUNC  PREFIX_self_test(ViSession vi,
                                             ViInt16 _VI_FAR * test_result,
                                             ViChar _VI_FAR test_message[]);
```

Purpose

All LabWindows/CVI instrument drivers have a Self-Test function. The `PREFIX_self_test` function forces the instrument to perform a self-test. The `PREFIX_self_test` function waits for the instrument to complete the test, then queries the instrument for the results of the self-test and returns the results to the user.

Parameter

Input	**vi**	ViSession	Unique logical identifier to a session with an instrument
Output	**test_result**	ViInt16	Numeric result from self-test operation 0 = no error (test passed)
	test_message	ViChar[]	Self-test status message

Return Values

Type 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

If your 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 _VI_FAR *` **error,**
`ViChar _VI_FAR` **error_message**`[]`);

Purpose

All LabWindows/CVI instrument drivers have an Error Query function. This function queries the instrument and returns the instrument-specific error information.

Parameter

Input	**vi**	ViSession	Unique logical identifier to a session with an instrument
Output	**error**	ViInt32	Instrument error code
	error_message	ViChar[]	Instrument error message

Return Values

Type `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

If your 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` **error**,
`ViChar _VI_FAR` **message**`[]`);

Purpose

LabWindows/CVI instrument drivers have an Error Message function. This function translates the error return value from a LabWindows/CVI instrument driver function to a user-readable string.

Parameter

Input	**vi**	ViSession	Unique logical identifier to a session with an instrument
	error	ViStatus	Instrument driver error code
Output	**message**	ViChar[]	Instrument driver error message

Return Values

Type `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

The `PREFIX_error_message` function should accept a value of `VI_NULL` for the **vi** input parameter. If the value `VI_NULL` is passed into the function, the **vi** parameter is ignored; otherwise the value of the **vi** parameter may be used by the function. This allows the `PREFIX_error_message` function to execute even if the `PREFIX_init` function fails.

PREFIX_revision

```
ViStatus status = _VI_FUNC PREFIX_revision (ViSession vi,
                                            ViChar _VI_FAR driver_rev[],
                                            ViChar _VI_FAR instr_rev[])
```

Purpose

All LabWindows/CVI instrument drivers have a Revision function. This function outputs the following.

- The revision of the instrument driver
- The firmware revision of the instrument being used

Parameter

Input	**vi**	ViSession	Unique logical identifier to a session with an instrument
Output	**driver_rev**	ViChar[]	Instrument driver revision
	instr_rev	ViChar[]	Instrument firmware revision

Return Values

Type `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

If the instrument firmware revision cannot be queried, the Revision function returns the literal string "Not Available" in the **instr_rev** output parameter, and the function returns the warning VI_WARN_NSUP_REV_QUERY.

Chapter 8
Instrument Driver Example

This chapter shows you how to create a complete GPIB instrument driver. The example presented in this chapter can serve as a model for your own instrument driver development.

The steps you will learn in this chapter include

- Modifying the file `core_gpb.fp` to create the function tree and panels for the new driver.
- Modifying the files `core_gpb.c` and `core_gpb.h` to create the instrument program for the new driver.
- Loading and testing the driver.

Example—Creating a GPIB Instrument Driver

This example illustrates all of the steps for creating a complete GPIB instrument driver. An overview of the procedure appears in the following list.

- Modify the file `core_gpb.fp` to create the function tree and panels for the new driver.
- Modify the files `core_gpb.c` and `core_gpb.h` to create the instrument program for the new driver.
- Load and test the instrument driver.

The instrument used in this example is a Tektronix 2430A oscilloscope. For simplicity, only the following functions are created.

- Initialize
- Configure vertical sensitivity and horizontal timebase
- Read waveform
- Close

In many cases, you do not need to start from the beginning of the procedure as done in this example. You can frequently modify an existing driver for a similar instrument.

Creating the Function Tree

To create the instrument driver, you first create the function tree using the Function Tree Editor. To invoke the Function Tree Editor, select the **Function Tree (*.fp)** option from either the **New** or **Open** commands in the **File** menu.

Use the file `CORE_GPB.FP`, located in the `INSTR` subdirectory, as a template for building your instrument driver. Load this file using the **Open** command in the **File** menu. The screen appears as shown in Figure 8-1.

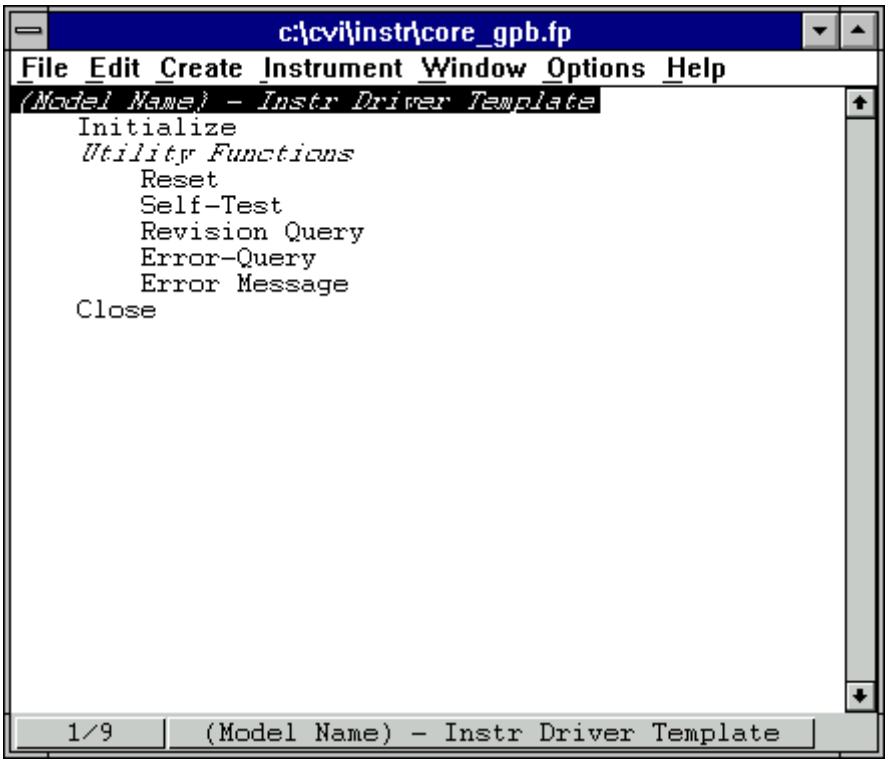


Figure 8-1. The Function Tree for CORE_GPB.FP

Modify the instrument name at the top of the function tree as follows.

1. Position the highlight on the statement `(Instrument Model Name)`.
2. Select **Edit Node** from the **Edit** menu.
3. Complete the Edit Instrument Node dialog box as follows: Type `Tektronix 2430A Oscilloscope` in the Name box, and `tek2430a` in the Prefix box.
4. Select **OK**.
5. With the highlight still on the instrument name, select **Edit Help** from the **Edit** menu, or click on the name with the right mouse button.

6. To modify the help information:

   a. Change the phrase `(Instrument Name)` to `Tektronix 2430A Oscilloscope`.

   b. Insert the following text between the Initialize and Utility Functions descriptions.

```
2. Configure - Set the volts per division and timebase of
        the oscilloscope

3. Read waveform - Read a waveform from the oscilloscope
```

   c. Give Utility Functions the number 4 and Close the number 5.

7. Select **Save .FP File As** from the **File** menu and save the file as `TEK2430A.FP`.

Add two new functions to the function tree as follows.

1. Position the highlight on `Initialize`.

2. Select **Function Panel Window** from the **Create** menu.

3. Complete the Create Function Panel Window Node dialog box as follows.

   Type `Configure` in the Name box and `config` in the Function box.

4. Select **OK**.

5. Select **Function Panel Window** from the **Create** menu.

6. Complete the Create Function Panel Window Node dialog box as follows.

   Type `Read Waveform` in the Name box and `read_waveform` in the Function Name box.

7. Select **OK**.

The function tree should now look like the one in Figure 8-2.

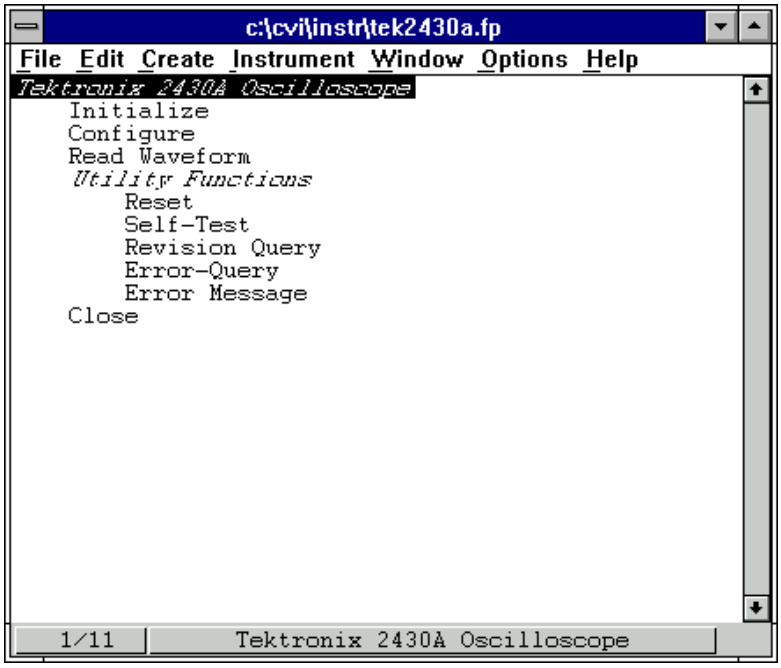


Figure 8-2. The New Function Tree for the Tektronix 2430A Instrument Driver

Select **Save .FP File** from the **File** menu.

Creating the Configure Function Panel Window

Add help information to the Configure function panel as follows.

1. Position the highlight on `Configure` and select **Edit Function Panel Window** from the **Edit** menu.

2. Select **Function Help** from the **Edit** menu.

3. Type the following help text.

