Getting Results with ComponentWorks™

NATIONAL INSTRUMENTS™
The Software is the Instrument™

April 1998 Edition
Part Number 321170C-01

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

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

Fax-on-Demand Support
512 418 1111

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

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

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

© Copyright 1996, 1998 National Instruments Corporation. All rights reserved.

Important Information

Warranty

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

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

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

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

Copyright

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

Trademarks

ComponentWorks™, DAQ-STC™, DataSocket™, LabVIEW™, LabWindows™/CVI, NI-488.2™, NI-DAQ™, NI-VISA™, NI-VXI™, and SCXI™ are trademarks of National Instruments Corporation.

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

WARNING REGARDING MEDICAL AND CLINICAL USE OF NATIONAL INSTRUMENTS PRODUCTS

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

Contents

About This Manual

Organization of This Manual xix
Conventions Used in This Manual xxii
Related Documentation xxiii
Customer Communication xxiii

Chapter 1
Introduction to ComponentWorks

What is ComponentWorks? 1-1
Installing ComponentWorks 1-3
System Requirements 1-3
Installation Instructions 1-3
Installing the ComponentWorks ActiveX Controls 1-4
Installing From Floppy Disks 1-4
Installing the Instrument Driver Factory 1-5
Installing the Instrument Drivers DLLs 1-5
Installed Files 1-6
About the ComponentWorks Controls 1-7
Properties, Methods, and Events 1-7
Object Hierarchy 1-8
Collection Objects 1-9
Setting the Properties of an ActiveX Control 1-10
Using Property Pages 1-10
Changing Properties Programmatically 1-12
Item Method 1-13
Working with Control Methods 1-13
Developing Event Handler Routines 1-14
Using the Analysis Library and Instrument Driver DLLs 1-15
The Online Reference—Learning the Properties, Methods, and Events 1-15

Chapter 2
Getting Started with ComponentWorks

Installing and Configuring Driver Software 2-1
Exploring the ComponentWorks Documentation 2-2
Getting Results with ComponentWorks Manual 2-2
ComponentWorks Online Reference 2-3
Accessing the Online Reference 2-4
Finding Specific Information 2-4

Becoming Familiar with the Examples Structure 2-4
Developing Your Application 2-5
Seeking Information from Additional Sources 2-7

Chapter 3
Building ComponentWorks Applications with Visual Basic

Developing Visual Basic Applications 3-1
Loading the ComponentWorks Controls into the Toolbox 3-2
Building the User Interface Using ComponentWorks 3-3
Using Property Sheets 3-3
Using Your Program to Edit Properties 3-4
Working with Control Methods 3-5
Developing Control Event Routines 3-6
Using the ComponentWorks Instrument Driver DLLs in Visual Basic 3-7
Using the Object Browser to Build Code in Visual Basic 3-8
Pasting Code into Your Program 3-10
Adding Code Using Visual Basic Code Completion 3-11
Learning to Use Specific ComponentWorks Controls 3-12

Chapter 4
Building ComponentWorks Applications with Visual C++

Developing Visual C++ Applications 4-1
Creating Your Application 4-2
Adding ComponentWorks Controls to the Visual C++ Controls Toolbar 4-4
Building the User Interface Using ComponentWorks Controls 4-4
Programming with the ComponentWorks Controls 4-5
Using Properties 4-6
Using Methods 4-8
Using Events 4-9
Learning to Use Specific ComponentWorks Controls 4-10

Chapter 5
Building ComponentWorks Applications with Delphi

Running Delphi Examples 5-1
Upgrading from a Previous Version of ComponentWorks 5-2
Developing Delphi Applications 5-2
Loading the ComponentWorks Controls into the Component Palette 5-2
Building the User Interface 5-4
Placing Controls 5-4
Using Property Sheets 5-5

Programming with ComponentWorks 5-6
Using Your Program to Edit Properties 5-6
Using Methods 5-7
Using Events 5-8
Learning to Use Specific ComponentWorks Controls 5-9

Chapter 6
Using the Graphical User Interface Controls

What Are the UI Controls? 6-2
Object Hierarchy and Common Objects 6-2
The Knob and Slide Controls 6-3
Knob and Slide Object 6-3
Pointers Collection 6-4
Pointer Object 6-4
Axis Object 6-4
Ticks Object 6-5
ValuePairs Collection 6-5
ValuePair Object 6-6
Statistics Object 6-6
Events 6-6
The Numeric Edit Box Control 6-7
Events 6-7
Tutorial: Knob, Slide, and Numeric Edit Box Controls 6-8
Designing the Form 6-9
Developing the Program Code 6-10
Testing Your Program 6-11
The Button Control 6-12
Events 6-13
The Graph Control 6-13
Graph Object 6-14
Plot Methods 6-15
Chart Methods 6-16
Plots Collection 6-16
Plot Object 6-17
PlotTemplate Object 6-18
Cursors Collection 6-18
Cursor Object 6-19
Axes Collection 6-20
Axis Object 6-20
Events 6-21
Panning and Zooming 6-21

Tutorial: Graph and Button Controls 6-22
Designing the Form 6-22
Developing the Code 6-24
Testing Your Program 6-25

Chapter 7
Using the Data Acquisition Controls

What Are the Data Acquisition Controls? 7-1
Data Acquisition Configuration 7-2
Object Hierarchy and Common Properties 7-3
Device, DeviceName, and DeviceType 7-3
Channel Strings 7-3
SCXI Channel Strings 7-4
ExceptionOnError and ErrorEventMask 7-5
AIPoint Control—Single Point Analog Input 7-6
AIPoint Object 7-6
Channels Collection 7-7
Channel Object 7-8
ChannelClock Object 7-8
AI Control—Waveform Analog Input 7-9
AI Object 7-10
Methods and Events 7-10
Asynchronous Acquisition 7-10
Synchronous Acquisition 7-11
Error Handling 7-12
ScanClock and ChannelClock Objects 7-12
StartCondition, PauseCondition and StopCondition Objects 7-13
Tutorial: Using the AIPoint and AI DAQ Controls 7-14
Designing the Form 7-15
Setting the DAQ Properties 7-16
Developing the Code 7-16
Testing Your Program 7-18
AOPoint Control—Single Point Analog Output 7-18
AOPoint Object 7-19
Methods 7-19
AO Control—Waveform Analog Output 7-20
AO Object 7-21
Methods and Events 7-21
UpdateClock and IntervalClock Objects 7-22
StartCondition Object 7-23

Tutorial: Using the AOPoint Control 7-24
Designing the Form 7-24
Developing the Code 7-25
Testing Your Program 7-27
Digital Controls and Hardware 7-28
DIO Control—Single Point Digital Input and Output 7-29
DIO Object 7-30
Ports Collection and Port Object 7-31
Lines Collection and Line Object 7-32
Common Properties and Methods 7-32
DI Control—Buffered Waveform Digital Input 7-34
DI Object 7-35
UpdateClock Object 7-36
Methods and Events 7-36
DO Control—Buffered Waveform Digital Output 7-37
DO Object 7-38
UpdateClock Object 7-39
Methods and Events 7-40
Tutorial: Using the DIO Control 7-41
Designing the Form 7-42
Developing the Code 7-42
Testing Your Program 7-44
Counter/Timer Hardware 7-45
Counter Control—Counting and Measurement Operations 7-46
Counter Object 7-46
Methods and Events 7-48
Buffered Measurements 7-50
Pulse Control—Digital Pulse and Pulsetrain Generation 7-51
Pulse Object 7-51
Methods 7-53
FSK and ETS Pulse Generation 7-54
Tutorial: Using the Counter and Pulse Controls 7-55
Designing the Form 7-55
Developing the Code 7-56
Testing Your Program 7-59
DAQTools—Data Acquisition Utility Functions 7-60
Using DAQ Tools Functions 7-61