```
Configures the vertical volts per division and horizontal timebase of the
oscilloscope.

The Configure function configures the scope so that only the channel
specified by the Channel control is displayed, i.e. the vertical mode is
either channel 1 only or channel 2 only.
```

4. Select **Save .FP File** and then select **Close** from the **File** menu of the Help Editor dialog box.

Add an Instrument Handle control to specify which instrument to talk to as follows.

1. Press <Ctrl-Page Down> twice to display the **Reset** function panel.

2. Position the highlight on the Instrument Handle control.
3. Select **Copy Controls** from the **Edit** menu.
4. Press <Ctrl-Page Up> twice to display the **Configure** function panel.
5. Select **Paste** from the **Edit** menu to place a copy of the Instrument Handle control on the Configure panel.
6. Position the Instrument Handle control in the lower left corner of the panel.
7. Select **Save .FP File** from the **File** menu.

Add a control for specifying the channel to configure as follows.

1. Select **Binary** from the **Create** menu.
2. Complete the Edit Binary Control and Edit On/Off Settings dialog boxes as shown in Figures 8-3 and 8-4.

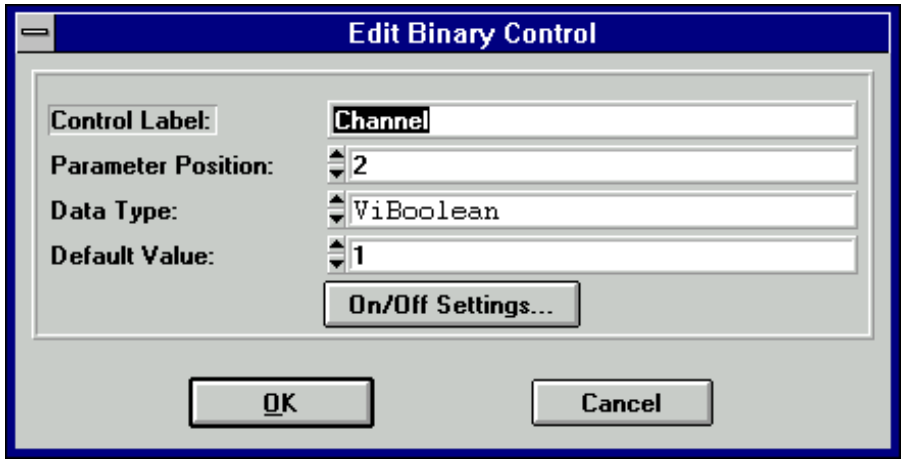


Figure 8-3. The Edit Binary Control Dialog Box

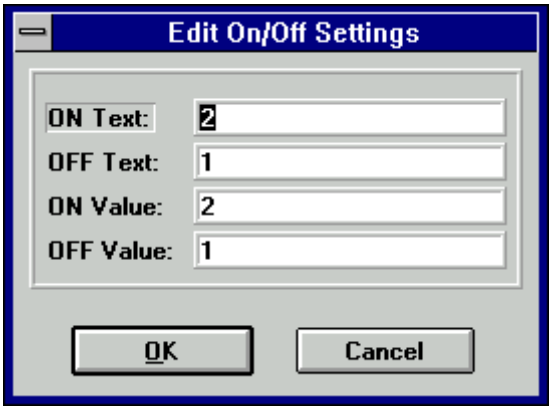


Figure 8-4. The Channel Edit On/Off Settings Dialog Box

3. Select **OK** twice.

4. Position the Channel control in the upper left portion of the panel.

Add help to the Channel control as follows.

1. Highlight the Channel control and select **Control Help** from the **Edit** menu.

2. Enter the following text in the Edit Help dialog box.

```
Specifies the channel to that Volts/Div and Coupling apply.  Channel also
indicates the mode in that to place the scope – channel 1 only or channel 2
only.

   Valid Range:   1 - Channel 1
                  2 - Channel 2
```

3. Select **Save .FP File** and then select **Close** from the **File** menu.

Add a control for specifying the vertical volts-per-division as follows.

1. Select **Ring** from the **Create** menu.

2. Complete the Edit Ring Control dialog box as shown in Figure 8-5.

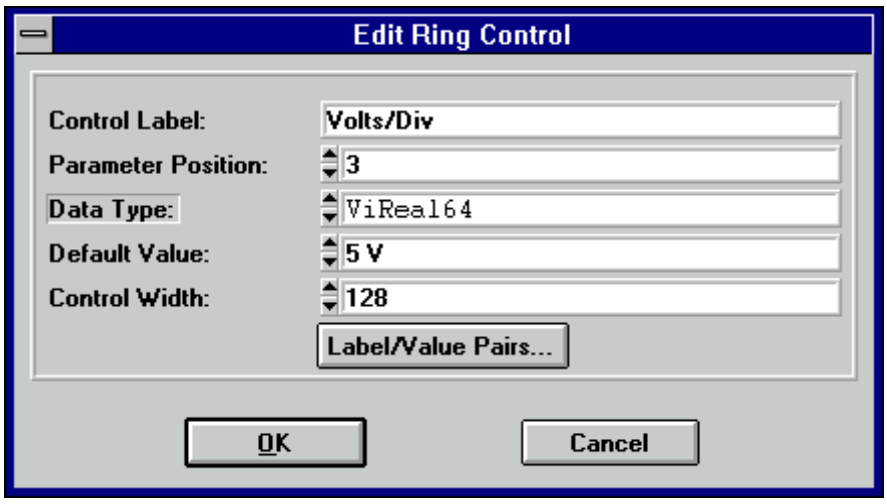


Figure 8-5. The Edit Ring Control Dialog Box for the Volts/Div Ring Control

3. Press the **Label/Value Pairs** button.

4. Complete the Edit Label/Value Pairs dialog box as shown in Figure 8-6.

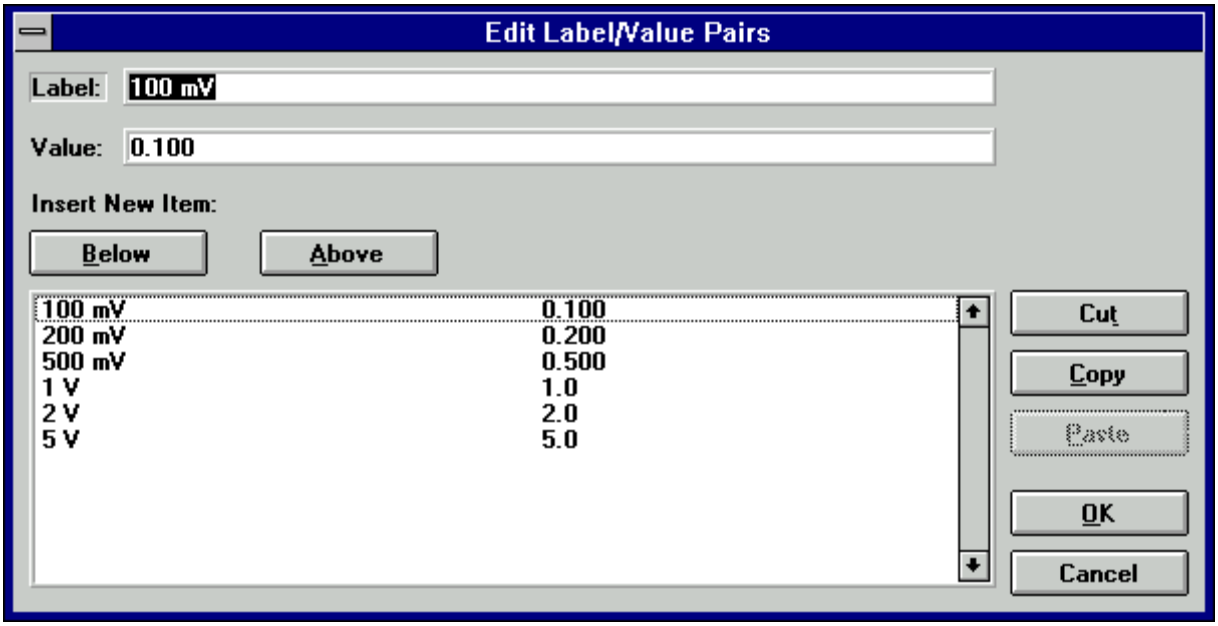


Figure 8-6. The Volts/Div Ring Control Edit Label/Value Pairs Dialog Box

5. Select **OK** twice.

6. Position the Volts/Div control in the upper middle portion of the panel.

Add help to the Volts/Div control as follows.

1. Highlight the Volts/Div control and select **Control Help** from the **Edit** menu.

2. Enter the following text in the Help Editor dialog box.

```
Specifies the volts/division setting of the channel specified by the
Channel control.

Valid Range:   100 mV to 50 V
```

3. Select **Save .FP File** and then select **Close** from the **File** menu.

Add a control for specifying the horizontal timebase as follows.

1. Select **Ring** from the **Create** menu.

2. Complete the Edit Ring Control dialog box as shown in Figure 8-7.

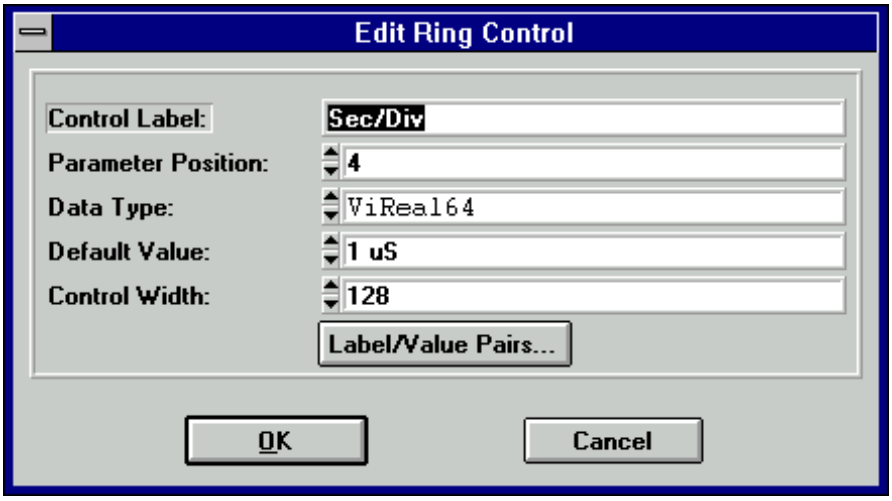


Figure 8-7. The Edit Ring Control Dialog Box

3. Press the **Label/Value Pairs** button.

4. Complete the Edit Label/Value Pairs dialog box as shown in Figure 8-8.

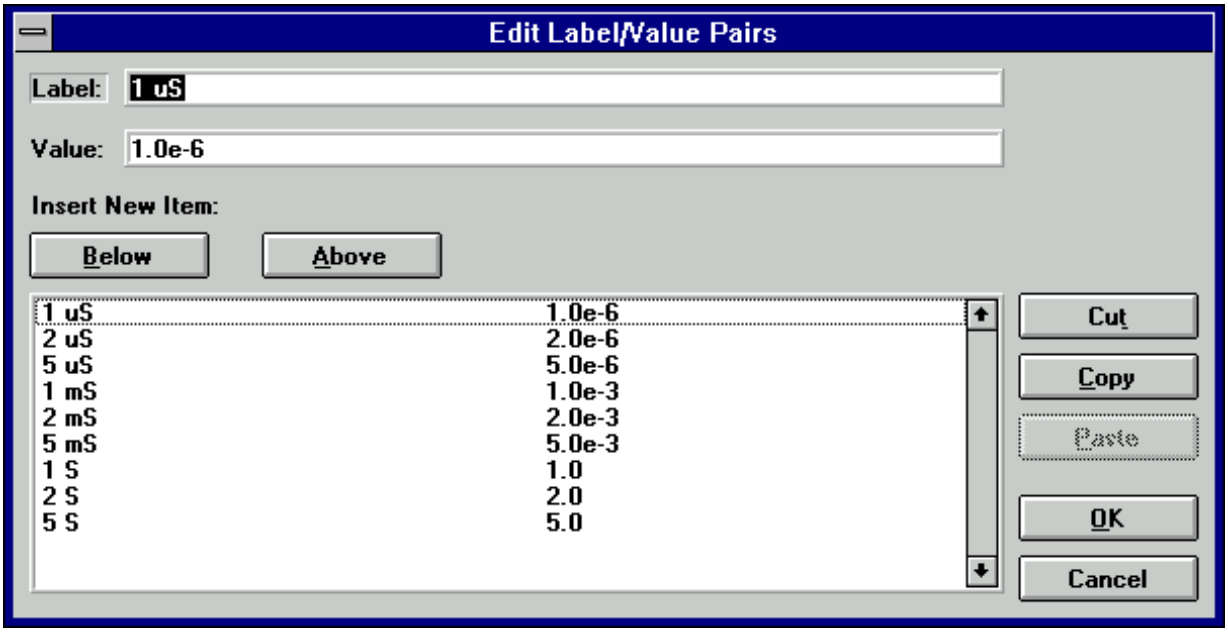


Figure 8-8. The Edit Label/Value Pairs Dialog Box

Note: ***Figure 8-7 shows an abbreviated listing of the possible timebase settings of the 2430A. You can specify other values when operating the panel by selecting Toggle Control Style and entering the desired value.***

5. Select **OK** twice.

6. Position the Sec/Div control in the upper right portion of the panel.

Add help to the Sec/Div control as follows.

1. Highlight the Sec/Div control and select **Control Help** from the **Edit** menu.
2. Enter the following text in the Help Editor dialog box.

```
Specifies the seconds/division setting for the main (A) timebase of the
oscilloscope

Valid Range:   10 nS to 5 S
```

3. Select **Save .FP File** and then select **Close** from the **File** menu.

Add a return control to indicate errors as follows.

1. Press <Ctrl-Page Up> to display the Initialize function panel.
2. Position the highlight on the Error control.
3. Select **Copy Controls** from the **Edit** menu.
4. Press <Ctrl-Page Down> to display the Configure function panel.
5. Select **Paste** from the **Edit** menu to place a copy of the Error control on the Configure panel.
6. Position the Error control in the lower right corner of the panel.

Add help to the Error control as follows.

1. Select **Control Help** from the **Edit** menu.
2. Modify the text in the Help Editor dialog box so that it appears as follows.

```
Reports the status of the function call.