Chapter 8
Using the GPIB and Serial Controls

What Are the GPIB and Serial Controls? 8-1
Object Hierarchy and Common Features 8-2
Common Properties 8-2
Parsing 8-3
Advanced Parsing Features 8-3
CWTask Object 8-4
CWPattern Object 8-5
CWToken Object 8-6
The GPIB Control 8-6
CWGPIB Object 8-7
Methods and Events 8-8
Synchronous I/O 8-8
Asynchronous I/O 8-8
Other GPIB Operations 8-9
Tutorial: Using the GPIB Control 8-9
Designing the Form 8-10
Setting the GPIB Control Properties 8-10
Developing the Code 8-11
Testing Your Program 8-11
The Serial Control 8-12
CWSerial Object 8-12
Methods and Events 8-13
Synchronous I/O 8-13
Asynchronous I/O 8-14
Tutorial: Using the Serial Control 8-14
Designing the Form 8-14
Setting the Serial Control Properties 8-15
Developing the Code 8-17
Testing Your Program 8-17

Chapter 9
Using the VISA Control

Overview of the VISA API 9-1
VISA Structure 9-1
VISA Advantages 9-2
What is the VISA Control? 9-2
Object Hierarchy and Common Properties 9-2
Common Instrument Control Features 9-4
Parsing 9-5
VISA Object 9-5

RdWrt Object....9-7
Serial (ASRL) Object....9-8
GPIB Object....9-9
VXI Object....9-9
Methods and Events....9-9
Message-Based Communication....9-9
Synchronous I/O....9-10
Asynchronous I/O....9-10
Register-Based Communication....9-11
Events....9-14
Event Types....9-14
Event Handling With The Event Queue....9-15
Checking Events in the Queue....9-16
Discarding Events From The Queue....9-16
Disabling The Event Queue....9-16
Tutorial: Using the VISA Control for Message-Based Communication....9-17
Designing the Form....9-17
Setting the VISA Control Properties....9-18
Developing the Code....9-19
Testing Your Program....9-19
Tutorial: Using the VISA Control for Register-Based Communication....9-20
Designing the Form....9-20
Setting the VISA Control Properties....9-21
Developing the Code....9-21
Testing Your Program....9-22

Chapter 10
Using the Analysis Controls and Functions

What Are the Analysis Controls?....10-1
Analysis Library Versions....10-2
Controls....10-12
Analysis Function Descriptions....10-13
Error Messages....10-13
Tutorial: Using Simple Statistics Functions....10-14
Designing the Form....10-15
Developing the Program Code....10-16
Testing Your Program....10-17

Chapter 11
Using the DataSocket Control and Tools

What is DataSocket? 11-1
DataSocket Basics 11-2
Locating a Data Source 11-3
Reading Data from a Data Source 11-3
OnDataUpdated Event 11-4
Updating the Data 11-5
Automatically Updating Data 11-5
OnStatusUpdated Event 11-5
Disconnecting from a Data Source 11-5
Tutorial: Reading a Waveform 11-6
Designing the Form 11-6
Developing the Program Code 11-7
Testing Your Program 11-8
Writing Data to a Data Target 11-9
Updating a Data Target 11-10
Automatically Updating a Target 11-10
Working with CWData 11-11
Working with Attributes 11-12
Standalone CWData Objects 11-12
Setting Up a DataSocket Server 11-13
Requirements for Running the DataSocket Server 11-14
Checking the Status of the DataSocket Server 11-14
Creating Data Items on the Server 11-14
Connecting to Data Items and Reading Them 11-15
Tutorial: Sharing Data between Applications 11-15
Configuring the DataSocket Server 11-16

Chapter 12
Building Advanced Applications

Using Advanced ComponentWorks Features 12-1
A Virtual Oscilloscope 12-1
Data Acquisition Stop Condition Modes 12-2
Data Acquisition Pretriggering 12-3
User Interface Value Pairs 12-3
Virtual Spectrum Meter 12-4
DSP Analysis Library 12-5
Cursors 12-7
Graph Track Mode 12-8

A Virtual Data Logger 12-9
Multiple Graph Axes 12-10
Graph Axes Formats 12-11
File Input/Output 12-12
Adding Testing and Debugging to Your Application 12-12
Error Checking 12-12
Exceptions 12-13
Return Codes 12-14
Error and Warning Events 12-15
GetErrorText Function 12-16
Debugging 12-17
Debug Print 12-17
Breakpoint 12-17
Watch Window 12-18
Single Step, Step Into, and Step Over 12-18

Appendix A
Using Previous Versions of Visual Basic, Visual C++, and Delphi with ComponentWorks

Visual Basic 4 ..... A-1
Menus and Commands A-1
Object Browser A-2
Code Completion A-3
Creating a Default ComponentWorks Project A-3
Visual C++ 4.x A-4
Creating Your Application A-4
Adding ComponentWorks Controls to the Visual C++ Controls Toolbar A-4
Building Your User Interface and Code A-6
Delphi 2 ..... A-6
Loading the ComponentWorks Controls into the Component Palette A-6

Appendix B
Background Information about Data Acquisition

Installation B-1
Configuration B-2
SCXI B-2
Device Number B-2
Channel Wizard B-3
Programming B-3
Device Number and Channels B-3
Buffers B-4

Clocks B-5
SCXI B-6
Hardware B-7
SCXI B-7
RTSI B-8
PFI B-9

Appendix C
Common Questions

Installation and Getting Started C-1
Visual Basic C-3
User Interface Controls C-4
Data Acquisition Controls C-8
GPIB, Serial, and VISA Controls C-12
Analysis Controls C-14

Appendix D
Error Codes

Appendix E
Distribution and Redistributable Files

Files E-1
Distribution E-1
Automatic Installers E-2
Manual Installation E-2
DataSocket Server E-3
Instrument Drivers E-3
ComponentWorks Evaluation E-4
Run-Time Licenses E-4
Troubleshooting E-4

Appendix F
Customer Communication

Glossary

Index

Figures

Figure 1-1. Slide Control Object Hierarchy 1-9
Figure 1-2. Visual Basic Default Property Sheets 1-11
Figure 1-3. ComponentWorks Custom Property Pages 1-11

Figure 3-1. Visual Basic Property Pages 3-4
Figure 3-2. ComponentWorks Custom Property Pages 3-4
Figure 3-3. Selecting Events in the Code Window 3-7
Figure 3-4. Viewing CWGraph in the Object Browser 3-9
Figure 3-5. Viewing CWKnob in the Object Browser 3-10
Figure 3-6. Visual Basic 5 Code Completion 3-11

Figure 4-1. New Dialog Box 4-2
Figure 4-2. MFC AppWizard—Step 1 4-3
Figure 4-3. CWGraph Control Property Sheets 4-5
Figure 4-4. MFC ClassWizard—Member Variable Tab 4-6
Figure 4-5. Viewing Property Functions and Methods in the Workspace Window 4-7
Figure 4-6. Event Handler for the PointerValueChanged Event of a Knob 4-10

Figure 5-1. Delphi Import ActiveX Control Dialog Box 5-3
Figure 5-2. ComponentWorks Controls on a Delphi Form 5-5
Figure 5-3. Delphi Object Inspector 5-5
Figure 5-4. ComponentWorks Graph Control Property Page 5-6
Figure 5-5. Delphi Object Inspector Events Tab 5-8

Figure 6-1. Knob/Slide Control Object Hierarchy 6-3
Figure 6-2. SimpleUI Form 6-9
Figure 6-3. Testing SimpleUI 6-11
Figure 6-4. Button Control Modes 6-12
Figure 6-5. Graph Control Object Hierarchy 6-14
Figure 6-6. ButtonGraphExample Form 6-23

Figure 7-1. AIPoint Control Object Hierarchy (Single Point Analog Input) 7-6
Figure 7-2. AI Control Object Hierarchy (Waveform Analog Input) 7-9
Figure 7-3. AIExample Form 7-15
Figure 7-4. Testing AIExample 7-18
Figure 7-5. AOPoint Control Object Hierarchy (Single Point Analog Output) 7-19
Figure 7-6. AOPoint Control Object Hierarchy (Waveform Analog Output) 7-20
Figure 7-7. AOPoint Form 7-25
Figure 7-8. Testing AOPoint 7-28
Figure 7-9. DIO Control Object Hierarchy 7-30
Figure 7-10. DI Control Object Hierarchy 7-34