Status Codes:
Status                      Description
-------------------------------------------------------------
VI_SUCCESS                  No error (the call was successful).
VI_ERROR_PARAMETER2         IDQuery parameter out of range.
VI_ERROR_PARAMETER3         reset parameter out of range.
VI_ERROR_INV_SESSION        The session is invalid.
VI_ERROR_TMO                Timeout expired.
VI_ERROR_RAW_WR_PROT_VIOL   Violation of raw write protocol.
VI_ERROR_BERR               Bus error occurred.
VI_ERROR_NCIC               Not the controller in charge.
VI_ERROR_NLISTENERS         No Listeners.
```

3. Select **Save .FP File** and then select **Close** from the **File** menu.

The Configure function panel window should now appear as shown in Figure 8-9.

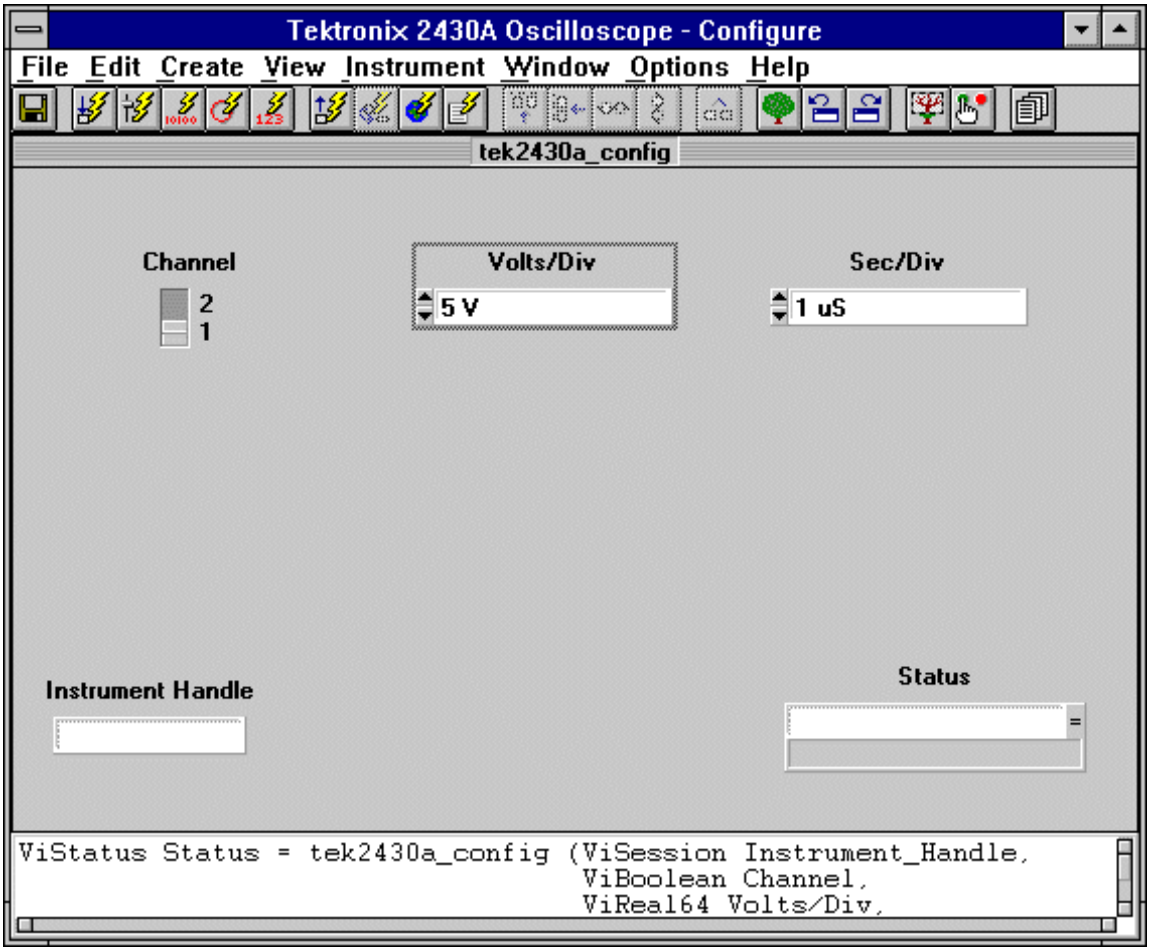


Figure 8-9. The Complete Configure Function Panel Window

Select **Save .FP File** from the **File** menu and save the function panels using the default filename, `tek2430a.fp`.

Creating the Read Waveform Function Panel

Press <Ctrl-Page Down> to move to the Read Waveform function panel.

Add help information to the panel as follows.

1. Select **Panel Help** from the **Edit** menu.
2. Enter the following help text.

```
Reads a waveform from the current acquisition channel of the 2430A.

Also returns the sampling period and trigger offset.
```

3. Select **Save .FP File** and then select **Close** from the **File** menu of the Help Editor dialog box.

Add an Instrument Handle control to specify which instrument to talk to as follows.

1. Press <Ctrl-Page Down> to display the **Reset** function panel.
2. Position the highlight on the Instrument Handle control.
3. Select **Copy Controls** from the **Edit** menu.
4. Press <Ctrl-Page Up> to display the **Read Waveform** function panel.
5. Select **Paste** from the **Edit** menu to place a copy of the Instrument Handle control on the Read Waveform panel.
6. Position the Instrument Handle control in the lower left corner of the panel.
7. Select **Save .FP File** from the **File** menu.

Add a control for specifying the waveform array as follows.

1. Select **Output** from the **Create** menu.
2. Complete the Create Output Control dialog box as shown in Figure 8-10.

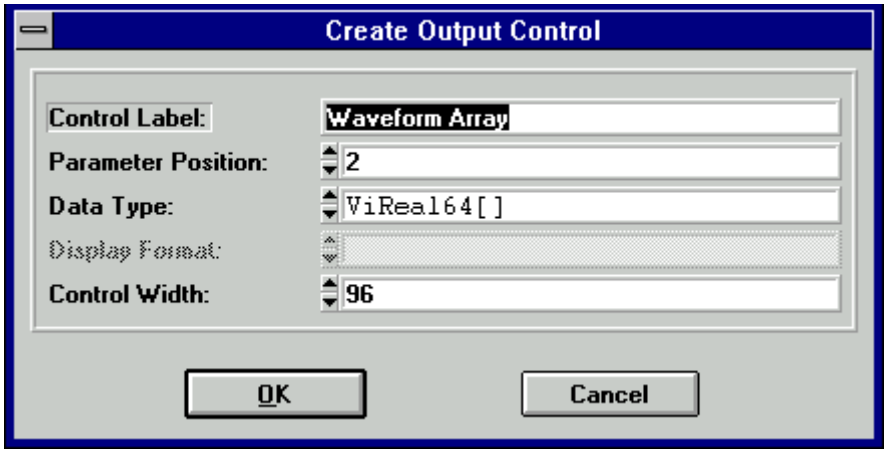


Figure 8-10. The Waveform Array Create Output Control Dialog Box

3. Press **OK**.
4. Position the Waveform Array control in the upper left portion of the panel.

Add help to the Waveform Array control as follows.

1. Highlight the Waveform Array control and select **Control Help** from the **Edit** menu.

2. Enter the following text in the Help Editor dialog box.

```
Specifies the name of the array in which to store the waveform values.
The dimension of the array must be greater than or equal to 1024 elements.
```

3. Select **Save .FP File** and then select **Close** from the **File** menu.

Add a control for displaying the sample period as follows.

1. Select **Output** from the **Create** menu.

2. Complete the Create Output Control dialog box as shown in Figure 8-11.

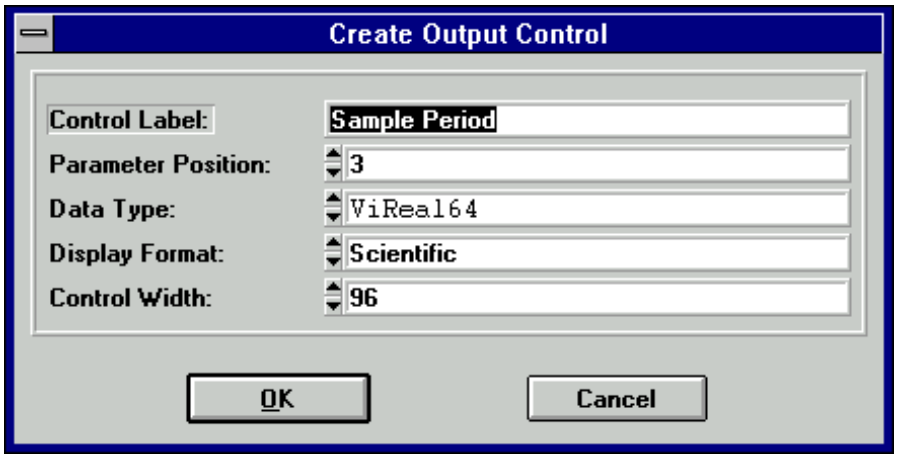


Figure 8-11. The Sample Period Create Output Control Dialog Box

3. Press **OK**.

4. Position the Sample Period control in the upper middle portion of the panel.

Add help to the Sample Period control as follows.

1. Highlight the Sample Period control and select **Control Help** from the **Edit** menu.

2. Enter the following text in the Help Editor dialog box.

```
Displays the sample rate in seconds at which the waveform was captured.
```

3. Select **Save .FP File** and then select **Close** from the **File** menu.

Add a control for displaying the trigger offset as follows.

1. Select **Output** from the **Create** menu.

2. Complete the Create Output Control dialog box as shown in Figure 8-12.

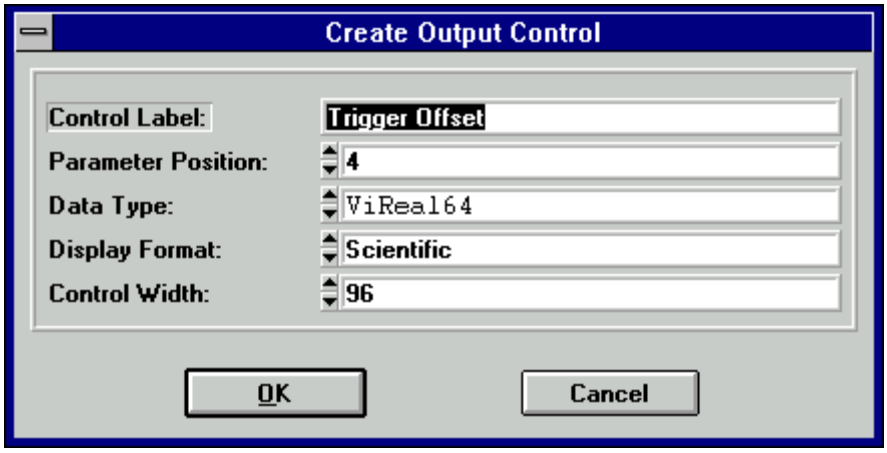


Figure 8-12. The Trigger Offset Create Output Control Dialog Box

3. Press **OK**.
4. Position the Trigger Offset control in the upper right portion of the panel.

Add help to the Trigger Offset control as follows.

1. Select **Control Help** from the **Edit** menu.
2. Type the following text in the Help Editor dialog box.

```
Displays the trigger offset of the waveform in seconds.
```

3. Select **Save .FP File** and then select **Close** from the **File** menu.

Add a return control to indicate errors as follows.

1. Press <Ctrl-Page Up> to display the Configure function panel.
2. Position the highlight on the Error control.
3. Select **Copy Controls** from the **Edit** menu.
4. Press <Ctrl-Page Down> to display the Read Waveform function panel.
5. Select **Paste** from the **Edit** menu to place a copy of the Error control on the Read Waveform panel.
6. Position the Error control in the lower right corner of the panel.

Add help to the Error control as follows.

1. Highlight the Error control and select **Control Help** from the **Edit** menu.

2. Modify the text in the Help Editor dialog box so that it appears as follows.

```
Reports the status of the function call.

Status Codes:
Status                      Description
-------------------------------------------------------------
VI_SUCCESS                  No error (the call was successful).
VI_ERROR_INV_SESSION        The session is invalid.
VI_ERROR_TMO                Timeout expired.
VI_ERROR_RAW_WR_PROT_VIOL   Violation of raw write protocol.
VI_ERROR_BERR               Bus error occurred.
VI_ERROR_NCIC               Not the controller in charge.
VI_ERROR_NLISTENERS         No Listeners.
```

3. Select **Save .FP File** and then select **Close** from the **File** menu.

The Read Waveform function panel should now appear as shown in Figure 8-13.

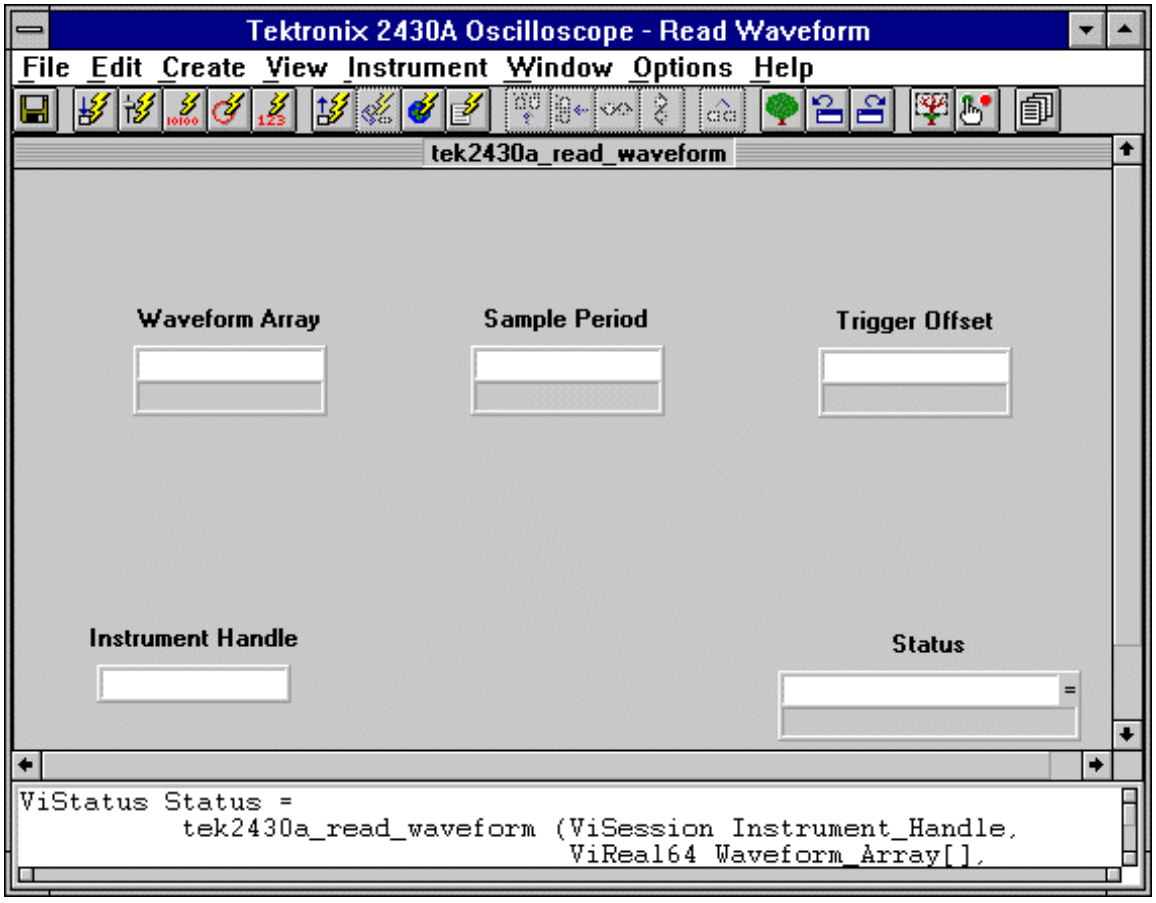


Figure 8-13. The Complete Read Waveform Function Panel Window

The function panels are now complete. Select **Save .FP File** from the **File** menu. Save the function panels again using the filename, `tek2430a.fp`.

Creating the Instrument Program

You are now ready to create the instrument program for the instrument driver. Create the program in three stages.

- Modify the device-dependent items in the file `core_gpb.c`.
- Modify the device-dependent items and add new declarations to the include file `core_gpb.h`.
- Add new functions to the file `core_gpb.c`.

Modifying CORE_GPB.C Source File

As discussed in Chapter 6, *Programming Guidelines for Instrument Drivers*, the file `core_gpb` contains the source code for some core functions for a GPIB instrument driver. Load this file into a source window.

The comments at the beginning of the file list the device-dependent items you must change. Make the following changes to the file.

1. Select **Replace** from the **Edit** menu and perform a global change with the **Case Sensitive** option selected as follows:

   Type `PREFIX` in the Find What box and type `tek2430a` in the Replace With box.

2. The comment `CHANGE` marks the locations of device-dependent items you must change. Select **Find** from the **Edit** menu and search for the word `CHANGE` with the **Case Sensitive** option selected. In Steps 2-6, delete the change comments when you have completed that step.

   The second occurrence of the word `CHANGE` appears at the location where you insert the Instrument Driver information. Insert the following information.

   - The instrument model name
   - The original release date
   - The name or names of the person or people who wrote or modified the instrument driver

3. Find the next occurrence of the word `CHANGE`. You must now specify the default setup string, if any, that is sent to the instrument every time the initialize function is called. Modify the code as follows:

   - Change the string `DEFAULT_STRING` to `PATH OFF`
   - Change the variable `DEFAULT_STRING_LENGTH` to `8`

4. Find the next occurrence of the word CHANGE. The following code performs an ID query. Modify the code as follows:

   Change
   ```
   if ((tek2430a_status = viWrite (*instrSession,
   "*IDN?", 5,&retCnt)) < 0)
   ```

   To
   ```
   if ((tek2430a_status = viWrite (*instrSession,
   "ID?", 3, &retCnt))< 0)
   ```

5. Find the next occurrence of the word CHANGE. You must now modify the parsing of the ID query to reflect the response of the instrument. Read the comments explaining the actions taking place in the ID query. The Tektronix 2430A responds with the string TEK/2430A. Modify the code as follows:

   - Change the string ID_RESPONSE to TEK/2430A
   - Change the variable ID_RESPONSE_LENGTH to 9

6. Find the next occurrence of the word CHANGE. You must now specify the optional reset string, if any, to be sent to the instrument. The command INIT resets the Tektronix 2430A to a known state. Modify the code as follows:

   - Change the string INIT_STRING to INIT
   - Change the variable INIT_STRING_LENGTH to 4

7. Select **Save .FP File As** from the **File** menu and save the modified file in a new file called tek2430a.c.

Modifying the CORE_GPB.H Include File

The instrument program you are creating uses an include file. You must modify the include file CORE_GPB.H as you did the instrument program.

Create the include file for the new driver as follows.

1. Load the include file core_gpb.h.

2. Change the phrase <Instrument Model Name> to Tektronix 2430A Oscilloscope.

3. Select **Change** from the **Edit** menu and perform a global change as follows:

   Type PREFIX in the Find What box and type tek2430a in the Change To box.

4. Add the declarations for the new functions you will write.

   ```
   ViStatus _VI_FUNC tek2430a_config (ViSession instrSession, ViInt32 chan,
                                      ViReal64 v_div, ViReal64 sec_div);
   ```