Figure 7-11. DO Control Object Hierarchy 7-38
Figure 7-12. DIO Form 7-42
Figure 7-13. Counter Control Object Hierarchy 7-46
Figure 7-14. Pulse Control Object Hierarchy 7-51
Figure 7-15. Counters Form 7-56
Figure 7-16. Testing Counters 7-59

Figure 8-1. GPIB Control Object Hierarchy 8-7
Figure 8-2. GPIBExample Form 8-10
Figure 8-3. Serial Control Object Hierarchy 8-12
Figure 8-4. Weigh Form 8-15
Figure 8-5. Serial Property Pages—Parsing Page 8-16

Figure 9-1. VISA Structure 9-1
Figure 9-2. VISA Control Object Hierarchy 9-3
Figure 9-3. VISA Property Pages—General Page 9-6
Figure 9-4. VISA Property Pages—RdWrt Page 9-7
Figure 9-5. VISA Property Pages—Serial Page 9-8
Figure 9-6. VISA Property Pages—VxiMemory Page 9-11
Figure 9-7. MbasedExample Form 9-18
Figure 9-8. RbasedExample Form 9-21

Figure 10-1. Stat Form 10-15
Figure 10-2. Testing Stat 10-17

Figure 11-1. DataSocket Connection 11-2
Figure 11-2. Specifying Data Source Locations 11-3
Figure 11-3. SimpleDS Form 11-7
Figure 11-4. DataSocket Control 11-11
Figure 11-5. DataSocket Server Tray Icon 11-14
Figure 11-6. DataSocket Server Status Window 11-14
Figure 11-7. DataSocket Server Manager 11-17

Figure 12-1. Virtual Oscilloscope 12-2
Figure 12-2. Knob Property Pages—Value Pairs Page 12-4
Figure 12-3. Virtual Spectrum Meter 12-5
Figure 12-4. Graph Property Pages—Cursors Property Page 12-7
Figure 12-5. Virtual Data Logger 12-9
Figure 12-6. Visual Basic Error Messages 12-13
Figure 12-7. Error Message Box 12-16
Figure 12-8. Error Handling Message Box 12-17

Figure A-1. Visual Basic 4 Object Browser A-2
Figure A-2. Visual Basic 5 Object Browser A-3
Figure A-3. Visual C++ Component Gallery A-5
Figure A-4. Delphi Import OLE Control Dialog Box A-7

Tables

Table 2-1. Chapters on Specific Programming Environments 2-6

Table 6-1. Graphical User Interface Control Styles 6-2

Table 7-1. Measurement Types 7-47
Table 7-2. Pulse Type Operations 7-52

Table 10-1. Analysis Control Function Tree 10-3

Table D-1. Data Acquisition Control Error Codes D-1
Table D-2. VISA Control Error Codes D-17
Table D-3. Analysis Error Codes D-20
Table D-4. General ComponentWorks Error Codes D-24

About This Manual

The *Getting Results with ComponentWorks* manual contains the information you need to get started with the ComponentWorks software package. ComponentWorks adds the instrumentation-specific tools for acquiring, analyzing, and displaying data in Visual Basic, Visual C,++, Delphi, and other ActiveX control environments.

This manual contains step-by-step instructions for building applications with ComponentWorks. You can modify these sample applications to suit your needs. This manual does not show you how to use every control or solve every possible programming problem. Use the online reference for further, function-specific information.

To use this manual, you already should be familiar with one of the supported programming environments and Windows 95 or Windows NT.

Organization of This Manual

The *Getting Results with ComponentWorks* manual is organized as follows:

- Chapter 1, *Introduction to ComponentWorks*, contains an overview of ComponentWorks, lists the ComponentWorks system requirements, describes how to install the software, and explains the basics of ActiveX controls.
- Chapter 2, *Getting Started with ComponentWorks*, describes approaches to help you get started using ComponentWorks, depending on your application needs, your experience using ActiveX controls in your particular programming environment, and your specific goals in using ComponentWorks.
- Chapter 3, *Building ComponentWorks Applications with Visual Basic*, describes how you can use the ComponentWorks controls with Visual Basic 5; insert the controls into the Visual Basic environment, set their properties, and use their methods and events; and perform these operations using ActiveX controls in general. This chapter also outlines Visual Basic features that simplify working with ActiveX controls.
- Chapter 4, *Building ComponentWorks Applications with Visual C++*, describes how you can use ComponentWorks controls with Visual C++, explains how to insert the controls into the Visual C++ environment and create the necessary wrapper classes, shows you how

to create an application compatible with the ComponentWorks controls using the Microsoft Foundation Classes Application Wizard (MFC AppWizard) and how to build your program using the ClassWizard with the controls, and discusses how to perform these operations using ActiveX controls in general.

- Chapter 5, *Building ComponentWorks Applications with Delphi*, describes how you can use ComponentWorks controls with Delphi; insert the controls into the Delphi environment, set their properties, and use their methods and events; and perform these operations using ActiveX controls. This chapter also outlines Delphi features that simplify working with ActiveX controls.
- Chapter 6, *Using the User Interface Controls*, describes how you can use the ComponentWorks User Interface (UI) controls to customize your application interface; explains the individual controls and their most commonly used properties, methods, and events; and includes tutorial exercises that give step-by-step instructions on using the controls in simple programs.
- Chapter 7, *Using the Data Acquisition Controls*, describes how you can use the ComponentWorks Data Acquisition (DAQ) controls in your application to perform input and output operations using your DAQ hardware. It explains the individual controls and their most commonly used properties, methods, and events and includes tutorials that give step-by-step instructions on using the controls in simple programs.
- Chapter 8, *Using the GPIB and Serial Controls*, describes how you can use the ComponentWorks Instrument controls in your application to perform input and output operations using GPIB and serial hardware; explains the individual controls and their most commonly used properties, methods, and events; and includes tutorial exercises that give step-by-step instructions on using the controls in simple programs.
- Chapter 9, *Using the VISA Control*, describes the basic structure of the VISA API; shows you how to use the ComponentWorks VISA control in your application to perform input and output operations using GPIB, Serial, and VXI hardware; explains the individual control and its most commonly used properties, methods, and events; and includes tutorial exercises that give step-by-step instructions on using the control in simple programs.
- Chapter 10, *Using the Analysis Controls and Functions*, describes how you can use the ComponentWorks Analysis controls to perform data analysis, manipulation, and simulation. It explains the individual controls and some of their functions and includes a tutorial that gives

step-by-step instructions on using the Analysis controls in a simple program.

- Chapter 11, *Using the DataSocket Control and Tools*, describes how you can use the ComponentWorks DataSocket control to read, write, or share data on a single machine or between multiple machines and includes tutorial exercises that provide step-by-step instructions for using the DataSocket tools.
- Chapter 12, *Building Advanced Applications*, discusses how you can build applications using more advanced features of ComponentWorks, including advanced data acquisition techniques, the DSP Analysis Library, and advanced user interface controls and offers techniques for error tracking, error checking, and debugging.
- Appendix A, *Using Previous Versions of Visual Basic, Visual C++, and Delphi with ComponentWorks*, outlines differences between the current and previous version of the programming environments with respect to using the ComponentWorks controls. This revision of the *Getting Results with ComponentWorks* manual was written with the most current environments available: Visual Basic 5, Visual C++ 5, and Delphi 3. In this appendix, the most current versions are compared with Visual Basic 4, Visual C++ 4.*x*, and Delphi 2.
- Appendix B, *Background Information about Data Acquisition*, provides background information on data acquisition (DAQ) software and hardware specific to the ComponentWorks DAQ controls and describes the underlying architecture used by these controls.
- Appendix C, *Common Questions*, contains a list of answers to frequently asked questions. It contains general ComponentWorks questions as well as specific graphical user interface, data acquisition, instrument control, and analysis library questions.
- Appendix D, *Error Codes*, lists the error codes returned by the ComponentWorks DAQ controls, ComponentWorks VISA control, and Analysis Library functions. It also lists some general ComponentWorks error codes.
- Appendix E, *Distribution and Redistributable Files*, contains information about ComponentWorks 2.0 redistributable files and distributing applications that use ComponentWorks controls.
- Appendix F, *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, including acronyms, abbreviations, metric prefixes, mnemonics, and symbols.
- The *Index* contains an alphabetical list of key terms and topics in this manual, including the page where you can find each one.