```
ViStatus _VI_FUNC tek2430a_read_waveform (ViSession instrSession,
                                          ViReal64 _VI_FAR *wvfm,
                                          ViReal64 _VI_FAR *xin,
                                          double *trig_off);
```

5. Select **Save .FP File As** from the **File** menu and save the modified file in an include file named `tek2430a.h`.

Writing the New Functions

You are now ready to add the new functions to the instrument driver. Before doing this, compile the program. To compile the instrument driver, create a project for it by selecting **Project (*.prj)** from the **New** option in the **File** menu. Add the files `TEK2430A.C`, `TEK2430A.H`, and `TEK2430A.FP` to the project by selecting **Add Files To Project** from the **Edit** menu. Then highlight `TEK2430A.C` and select **Compile Project** from the **Build** menu. If any errors occur, compare the source and include files to the listings in the previous section. Correct any mistakes that you find.

Move the cursor to the line after the end of the function `tek2430a_init`.

Writing the Configure Function

This function sets the volts per division of the specified channel and the horizontal timebase of the oscilloscope. The function performs the following operations.

- Check parameters for valid values.
- Format command strings as follows.
  - `v_mode_string` specifies the Vertical Display mode.

    ```
    if chan = 1    the mode is "CH1:ON,CH2:OFF"
    if chan = 2    the mode is "CH1:OFF,CH2:ON"
    ```

  - `CH%d` specifies the channel.
  - `VOL:%f[ep3]` specifies the volts per division.
  - `HOR MOD:ASW,ASE:%f[ep3]` specifies the seconds per division.
  - `ATR MOD:AUTO` sets the trigger mode to AUTO.
  - `VMO DISP:YT,%s` specifies the vertical display.
  - `DAT SOU:CH%d` specifies the source of the acquired data.
- Write the command string to the oscilloscope.

Enter the following code for the `Configure` function.

```
/*=========================================================================*/
/* Function: Configure                                                     */
/* Purpose:  This function configures the vertical sensitivity, timebase,  */
/*           and trigger mode.                                             */
/*=========================================================================*/

ViStatus _VI_FUNC tek2430a_config (ViSession instrSession, ViInt32 chan,
                                   ViReal64 v_div, ViReal64 sec_div)
{

      static ViString v_mode_string[] = {"CH1:ON,CH2:OFF", "CH1:OFF,CH2:ON"};
                ViUInt32 retCnt;
      ViStatus tek2430a_status = VI_SUCCESS;

      if (tek2430a_invalidViInt32Range (chan, 1, 2))
            return VI_ERROR_PARAMETER2;
      if (tek2430a_invalidViReal64Range (v_div, 0.1, 5.0))
            return VI_ERROR_PARAMETER3;
      if (tek2430a_invalidViReal64Range (sec_div, 5.0e-9, 5.0))
            return VI_ERROR_PARAMETER4;
      tek2430a_status = viPrintf (instrSession, "%s<CH%d VOL:%f[p3];HOR
MOD:ASW,ASE:%f[ep3];ATE MOD:AUTO;VMO DISP:YT,%s;DAT SOU:CH%d",
            chan, v_div, sec_div, v_mode_string[chan-1], chan);
      return tek2430a_status;
}
```

Writing the Read Waveform Function

This function performs the following operations.

- Build command strings as follows.
  - `DAT END:RPB` specifies the data format.
  - `PATH OFF` specifies the format of the response to queries.
  - `WFM? XIN` queries for the sample rate (x increment).
  - `WFM? YMU` queries for the y multiplier.
  - `WFM? YOF` queries for the y offset.
  - `WFM? PT.O` queries for the point offset.
- Read the response to the queries, and parse out the x increment, y multiplier, y offset, and point offset.
- Query the waveform for `PATH ON;CURV?`

- Read the header and the 1,024 one-byte integer data points into the array `in.data` or `in_data`. The integers are packed low-byte/high-byte in the array.
- Place the three-byte header in the array header using the Scan function. The function converts the packed one-byte integers to floating-point values and places them in the array `wvfm`.
- Use the Analysis Library function `LinEv1D` to scale the data in `wvfm` using the `y offset` and `y multiplier` values.

Enter the following code for the Read Waveform function after the Configure function.

```
/*=========================================================================*/
/* Function: Read Waveform                                                 */
/* Purpose:  This function reads a waveform and returns x increment, and   */
/*           trigger offset.                                               */
/*=========================================================================*/
ViStatus _VI_FUNC tek2430a_read_waveform (ViSession instrSession,
                                          ViReal64 _VI_FAR *wvfm,
                                          ViReal64 _VI_FAR *xin,
                                          double *trig_off)
{
      ViUInt32 retCnt;
      ViStatus tek2430a_status = VI_SUCCESS;
      ViReal64 ymu, yof;
      ViChar header[2];

      if ((tek2430a_status = viWrite (instrSession,
               "DAT ENCD:RPB;PATH OFF;WVM? XIN;WFM? YMU;WVM? YOF;WFM? PT.0",
               58,&retCnt)) < 0)
            return tek2430a_status;
      if ((tek2430a_status = viScanf (instrSession, "%f;%f;%f;%f",
                                      xin, &ymu, &yof, trig_off)) < 0)
            return tek2430a_status;
      if ((tek2430a_status = viWrite (instrSession, "CURV?", 5, &retCnt)) < 0)
            return tek2430a_status;
      if ((tek2430a_status = viRead (instrSession, in_data, 1027,
                                     &retCnt)) < 0)
            return tek2430a_status;
      yof = -ymu * (yof + 128.0);
      if (Scan (in_data, "%1027i[b1u]>%3i[b1]%1024f", header, wvfm) != 2)
            return VI_ERROR_INV_RESPONSE;

      LinEv1D (wvfm, 1024, ymu, yof, wvfm);

      return tek2430a_status;
}
```

Adding New Include Statements and Variable Declarations

Before compiling the program, you must add several statements at the top of the program to include a file and declare variables. Move to the top of the Program window.

The Read Waveform function scales the waveform data using an Analysis Library routine `LinEv1D`. You must, therefore, add an include statement for the Analysis Library include file. The include statement you should insert appears here in bold type in the following C language code excerpt.

```
#include <visa.h>
#include <formatio.h>
#include <string.h>
#include <analysis.h>
#include "tk2430.h"
```

The Read Waveform function reads binary data into a char array. You must set a dimension for this array. Add the following declaration and comment beneath the declaration of `cmd`.

```
/*    in_data is a buffer for binary data from the scope */
static char tek2430a_in_data[514];
```

Compile the program to verify that you entered the code correctly. Correct any syntax errors you find. Save the completed program in the file `tek2430a`.

Testing the Driver

If you have a Tektronix 2430A oscilloscope, you should now test the instrument driver using the function panels. You should always develop and test the instrument program in source code form, as described in Chapter 6, *Programming Guidelines for Instrument Drivers*.

Appendix A
Tektronix 2430A Instrument Driver Code Sample

This appendix contains instrument driver code samples for the Tektronix 2430A.

Tektronix 2430A Instrument Driver Header File

```
/*== Tektronix 2430A Oscilloscope Include File ============================*/

#ifndef __PREFIX_HEADER
#define __PREFIX_HEADER

#include <vpptype.h>

/*= GLOBAL FUNCTION DECLARATIONS ==========================================*/

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

ViStatus _VI_FUNC tek2430a_init (ViRsrc resourceName, ViBoolean IDQuery,
                                 ViBoolean reset, ViPSession instrSession);
ViStatus _VI_FUNC tek2430a_close (ViSession instrSession);
ViStatus _VI_FUNC tek2430a_reset (ViSession instrSession);
ViStatus _VI_FUNC tek2430a_config (ViSession instrSession, ViInt32 chan,
                                   ViReal64 v_div, ViReal64 sec_div);
ViStatus _VI_FUNC tek2430a_read_waveform (ViSession instrSession,
                                          ViReal64 _VI_FAR *wvfm,
                                          ViReal64 _VI_FAR *xin,
                                          ViReal64 _VI_FAR *trig_off);
ViStatus _VI_FUNC PREFIX_self_test (ViSession instrSession,
                                    ViPInt16 testResult,
                                    ViChar _VI_FAR testMessage[]);
ViStatus _VI_FUNC PREFIX_error_query (ViSession instrSession,
                                      ViPInt32 errCode,
                                      ViChar _VI_FAR errMessage[]);
ViStatus _VI_FUNC PREFIX_error_message (ViSession instrSession,
                                        ViStatus error,
                                        ViChar _VI_FAR message[]);
ViStatus _VI_FUNC PREFIX_revision_query (ViSession instrSession,
                                         ViChar _VI_FAR driverRev[],
                                         ViChar _VI_FAR instrRev[]);

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

#endif
/*=== END INCLUDE FILE ====================================================*/
```

Tektronix 2430A Instrument Driver Source File

```
#include <visa.h>
#include <formatio.h>
#include <string.h>
#include <analysis.h>
#include "tk2430.h"

#define tek2430a_REVISION     "A1.0"     /*  Instrument driver revision    */

/*= Instrument Model Name ==============================================*/
/* LabWindows Instrument Driver                                        */
/* Original Release:  November, 1993                                   */
/* By:  Bill Pitts                                                     */
/* Modification History: None                                          */
/*=====================================================================*/

/*= INTERNAL DATA ======================================================*/
static char in_data[1027];

/*= UTILITY ROUTINES ===================================================*/
ViBoolean tek2430a_invalidViBooleanRange (ViBoolean val);
ViBoolean tek2430a_invalidViInt32Range  (ViInt32 val, ViInt32 min,
                                         ViInt32max);
ViBoolean tek2430a_invalidViReal64Range (ViReal64 val, ViReal64 min, ViReal64
                                         max);
ViStatus  tek2430a_initCleanUp (ViSession openRMSession,
                                ViSession *openInstrSession, ViStatus
                                currentStatus);

/*=====================================================================*/
/* Function: Initialize                                                */
/* Purpose:  This function opens the instrument, queries the instrument    */
/*           for its ID, and initializes the instrument to a known state.  */
/*=====================================================================*/
ViStatus _VI_FUNC tek2430a_init (ViRsrc resourceName, ViBoolean IDQuery,
                                 ViBoolean reset, ViPSession instrSession)
{
      ViStatus tek2430a_status = VI_SUCCESS;
      ViSession rmSession = 0;
      ViUInt32 retCnt = 0;

      /*- Check input parameter ranges ---------------------------------*/

      if (tek2430a_invalidViBooleanRange (IDQuery))
          return VI_ERROR_PARAMETER2;
      if (tek2430a_invalidViBooleanRange (reset))
            return VI_ERROR_PARAMETER3;
```

```
/*- Open instrument session ---------------------------------------*/

if ((tek2430a_status = viOpenDefaultRM (&rmSession)) < 0)
      return tek2430a_status;

if ((tek2430a_status = viOpen (rmSession, resourceName, VI_NULL,
                               VI_NULL, instrSession)) < 0) {
      viClose (rmSession);
      return tek2430a_status;
}

/*- Configure VISA Formatted I/O ----------------------------------*/

if ((tek2430a_status = viSetAttribute (*instrSession,
                                       VI_ATTR_TMO_VALUE, 10000)) < 0)
      return tek2430a_initCleanUp (rmSession, instrSession,
                                   tek2430a_status);
if ((tek2430a_status = viSetBuf (*instrSession,
                                 VI_READ_BUF|VI_WRITE_BUF, 4000)) < 0)
      return tek2430a_initCleanUp (rmSession, instrSession,
                                   tek2430a_status);
if ((tek2430a_status = viSetAttribute (*instrSession,
                                       VI_ATTR_WR_BUF_OPER_MODE,
                                       VI_FLUSH_ON_ACCESS)) < 0)
      return tek2430a_initCleanUp (rmSession, instrSession,
                                   tek2430a_status);
if ((tek2430a_status = viSetAttribute (*instrSession,
                                       VI_ATTR_RD_BUF_OPER_MODE,
                                       VI_FLUSH_ON_ACCESS)) < 0)
      return tek2430a_initCleanUp (rmSession, instrSession,
                                   tek2430a_status);

/*- Identification Query ------------------------------------------*/

if (IDQuery) {
      if ((tek2430a_status = viWrite (*instrSession, "ID?", 3,
                                      &retCnt)) < 0)
            return tek2430a_initCleanUp (rmSession, instrSession,
                                         tek2430a_status);

      if ((tek2430a_status = viScanf (*instrSession,
                                      "TEK/2430A%*[\n]")) < 0)
            return tek2430a_initCleanUp (rmSession, instrSession,
                                         VI_ERROR_FAIL_ID_QUERY);
}

/*- Reset instrument ----------------------------------------------*/

if (reset)
      if ((tek2430a_status = tek2430a_reset (*instrSession)) < 0)
            return tek2430a_initCleanUp (rmSession, instrSession,
                                         tek2430a_status);
```

```
      /*- Send Default Instrument Setup ---------------------------------*/

      if ((tek2430a_status = viWrite (*instrSession, "PATH OFF", 8,
                                      &retCnt)) < 0)
            return tek2430a_initCleanUp (rmSession, instrSession,
                                         tek2430a_status);

      return tek2430a_status;
}

/*=========================================================================*/
/* Function: Reset                                                         */
/* Purpose:  This function resets the instrument.  If the reset function   */
/*           is not supported by the instrument, this function returns     */
/*           the warning VI_WARN_NSUP_RESET.                               */
/*=========================================================================*/
ViStatus _VI_FUNC tek2430a_reset (ViSession instrSession)
{
      ViUInt32 retCnt;
      ViStatus tek2430a_status = VI_SUCCESS;

      /*  Initialize the instrument to a known state.  */

      if ((tek2430a_status = viWrite (instrSession, "INIT", 4, &retCnt)) < 0)
            return tek2430a_status;

      return tek2430a_status;
}

/*=========================================================================*/
/* Function: Configure                                                     */
/* Purpose:  This function configures the vertical sensitivity, timebase,  */
/*           and trigger mode.                                             */
/*=========================================================================*/
ViStatus _VI_FUNC tek2430a_config (ViSession instrSession, ViInt32 chan,
                                   ViReal64 v_div, ViReal64 sec_div)
{
      static ViString v_mode_string[] = {"CH1:ON,CH2:OFF", "CH1:OFF,CH2:ON"};
      ViUInt32 retCnt;
      ViStatus tek2430a_status = VI_SUCCESS;

      if (tek2430a_invalidViInt32Range (chan, 1, 2))
            return VI_ERROR_PARAMETER2;
      if (tek2430a_invalidViReal64Range (v_div, 0.1, 5.0))
            return VI_ERROR_PARAMETER3;
      if (tek2430a_invalidViReal64Range (sec_div, 5.0e-9, 5.0))
            return VI_ERROR_PARAMETER4;

      tek2430a_status = viPrintf (instrSession, "%s<CH%d VOL:%f[p3];HOR
MOD:ASW,ASE:%f[ep3];ATE MOD:AUTO;VMO DISP:YT,%s;DAT SOU:CH%d",
            chan, v_div, sec_div, v_mode_string[chan-1], chan);

      return tek2430a_status;
}
```

```
/*=========================================================================*/
/* Function: Read Waveform                                                 */
/* Purpose:  This function reads a waveform and returns x increment, and   */
/*           trigger offset.                                               */
/*=========================================================================*/
ViStatus _VI_FUNC tek2430a_read_waveform (ViSession instrSession, ViReal64
_VI_FAR *wvfm, ViReal64 _VI_FAR *xin, ViReal64 _VI_FAR *trig_off)

{
      ViUInt32 retCnt;
      ViStatus tek2430a_status = VI_SUCCESS;
      ViReal64 ymu, yof;
      ViChar header[2];

      if ((tek2430a_status = viWrite (instrSession,
              "DAT ENCD:RPB;PATH OFF;WVM? XIN;WFM? YMU;WVM? YOF;WFM? PT.0",
              58, &retCnt)) < 0)
            return tek2430a_status;

      if ((tek2430a_status = viScanf (instrSession, "%f;%f;%f;%f",
                                      xin, &ymu, &yof, trig_off)) < 0)
            return tek2430a_status;

      if ((tek2430a_status = viWrite (instrSession, "CURV?", 5,
                                      &retCnt)) < 0)
            return tek2430a_status;

      if ((tek2430a_status = viRead (instrSession, in_data, 1027,
                                     &retCnt)) < 0)
            return tek2430a_status;

      yof = -ymu * (yof + 128.0);
      if (Scan (in_data, "%1027i[b1u]>%3i[b1]%1024f", header, wvfm) != 2)
            return VI_ERROR_INV_RESPONSE;
      LinEv1D (wvfm, 1024, ymu, yof, wvfm);
      return tek2430a_status;
}
/*=========================================================================*/
/* Function: Self-Test                                                     */
/* Purpose:  This function executes the instrument self-test and returns   */
/* the result.                                                             */
/*=========================================================================*/
ViStatus _VI_FUNC tek2430a_self_test (ViSession instrSession, ViPInt16
testResult, ViChar _VI_FAR testMessage[])
{
      ViUInt32 retCnt;
      ViStatus tek2430a_status = VI_SUCCESS;

      if ((tek2430a_status = viWrite (instrSession, "TESTT SELFD;EXE;ERR?",
                                      20, &retCnt)) < 0)
        return tek2430a_status;
```

```
    if ((tek2430a_status = viScanf (instrSession, "%d%[^\"]",
                                    testResult, testMessage)) < 0)
        return tek2430a_status;

    return tek2430a_status;
}

/*============================================================================*/
/* Function: Error Query                                                      */
/* Purpose:  This function queries the instrument error queue.                */
/*============================================================================*/
ViStatus _VI_FUNC tek2430a_error_query (ViSession instrSession, ViPInt32
errCode, ViChar _VI_FAR errMessage[])
{
    return VI_WARN_NSUP_ERROR_QUERY;
}