Conventions Used in This Manual

The following conventions are used in this manual:

<>

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

-

A hyphen between two or more key names enclosed in angle brackets denotes that you should simultaneously press the named keys—for example, <Control-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** options from the last dialog box.

☞

This icon to the left of bold italicized text denotes a note, which alerts you to important information.

bold

Bold text denotes the names of menus, menu items, parameters, dialog boxes, dialog box buttons or options, icons, and windows.

bold italic

Bold italic text denotes an activity objective, note, caution, or warning.

<Control>

Key names are capitalized.

italic

Italic text denotes variables, emphasis, a cross reference, or an introduction to a key concept.

`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. This font is also used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, operations, properties and methods, filenames and extensions, and for statements and comments taken from programs.

paths — Paths in this manual are denoted using backslashes (\) to separate drive names, directories, folders, and files.

Related Documentation

The following documents contain information you might find useful as you read this manual:

- *Getting Results with ComponentWorks*
- ComponentWorks online reference (available by selecting **Start»Programs»National Instruments ComponentWorks» ComponentWorks Reference**

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 if you have problems with them. To make it easy for you to contact us, this manual contains comment and configuration forms for you to complete. These forms are in Appendix F, *Customer Communication*, at the end of this manual.

1

Introduction to ComponentWorks

This chapter contains an overview of ComponentWorks, lists the ComponentWorks system requirements, describes how to install the software, and explains the basics of ActiveX controls.

What is ComponentWorks?

ComponentWorks is a collection of ActiveX controls for acquiring, analyzing, and presenting data within any compatible ActiveX control container. ActiveX controls also are known as OLE (Object Linking and Embedding) controls, and the two terms can be used interchangeably in this context. Use the online reference for specific information about the properties, methods, and events of the individual ActiveX controls. You can access this information by selecting **Programs»National Instruments ComponentWorks»ComponentWorks Reference** from the Windows **Start** menu.

With ComponentWorks, you can easily develop complex custom user interfaces to display your data, control your National Instruments Data Acquisition (DAQ) boards, and analyze data you acquired or received from some other source. The ComponentWorks package contains the following components:

- User Interface Controls—32-bit ActiveX controls for presenting your data in a technical format. These controls include a graph/strip chart control, sliders, thermometers, tanks, knobs, gauges, meters, LEDs, and switches.
- DAQ Controls—32-bit ActiveX controls for analog I/O, digital I/O, and counter/timer I/O operations using National Instruments DAQ products.
- GPIB, Serial, and VISA Controls—32-bit ActiveX controls for controlling and retrieving data from instruments or devices connected to a GPIB, serial, or VXI port in your computer. You must connect GPIB instruments with a National Instruments GPIB interface card.

- DataSocket Control and Tools (shipped only with the Standard and Full Development Systems)—32-bit ActiveX control and tools for sharing and exchanging data between an application and a number of different targets, including other applications, files, and FTP and Web servers.
- Analysis Library Controls—Functions for statistics, advanced signal processing, windowing, filters, curve-fitting, vector and matrix algebra routines, probability, and array manipulations. These functions are packaged in 32-bit ActiveX controls. Each ComponentWorks package (Base, Standard, and Full Development System) contains different options for analysis. The Base package includes functions for basic statistics and array operations; the Standard Development System includes additional Digital Signal Processing (DSP) functions for signal processing, windowing, and filtering operations; and the Full Development System adds advanced statistics and probability functions.
- Instrument Driver Factory (shipped only with the Full Development System)—Complete development environment for building, compiling, debugging, and analyzing instrument drivers, including a Visual Basic wizard to automate the steps of compiling an instrument driver.
- Instrument Drivers (shipped only with the Full Development System)—Source code and 32-bit DLLs for controlling common GPIB instruments with high-level instrument control routines. You can load and edit the source code in the Instrument Driver Factory. If you do not need to make any changes to the driver, you can use the pre-compiled DLLs.

The ComponentWorks ActiveX controls are designed for use in Visual Basic, a premier ActiveX control container application. Some ComponentWorks features and utilities have been incorporated with the Visual Basic user in mind. However, you can use ActiveX controls in any application that supports them, including Visual C++, Access, and Delphi.

Installing ComponentWorks

The ComponentWorks setup program installs ComponentWorks through a process that lasts approximately five minutes. The ComponentWorks CD contains separate installers for the Instrument Driver Factory and Instrument Drivers (Full Development System only) and for various driver software (NI-DAQ, NI-488, and so on) that you might need.

System Requirements

To use the ComponentWorks ActiveX controls and Analysis Library, you must have the following:

- Microsoft Windows 95 or Windows NT (Windows NT 4.0 for DAQ controls) operating system
- Personal computer using at least a 33 MHz 80486 or higher microprocessor (National Instruments recommends a 66 MHz 80486 or higher microprocessor)
- VGA resolution (or higher) video adapter
- ActiveX control container such as Visual Basic (32-bit version), Visual C++, or Delphi (32-bit version)
- NI-DAQ 5.0 or later for Windows 95 or Windows NT (if you are using DAQ controls)
- Minimum of 16 MB of memory
- Minimum of 10 MB of free hard disk space
- Microsoft-compatible mouse

Installation Instructions

This section provides instructions for installing different pieces of your ComponentWorks software. You can start most of these installers directly from the startup screen that appears when you load the ComponentWorks CD.

Installing the ComponentWorks ActiveX Controls

Complete the following steps to install ComponentWorks.

Note ***To install ComponentWorks on a Windows NT system, you must be logged in with Administrator privileges to complete the installation.***

1. Make sure that your computer and monitor are turned on and that you have installed Windows 95 or Windows NT.
2. Insert the ComponentWorks CD in the CD drive of your computer. From the CD startup screen, click on **Install ComponentWorks 2.0**. If the CD startup screen does not appear, use the Windows Explorer or File Manager to run the `SETUP.EXE` program in the `\Setup` directory on the CD.
3. Follow the instructions on the screen. The installer provides different options for setting the directory in which ComponentWorks is installed and choosing examples for different programming environments. Use the default settings in the installer if you are unsure about them. If necessary, you can run the installer at a later time to install additional components.

Installing From Floppy Disks

If your computer does not have a CD drive, follow these instructions for installing the software.

1. On another computer with a CD drive and disk drive, copy the files in the individual subdirectories of the `\Setup\disks` directory on the CD onto individual 3.5” floppy disks. The floppy disks should not contain any directories and should be labeled `disk1`, `disk2`, and so on, following the name of the source directories.
2. On the target computer, insert the floppy labeled `disk1` and run the `setup.exe` program from the floppy.
3. Follow the on-screen instructions to complete the installation.

Installing the Instrument Driver Factory

If you have purchased the ComponentWorks Full Development System, you can install the Instrument Driver Factory (IDF) and LabWindows/CVI development environment. Use the following procedure to install the IDF.

1. Make sure that your computer and monitor are turned on and that you have installed Windows 95 or Windows NT.
2. Insert the ComponentWorks CD in the CD drive of your computer. From the CD startup screen, click on **Install Instrument Driver Factory**. If the CD startup screen does not appear, use the Windows Explorer or File Manager to run `SETUP.EXE` in the `\CVI\DISK1` directory and `SETUP.EXE` in the `\Instrument Driver Factory` directory on the CD.
3. Follow the instructions on the screen to install the LabWindows/CVI environment and the Instrument Driver Factory for compiling instrument drivers from Visual Basic.

Installing the Instrument Drivers DLLs

If you have purchased the ComponentWorks Full Development System, you can install any of more than 600 instrument drivers and their source code. Use the following procedure to install an instrument driver.

1. Make sure that your computer and monitor are turned on and that you have installed Windows 95 or Windows NT.
2. Insert the ComponentWorks CD in the CD drive of your computer. From the CD startup screen, click on **Install Instrument Drivers**. If the CD startup screen does not appear, use the Windows Explorer or File Manager to run `SETUP.EXE` in the `\Instrument Drivers` directory on the CD.
3. Following the instructions in the installation program, select the drivers you want to install. You can install additional drivers later by running the setup program again.
4. Follow the installer instructions to complete the installation.