/*============================================================================*/
/* Function: Error Message                                                    */
/* Purpose:  This function Translates the error return value from the         */
/*           instrument driver into a user-readable string.                   */
/*============================================================================*/
ViStatus _VI_FUNC tek2430a_error_message (ViSession instrSession, ViStatus
errorCode, ViChar _VI_FAR errMessage[])
{
      ViStatus tek2430a_status = VI_SUCCESS;

      tek2430a_status = viStatusDesc (instrSession, errorCode, errMessage);
      if (tek2430a_status = VI_WARN_UNKNOWN_STATUS) {
            switch (errorCode) {
                  case VI_WARN_NSUP_ERROR_QUERY:
                        errMessage = "WARNING: Error Query not supported";
                        tek2430a_status = VI_SUCCESS;
                        break;
                  case VI_ERROR_PARAMETER2:
                        errMessage = "ERROR: Parameter 2 out of range";
                        tek2430a_status = VI_SUCCESS;
                        break;
                  case VI_ERROR_PARAMETER3:
                        errMessage = "ERROR: Parameter 3 out of range";
                        tek2430a_status = VI_SUCCESS;
                        break;
                  case VI_ERROR_FAIL_ID_QUERY:
                        errMessage = "ERROR: Identification query failed";
                        tek2430a_status = VI_SUCCESS;
                        break;
                  case VI_ERROR_INV_RESPONSE:
                        errMessage = "ERROR: Interpreting instrument response";
                        tek2430a_status = VI_SUCCESS;
                        break;
```

```
                  default:
                        errMessage = "Unknown Error";
                        tek2430a_status = VI_WARN_UNKNOWN_STATUS;
                        break;
            }
      }
    return tek2430a_status;
}

/*=============================================================================*/
/* Function: Revision                                                          */
/* Purpose:  This function returns the driver and instrument revisions.        */
/*=============================================================================*/
ViStatus _VI_FUNC tek2430a_revision_query (ViSession instrSession, ViChar
                                           _VI_FAR driverRev[], ViChar _VI_FAR
                                           instrRev[])
{
      ViUInt32 retCnt = 0;
      ViUInt16 i = 0;
      ViStatus tek2430a_status = VI_SUCCESS;

      strcpy (driverRev, tek2430a_REVISION);

      if ((tek2430a_status = viWrite (instrSession, "ID?", 3, &retCnt)) < 0)
        return tek2430a_status;

      if ((tek2430a_status = viScanf (instrSession, "%*[^,],%[^\n]",
                                      instrRev)) < 0)
        return tek2430a_status;

      return tek2430a_status;
}

/*=============================================================================*/
/* Function: Close                                                             */
/* Purpose:  This function closes the instrument.                              */
/*=============================================================================*/
ViStatus _VI_FUNC tek2430a_close (ViSession instrSession)
{
      ViSession rmSession;
      ViStatus tek2430a_status = VI_SUCCESS;

      if ((tek2430a_status = viGetAttribute (instrSession,
                                             VI_ATTR_RM_SESSION,
                                             &rmSession)) < 0)
            return tek2430a_status;

      tek2430a_status = viClose (instrSession);
      viClose (rmSession);

      return tek2430a_status;
}
```

```
/*= UTILITY ROUTINES =======================================================*/

/*==========================================================================*/
/* Function: Invalid Boolean Range                                          */
/* Purpose:  This function checks a boolian to see if it lies between a     */
/*           minimum and maximum value.  If the value is out of range, set  */
/*           the return value to VI_FALSE.  If the value is OK, set the     */
/*           return value to VI_TRUE.                                       */
/*==========================================================================*/
ViBoolean tek2430a_invalidViBooleanRange (ViBoolean val)
{
    return (val < VI_FALSE || val > VI_TRUE);
}

/*==========================================================================*/
/* Function: Invalid Long Integer Range                                     */
/* Purpose:  This function checks a long integer to see if it lies between  */
/*           a minimum and maximum value.  If the value is out of range,    */
/*           set the global error variable to the value err_code.  If the   */
/*           value is OK, error = 0.  The return value is equal to the      */
/*           global error value.                                            */
/*==========================================================================*/
ViBoolean tek2430a_invalidViInt32Range (ViInt32 val, ViInt32 min,
                                        ViInt32 max)
{
    return (val < min || val > max);
}

/*==========================================================================*/
/* Function: Invalid Real Range                                             */
/* Purpose:  This function checks a real number to see if it lies between   */
/*           a minimum and maximum value.  If the value is out of range,    */
/*           set the global error variable to the value err_code.  If the   */
/*           value is OK, error = 0.  The return value is equal to the      */
/*           global error value.                                            */
/*==========================================================================*/
ViBoolean tek2430a_invalidViReal64Range (ViReal64 val, ViReal64 min,
                                         ViReal64 max)
{
    return (val < min || val > max);
}

/*==========================================================================*/
/* Function: Initialize Clean Up                                            */
/* Purpose:  This function is used only by the tek2430a_init function.      */
/*           When an error is detected this function is called to close     */
/*           the open resource manager and instrument object sessions and   */
/*           to set the instrSession that is returned from tek2430a_init    */
/*           to VI_NULL.                                                    */
/*==========================================================================*/
```

```
ViStatus tek2430a_initCleanUp (ViSession openRMSession, ViSession
                               *openInstrSession, ViStatus currentStatus)
{
      viClose (*openInstrSession);
      viClose (openRMSession);
      *openInstrSession = VI_NULL;
      return currentStatus;
}

/*=== THE END ================================================================*/
```

Appendix B
Customer Communication

For your convenience, this appendix contains forms to help you gather the information necessary to help us solve technical problems you might have as well as a form you can use to comment on the product documentation. Filling out a copy of the *Technical Support Form* before contacting National Instruments helps us help you better and faster.

National Instruments provides comprehensive technical assistance around the world. In the U.S. and Canada, applications engineers are available Monday through Friday from 8:00 a.m. to 6:00 p.m. (central time). In other countries, contact the nearest branch office. You may fax questions to us at any time.

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 or (800) 327-3077
  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: 1 48 65 15 59
  Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

FaxBack Support

FaxBack is a 24-hour information retrieval system containing a library of documents on a wide range of technical information. You can access FaxBack from a touch-tone telephone at the following number: (512) 418-1111.

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.

E-Mail Support (currently U.S. only)

You can submit technical support questions to the appropriate applications engineering team through e-mail at the Internet addresses listed below. Remember to include your name, address, and phone number so we can contact you with solutions and suggestions.

GPIB:	`gpib.support@natinst.com`
DAQ:	`daq.support@natinst.com`
VXI:	`vxi.support@natinst.com`
LabVIEW:	`lv.support@natinst.com`
LabWindows:	`lw.support@natinst.com`
HiQ:	`hiq.support@natinst.com`
Lookout:	`lookout.support@natinst.com`
VISA:	`visa.support@natinst.com`

Fax and Telephone 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.

	Telephone	Fax
Australia	03 9 879 9422	03 9 879 9179
Austria	0662 45 79 90 0	0662 45 79 90 19
Belgium	02 757 00 20	02 757 03 11
Canada (Ontario)	519 622 9310	
Canada (Quebec)	514 694 8521	514 694 4399
Denmark	45 76 26 00	45 76 26 02
Finland	90 527 2321	90 502 2930
France	1 48 14 24 24	1 48 14 24 14
Germany	089 741 31 30	089 714 60 35
Hong Kong	2645 3186	2686 8505
Italy	02 413091	02 41309215
Japan	03 5472 2970	03 5472 2977
Korea	02 596 7456	02 596 7455
Mexico	95 800 010 0793	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
U.K.	01635 523545	01635 523154

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: Windows 3.1, Windows for Workgroups 3.11, Windows NT 3.1, Windows NT 3.5, Windows 95, other (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 will reproduce the problem _____

_____

_____

_____

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. When you complete this form accurately before contacting National Instruments for technical support, our applications engineers can answer your questions more efficiently.

National Instruments Products

Data Acquisition Hardware Revision _____

Interrupt Level of Hardware _____

DMA Channels of Hardware _____

Base I/O Address of Hardware _____

NI-DAQ, LabVIEW, or
LabWindows Version _____

Other Products

Computer Make and Model _____

Microprocessor _____

Clock Frequency _____

Type of Video Board Installed _____

Operating System _____

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: **July 1996**

Part Number: **320684C-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 _____

_____

Fax (_____)_____ Phone (_____)_____

Mail to: Technical Publications
National Instruments Corporation
6504 Bridge Point Parkway
Austin, TX 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

Hz	hertz
hex	hexadecimal

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

return value control	A function panel control that displays a value returned from the primary function.
return value error reporting method	The method used to declare each instrument driver routine as an integer function and return the appropriate value.
ring control	A control that displays a list of options one option at a time.

S

s	seconds
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.
s/pt	seconds per point
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

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W