Installed Files

The ComponentWorks setup program installs the following groups of files on your hard disk.

- ActiveX Controls, Documentation, and other associated files

  Directory: `\Windows\System\`

  Files: `cwdaq.ocx`, `cwdaq.dep`, `cwui.ocx`, `cwui.dep`, `cwanalysis.ocx`, `cwanalysis.dep`, `cwinstr.ocx`, `cwinstr.dep`, `cwvisa.ocx`, `cwvisa.dep`, `cwds.ocx`, `cwds.dep`, `cwref.hlp`, `cwref.cnt`

- Example Programs and Applications

  Directory: `\ComponentWorks\samples\...`

- Tutorial Programs

  Directory: `\ComponentWorks\tutorials\...`

- DataSocket Server (Standard and Full Development Systems only)

  Directory: `\ComponentWorks\DataSocket Server`

- Miscellaneous Files

  Directory: `\ComponentWorks\`

If you have purchased the ComponentWorks Full Development System, you can install the following components.

- LabWindows/CVI Development Environment

  Directory: `\CVI`

- Instrument Driver Factory

  Directory: `\ComponentWorks\Instrument Driver Factory`

- Instrument Driver Files

  Directory: `\Instrument Drivers\...`

Note ***You select the location of the*** `\ComponentWorks\...` ***and*** `\CVI\...` ***directories during installation.***

About the ComponentWorks Controls

Before learning how to use ComponentWorks, you should be familiar with using ActiveX controls. This section outlines some background information about ActiveX controls, in particular the ComponentWorks controls. If you are not already familiar with the concepts outlined in this section, make sure you understand them before continuing. You also might want to refer to your programming environment documentation for more information on using ActiveX (OLE) controls in your particular environment.

Properties, Methods, and Events

ActiveX controls consist of three different parts—properties, methods, and events—used to implement and program the controls.

Properties are the attributes of a control. These attributes describe the current state of the control and affect the display and behavior of the control. The values of the properties are stored in variables that are part of the control.

Methods are functions defined as part of the control. Methods are called with respect to a particular control and usually have some effect on the control itself. The operation of most methods also is affected by the current property values of the control.

Events are notifications generated by a control in response to some particular occurrence. The events are passed to the control container application to execute a particular subroutine in the program (event handler).

For example, the ComponentWorks Graph control has a wide variety of properties that determine how the graph looks and operates. To customize the graph appearance and behavior, set properties for color, axes, scaling, tick marks, cursors, and labels.

The Graph control also has a series of high-level methods, or functions, that you can invoke to set several properties at once and to perform a particular operation. For example, you can use the `PlotY` method to pass an array of data to the Graph control.

The Graph control generates events when particular operations take place. For example, when you drag a cursor on the graph, the control passes an event to your program so you can respond to cursor movements. This might be useful if you might want to retrieve the X- and Y-coordinate positions of a cursor as it is being dragged and display the coordinates in text boxes.

☞ **Note** ***Use the ComponentWorks online reference for specific information about the properties, methods, and events of the ActiveX controls. You can access the online reference by selecting* Programs»National Instruments ComponentWorks» ComponentWorks Reference *from the Windows* Start *menu.***

Object Hierarchy

As described in the previous section, each ActiveX control has properties, methods, and events. These three parts are stored in a *software object*, which is the piece of software that makes up the ActiveX control and contains all its parts. Certain ActiveX controls are very complex, containing many different parts (properties). For this reason, complex ActiveX controls are often subdivided into different software objects, the sum of which make up the ActiveX control. Each individual object in a control contains some specific parts (properties) and functionality (methods and events) of the ActiveX control. The relationships between different objects of a control are maintained in an object hierarchy. At the top of the hierarchy is the actual control itself.

This top-level object contains its own properties, methods, and events. Some of the top-level object properties are actually references to other objects that define specific parts of the control. Objects below the top-level have their own methods and properties, and their properties can be references to other objects. The number of objects in this hierarchy is not limited.

Another advantage of subdividing controls is the re-use of different objects between different controls. One object might be used at different places in the same object hierarchy or in several different controls/object hierarchies.

Figure 1-1 shows part of the object hierarchy of the ComponentWorks Slide control.

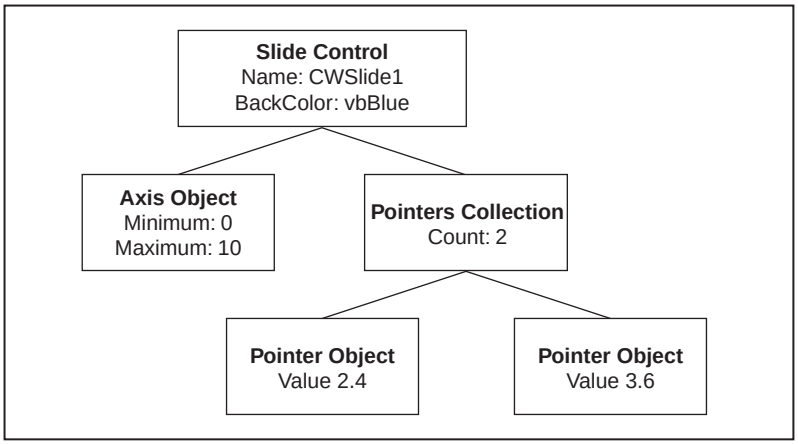

Figure 1-1. Slide Control Object Hierarchy

The Slide object contains some of its own properties, such as `Name` and `BackColor`. It also contains properties such as `Axis` and `Pointers`, which are separate objects from the Slide object. The Axis object contains all the information about the axis used on the slide and has properties such as `Maximum` and `Minimum`. The Pointers Collection object contains several Pointer objects of its own, each describing one pointer on the Slide control. Each Pointer object has properties, such as `Value`, while the Pointers Collection object has the property `Count`. The Pointers Collection object is a special type of object referred to as a collection, which is described in the *Collection Objects* section.

Collection Objects

One object can contain several objects of the same type. For example, a Graph object contains several Axis objects, each representing one of the axes on the graph. Additionally, the number of objects in the group of objects might not be defined and might change while the program is running (that is, you can add or remove axes as part of your program). To handle these groups of objects more easily, an object called a *Collection* is created.

A collection is an object that contains or stores a varying number of objects of the same type. It also can be regarded as an array of objects. The name of a collection object is usually the plural of the name of the object type contained within the collection. For example, a collection of Pointer objects is referred to as Pointers, a collection of Plot objects as Plots, and a collection of Axis objects as Axes. In the ComponentWorks software, the terms *object* and *collection* are not used, only the type names Pointer and Pointers.

Each collection object contains an `Item` method that you can use to access any particular object stored in the collection. Refer to *Changing Properties Programmatically* later in this chapter for information about the `Item` method and how to access particular objects stored in the collection.

Setting the Properties of an ActiveX Control

You can set the properties of an ActiveX control from its property pages or from within your program.

Using Property Pages

Property pages are common throughout the Windows 95 and Windows NT interface. When you want to change the appearance or options of a particular object, right click on the object and select **Properties**. A property page or tabbed dialog box appears with a variety of properties that you can set for that particular object. You customize ActiveX controls in exactly the same way. Once you place the control on a form in your programming environment, right click on the control and select **Properties…** to customize the appearance and operation of the control.

Use the property pages to set the property values for each ActiveX control while you are creating your application. The property values you select at this point represent the state of the control at the beginning of your application. You can change the property values from within your program, as shown in the next section, *Changing Properties Programmatically*.

In some programming environments (such as Visual Basic and Delphi), you have two different property pages. The property page common to the programming environment is called the *default property sheet*; it contains the most basic properties of a control.

Your programming environment assigns default values for some of the basic properties, such as the control name and the tab order. You must edit these properties through the default property sheet.

The following illustration shows the Visual Basic default property sheet for the CWGraph control.

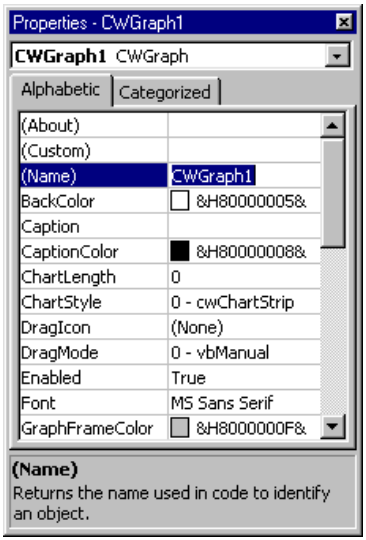

Figure 1-2. Visual Basic Default Property Sheets

The second property sheet is called the *custom property page*. The layout and functionality of the custom property pages vary for different controls. The following illustration shows the custom property page for the CWGraph control.

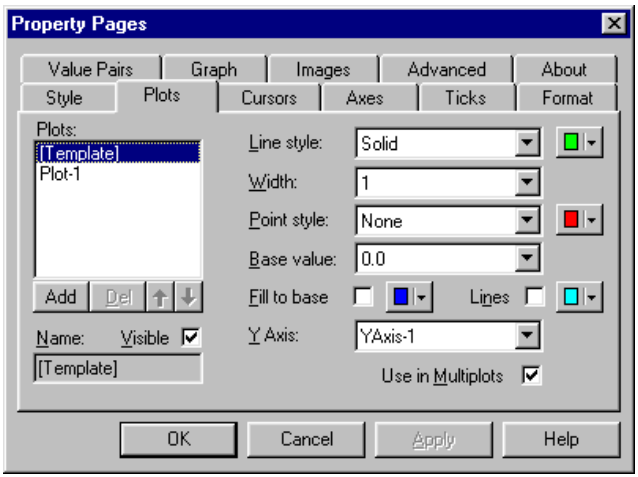

Figure 1-3. ComponentWorks Custom Property Pages

Changing Properties Programmatically

You also can set or read the properties of your controls programmatically. For example, if you want to change the state of an LED control during program execution, change the `Value` property from `True` to `False` or from `False` to `True`. The exact syntax for reading and writing property values depends on your programming language, so consult the appropriate chapter for using your programming environment. In this discussion, properties are set with Visual Basic syntax, which is similar to most implementations.

Each control you create in your program has a name (like a variable name) which you use to reference the control in your program. You can set the value of a property on a top-level object with the following syntax.

```
name.property = new_value
```

For example, you can change the `Value` property of an LED control to `off` using the following line of code, where `CWButton1` is the default name of the LED control. An LED is one style of the CWButton control.

```
CWButton1.Value = False
```

To access properties of sub-objects referenced by the top-level object, use the control name, followed by the name of the sub-object and the property name. For example, consider the following code for the ComponentWorks data acquisition analog input (CWAI) control.

```
CWAI1.ScanClock.Frequency = 10000
```

In the above code, `ScanClock` is a property of the CWAI control and refers to a CWAIClock object. `Frequency` is one of several CWAIClock properties. The CWAI control also has a ChannelClock property that refers to a different CWAIClock object.

You can retrieve the value of control properties from your program in the same way. For example, you can print the value of the LED control.

```
Print CWButton1.Value
```

You can display the frequency used by the CWAI control in a Visual Basic text box with the following code.

```
Text1.Text = CWAI1.ScanClock.ActualFrequency
```

Item Method

To access an object or its properties in a collection, use the `Item` method of the collection object. For example, set the value of the second pointer on a slide with the following code.

```
CWSlide1.Pointers.Item(2).Value = 5.0
```

The term `CWSlide1.Pointers.Item(2)` refers to the second Pointer object in the Pointers collection of the Slide object. The parameter of the `Item` method is either an integer representing the (one-based) index of the object in the collection or a string with the name of one of the objects in the collection.

```
CWSlide1.Pointers.Item("TemperaturePointer")
```

Because the `Item` method is the most commonly used method on a collection, it is referred to as the *default method*. Therefore, some programming environments do not require you to specify the `.Item` method. For example, in Visual Basic

```
CWSlide1.Pointers(2).Value = 5.0
```

is programmatically equivalent to

```
CWSlide1.Pointers.Item(2).Value = 5.0
```

Working with Control Methods

ActiveX controls and objects have their own methods, or functions, that you can call from your program. Methods can have parameters that are passed to the method and return values that pass information back to your program.

For example, the `PlotY` method for the ComponentWorks Graph control has a required parameter—the array of data to be plotted—that you must include when you call the method. If you want to plot the data returned from an Analog Input control, use the following line of code (the array `ScaledData` is automatically generated by the CWAI control).

```
CWGraph1.PlotY ScaledData
```

The `PlotY` method has additional parameters that are optional in some programming environments. For example, in addition to the first parameter representing the data to be plotted, you can pass a second parameter to represent the initial value for the X axis, a third parameter for an incremental change on the X axis corresponding to each data point, and a fourth parameter that determines how the graph should handle two-dimensional data.

```
CWGraph1.PlotY ScaledData, 0.0, 1.0, True
```

Depending on your programming environment, the parameters might be enclosed in parentheses. Visual Basic does not use parentheses to pass parameters if the function or method is not assigned a return variable. The `AcquireData` method in the DAQ Analog Input control has the following form when used with a return variable `lErr`.

```
lErr = CWAI1.AcquireData(ScaledData, BinaryCodes, 1)
```

Developing Event Handler Routines

After you configure your controls on a form, you can create event handler routines in your program to respond to events generated by the controls. For example, the DAQ Analog Input control has an `Acquired_Data` event that *fires* (occurs) when the acquired data is ready to be processed, based on the acquisition options you have configured in the control property pages.

You can configure the control to continuously collect 1,000 points of data from a particular channel at a rate of 1,000 points per second. Once every second, the data buffer is ready and the `Acquired_Data` event is fired. In your `Acquired_Data` event routine, you can write code to analyze the data buffer, plot it, or store it to disk.

To develop the event routine code, most programming environments generate a skeleton function to handle each event. Chapters 3, 4, and 5 of this manual outline how to generate these function skeletons to build your event handler routines. For example, the Visual Basic environment generates the following function skeleton into which you insert the functions to call when the `AcquireData` event occurs.

```
Private Sub CWAI1_AcquiredData(ScaledData As Variant,
  BinaryCodes As Variant)

End Sub
```

In most cases, the event also returns some data to the event handler, such as the `ScaledData` and `BinaryCodes` arrays in the previous example, that can be used in your event handler routine.

Using the Analysis Library and Instrument Driver DLLs

The ComponentWorks Analysis Library is packaged as a set of ActiveX controls, while the instrument drivers are packaged as 32-bit DLLs. You can add analysis functions to your project in the same way you add user interface or data acquisition controls. After adding the Analysis controls to your programming environment, use the analysis functions like any other method on a control. To use any specific function, place the appropriate Analysis control on a form. In your program, call the name of the control followed by the name of the analysis function:

```
MeanValue = CWStat1.Mean (Data)
```

Consult Chapter 10, *Using the Analysis Controls and Functions*, and the ComponentWorks online reference for more information on the individual analysis functions and their use.

To use the instrument driver DLLs, you must add a reference to the DLL in your project. After adding the appropriate reference to your project, you can use the functions included in the DLLs to control your instruments.

The Online Reference—Learning the Properties, Methods, and Events

The ComponentWorks online reference contains detailed information on each control and its associated properties, methods, and events. Refer to ComponentWorks online reference when you are using a control for the first time. Remember that many of the ComponentWorks controls share sub-objects and properties, so when you learn how to use one control, you also learn how to use others. You can open the online reference from within most programming environments by clicking on the **Help** button in the custom property pages, or you can open it from the Windows **Start** menu by selecting **Programs»National Instruments ComponentWorks» ComponentWorks Reference**.

Some programming environments have built-in mechanisms for detailing the available properties, methods, and events for a particular control and sometimes include automatic links to the help file. Refer to the chapter on your particular programming environment to learn about additional tools.

2

Getting Started with ComponentWorks

This chapter describes approaches to help you get started using ComponentWorks, depending on your application needs, your experience using ActiveX controls in your particular programming environment, and your specific goals in using ComponentWorks.

This chapter elaborates on the following steps for getting started with ComponentWorks.

1. Install and configure driver software if you plan to use hardware I/O controls.
2. Explore the ComponentWorks documentation.
3. Become familiar with the examples structure.
4. Develop your application.
5. Seek answers to your questions from additional information sources.

So far you should have installed the ComponentWorks software following the instructions in Chapter 1, *Introduction to ComponentWorks*, as well as the programming environment you plan to use with the controls. Chapter 1 also includes basic information about ActiveX controls. If you are unfamiliar with ActiveX controls, read Chapter 1 before proceeding.

Installing and Configuring Driver Software

If you need to use any of the hardware I/O controls in ComponentWorks (data acquisition, GPIB, serial, or VISA), you must install and configure the corresponding driver software before using these controls.

Driver software performs the low-level calls to your hardware. It is configured using a separate configuration utility provided with the driver software. In some cases, the configuration utility also provides parameters or values you need to use in your controls. For example, with the device number defined in the data acquisition (DAQ) configuration utility, you can select a specific piece of hardware in your controls.

Install the most current driver available. Sometimes, the ComponentWorks controls require features provided only in newest versions of the driver. The ComponentWorks CD includes installers for different drivers compatible with the ComponentWorks version on the CD. Check the `\Drivers` directory on the CD for these installation programs. If necessary, you can download newer versions of the drivers from the National Instruments Web or FTP sites.

To run the installation and configuration programs, follow the directions provided with each driver. Each driver includes a readme file or printed document that provides the latest information as well as any operating system details. Appendix B, *Background Information about Data Acquisition*, includes some commonly used information about the DAQ driver.

Exploring the ComponentWorks Documentation

The printed and online manuals contain the information necessary to learn and use the ComponentWorks controls to their full capabilities. The manuals are divided into different sections. Each section addresses a specific step on the learning curve.

Use the *Getting Results with ComponentWorks* manual to learn how to develop simple applications with the ComponentWorks controls. The manual contains information you can use in specific circumstances, such as debugging particular problems or distributing applications.

After you understand the operation and organization of the controls, use the ComponentWorks online reference to obtain information about specific features of each control.

Getting Results with ComponentWorks Manual

The *Getting Results with ComponentWorks* manual contains four different sections.

Section 1—Includes Chapter 1, *Introduction to ComponentWorks*, and Chapter 2, *Getting Started with ComponentWorks*. These chapters provide introductory information about ComponentWorks, including installation procedures, a basic overview of ActiveX controls, and information about getting started with the software.

Section 2—Includes Chapter 3, *Building ComponentWorks Applications with Visual Basic*, Chapter 4, *Building ComponentWorks Applications with Visual C++*, and Chapter 5, *Building ComponentWorks Applications with Delphi*. These chapters describe how to use ActiveX controls in the most commonly used programming environments—Visual Basic, Visual C++, and Borland Delphi.

If you are familiar with using ActiveX controls in these environments, you should not need to read these chapters. If you are using the controls in another environment, consult your programming environment documentation for information about using ActiveX controls. You can check the ComponentWorks Support Web site for information about additional environments.

Section 3—Includes chapters 6 through 12. These chapters describe the basic operation of the different controls in ComponentWorks. Each chapter contains an overview of a group of controls, describing their most commonly used properties, methods, and events. The description also includes short code segments to illustrate programmatic control and a number of simple tutorials that step you through building an application with those controls.

Section 4—Includes the appendices of this manual. The appendices contain additional information about using ComponentWorks, including information about using ComponentWorks in previous versions of Visual Basic, Visual C++, and Delphi; DAQ basics; common questions; and error code descriptions. If you have a particular question when using the ComponentWorks software, remember to check the appendices as well as the other documentation for an answer.

ComponentWorks Online Reference

The ComponentWorks online reference includes complete reference information for all controls—all properties, methods, and events for every control—as well as the text from *Getting Results with ComponentWorks* and the *Instrument Driver Factory* manual.

To use the online reference efficiently, you should understand the material presented in the *Getting Results with ComponentWorks* manual about using ComponentWorks ActiveX controls.

After going through the *Getting Results with ComponentWorks* manual and tutorials, use the online reference as your main source of information. Refer to it when you need specific information about a particular feature in ComponentWorks.

Accessing the Online Reference

You can open the online reference from the Windows **Start** menu (**Programs»National Instruments ComponentWorks» ComponentWorks Reference**). The reference opens to the main contents page. From the contents page, you can browse the contents of the online reference or search for a particular topic.

Most programming environments support some type of automatic link to the online reference (help) file from within their environment, often the <F1> key. Try selecting the control on a form or placing the cursor in code specific to a control and pressing <F1> to evoke the online reference.

In most environments, the property pages for the ComponentWorks controls include a **Help** button that provides information about the property pages.

Finding Specific Information

To find information about a particular control or feature of a control, select the **Index** tab under the **Help Topics** page. Enter the name of the control, property, method, or event. Control names always begin with `CW` (for example, `CWGraph`, `CWAI`, and `CWStat`). Property, method, and event names are identical to those used in the code (for example, `Axes`, `Plot`, `Channels`, and `Font`).

One group of objects that frequently generates questions are the Collection objects. Search the online reference for Collections and the `Item` method for more information. You also can find information about collection objects in the *Collection Objects* section of Chapter 1, *Introduction to ComponentWorks*.

Becoming Familiar with the Examples Structure

The examples installed with ComponentWorks show you how to use the controls in applications. You can use these examples as a reference to become more familiar with the use of the controls, or you can build your application by expanding one of the examples.

When you install ComponentWorks, you can install examples for selected programming environments. The examples are located in the `\ComponentWorks\samples` directory and organized by programming environment (`\Visual Basic`, `\Visual C++`, and so on) and control group (`\UI`, `\DAQ`, and so on). Within these directories, the examples are further subdivided by functionality.

The online reference includes a searchable list of all the examples included with ComponentWorks. Select **Examples** to see the list of examples.

The examples exist in the oldest commonly used version of a programming environment, such as Visual Basic 4. If you open an example in a newer version, you might see a warning message indicating that the example will be converted to the new version.

Developing Your Application

Depending on your experience with your programming environment, ActiveX controls, and ComponentWorks, you can get started using ComponentWorks in some of the following ways.

Are you new to your particular programming environment?

Spend some time using and programming in your development environment. Check the documentation that accompanies your programming environment for getting started information or tutorials, especially tutorials that describe using ActiveX controls in the environment. If you have specific questions, search the online documentation of your development environment. After becoming familiar with the programming environment, continue with the following steps.

Are you new to using ActiveX controls or do you need to learn how to use ActiveX controls in a specific programming environment?

Make sure you have read and understand the information regarding ActiveX controls in Chapter 1, *Introduction to ComponentWorks*, and the appropriate chapter about your specific programming environment. Refer to Table 2-1 to find out which chapter you should read for your specific programming environment.

If you use Borland C++ Builder, most of Chapter 5 pertains to you. If you use another programming environment, see the ComponentWorks Support Web site for current information about particular environments.

Table 2-1. Chapters on Specific Programming Environments

Environment	Read This Chapter
Microsoft Visual Basic	Chapter 3, *Building ComponentWorks Applications with Visual Basic*
Microsoft Visual C++	Chapter 4, *Building ComponentWorks Applications with Visual C++*
Borland Delphi	Chapter 5, *Building ComponentWorks Applications with Delphi*

Regardless of the programming environment you use, consult its documentation for information about using ActiveX controls. After becoming familiar with using ActiveX controls in your environment, continue with the following steps.

Are you familiar with ActiveX controls but need to learn ComponentWorks controls, hierarchies, and features?

If you are familiar with using ActiveX controls, including collection objects and the `Item` method, read the chapters pertaining to the controls you want to use. Chapters 6 through 11 provide basic information about each ComponentWorks control and describe their most commonly used properties, methods, and events. The chapters also offer simple tutorials to help you become more familiar with using the controls. Solutions to each of the tutorials are installed with your software in the `\ComponentWorks\tutorials` directory.

After becoming familiar with the information in these chapters, try building applications with the ComponentWorks controls. You can find detailed information about all properties, methods, and events for every control in the online reference.

Do you want to develop applications quickly or modify existing examples?

If you are very familiar with using ActiveX controls, including collections and the `Item` method, and have some experience using ComponentWorks or other National Instruments products, you can get started more quickly by looking at the examples.

Most examples demonstrate how to perform operations with a particular control or group of controls. Generally, the examples avoid presenting complex operations on more than one group of controls, such as UI and DAQ. To become familiar with an individual group of controls, look at the

example for that particular group. Then, you can combine different programming concepts from the different groups in your application.

The examples include comments to provide more information about the steps performed in the example. The examples avoid performing complex programming tasks specific to one programming environment; instead, they focus on showing you how to perform operations using the ComponentWorks controls. When developing applications with ActiveX controls, you do a considerable amount of programming by setting properties in the property pages. Check the value of the control properties in the examples because the values greatly affect the operation of the example program. In some cases, the actual source code used by an example might not greatly differ from other examples; however, the values of the properties change the example significantly.

Seeking Information from Additional Sources

After working with the ComponentWorks controls, you might need to consult other sources if you have questions. The following sources can provide you with more specific information.

- *Getting Results with ComponentWorks* Appendices—The appendices include information about using previous versions of development tools, DAQ basics, common questions, and error descriptions.
- *ComponentWorks Instrument Driver* Manual—If you are using the Instrument Driver Factory (ComponentWorks Full Development system only), you have received a separate manual detailing this tool and its use.
- ComponentWorks Online Reference—The online reference includes the complete reference documentation, text of the *Getting Results with ComponentWorks* manual as well as additional information. If you are not able to find a particular topic in the index, choose the **Find** tab in the **Help Topics** page and search the complete text of the online reference.
- ComponentWorks Support Web Site—The ComponentWorks Support Web site, as part of the National Instruments Support Web site (`www.natinst.com/support`), contains support information, updated continually. You can find application and support notes and information about using ComponentWorks in additional programming environments. The Web site also contains the KnowledeBase, a searchable database containing thousands of entries answering common questions related to the use of ComponentWorks and other National Instruments products.

3

Building ComponentWorks Applications with Visual Basic

This chapter describes how you can use the ComponentWorks controls with Visual Basic 5; insert the controls into the Visual Basic environment, set their properties, and use their methods and events; and perform these operations using ActiveX controls in general. This chapter also outlines Visual Basic features that simplify working with ActiveX controls.

At this point you should be familiar with the general structure of ActiveX controls described in Chapter 1, *Introduction to ComponentWorks*. The individual ComponentWorks controls are described later in this manual.

Note ***The descriptions and figures in this chapter apply specifically to the Visual Basic 5 environment. Although most of this information applies to Visual Basic 4 as well, some menu and option names and environment components are different. See Appendix A, Using Previous Versions of Visual Basic, Visual C++, and Delphi with ComponentWorks, for a description of key differences between these versions of Visual Basic as they apply to ComponentWorks.***

Developing Visual Basic Applications

The following procedure explains how you can start developing Visual Basic applications with ComponentWorks.

1. Select the type of application you want to build. Initially select a Standard EXE for your application type.
2. Design the form. A form is a window or area on the screen on which you can place controls and indicators to create the user interface for your program. The toolbox in Visual Basic contains all of the controls available for developing the form.

3. After placing each control on the form, configure the properties of the control using the default and custom property pages.

   Each control on the form has associated code (event handler routines) in your Visual Basic program that automatically executes when the user operates that control.

4. To create this code, double click on the control while editing your application and the Visual Basic code editor opens to a default event handler routine.

Loading the ComponentWorks Controls into the Toolbox

Before building an application using the ComponentWorks controls and libraries, you must add them to the Visual Basic toolbox. The ComponentWorks ActiveX controls are divided into different groups including user interface controls (`CWUI.OCX`), data acquisition controls (`CWDAQ.OCX`), and analysis library controls (`CWANALYSIS.OCX`). The exact list of controls depends on the ComponentWorks package you use.

Use the following procedure to add ComponentWorks controls to the project toolbox.

1. In a new Visual Basic project, right click on the toolbox and select **Components...**.
2. Scroll down to the ComponentWorks controls, which you can find in the Controls list, beginning with National Instruments.
3. Place a checkmark in the box next to the control groups to select the controls you want to use in your project. If the ComponentWorks controls are not in the list, select the control files from the `\Windows\System(32)` directory by pressing the **Browse** button.

If you need to use the ComponentWorks controls in several projects, create a new default project in Visual Basic 5 to include the ComponentWorks controls and serve as a template.

1. Create a new Standard EXE application in the Visual Basic environment.
2. Add the ComponentWorks controls to the project toolbox as described in the preceding procedure.
3. Save the form and project in the `\Template\Projects` directory under your Visual Basic directory.
4. Give the form and project a descriptive name, such as `CWForm` and `CWProject`.

After creating this default project, you have a new option, `CWProject`, that includes the ComponentWorks controls in the **New Project** dialog by default. You can create additional project templates with different combinations of controls.

Building the User Interface Using ComponentWorks

After you add the ComponentWorks controls to the Visual Basic toolbox, use them to create the front panel of your application. To place the controls on the form, select the corresponding icon in the toolbox and click and drag the mouse on the form. This step creates the corresponding control. After you create controls, move and size them by using the mouse. To move a control, click and hold the mouse on the control and drag the control to the desired location. To resize a control, select the control and place the mouse pointer on one of the hot spots on the border of the control. Drag the border to the desired size. Notice that the icons for all but the user interface controls cannot be resized and will not be visible at run time.

Once ActiveX controls are placed on the form, you can edit their properties using their property sheets. You can also edit the properties from within the Visual Basic program at run time.

Using Property Sheets

After placing a control on a Visual Basic form, configure the control by setting its properties in the Visual Basic property pages (see Figure 3-1) and ComponentWorks custom control property pages (see Figure 3-2). Visual Basic assigns some default properties, such as the control name and the tab order. When you create the control, you can edit these stock properties in the Visual Basic default property sheet. To access this sheet, select a control and select **Properties Window** from the **View** menu, or press <F4>. To edit a property, highlight the property value on the right side of the property sheet and type in the new value or select it from a pull down menu. The most important property in the default property sheet is `Name`, which is used to reference the control in the program.

Edit all other properties of an ActiveX control in the custom property sheets. To open the custom property sheets, right click on the control on the form and select **Properties...** or select the controls and press <Shift-F4>.

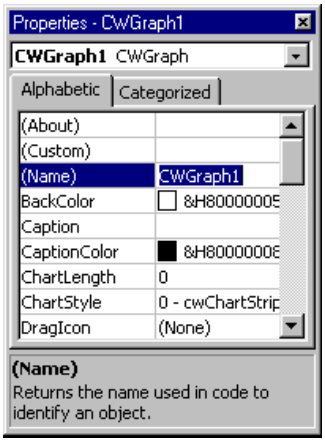

Figure 3-1. Visual Basic Property Pages

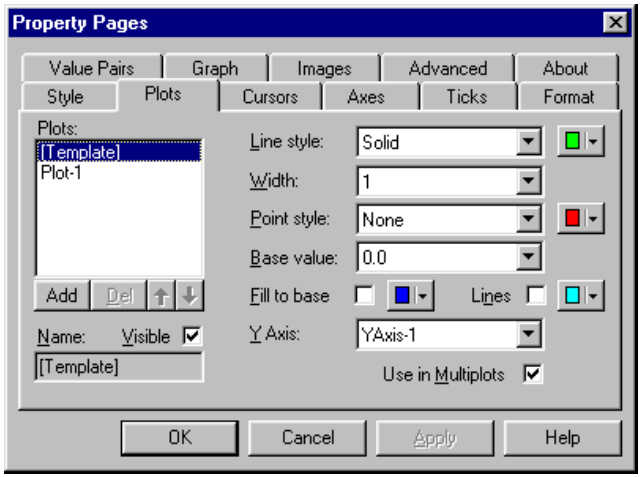

Figure 3-2. ComponentWorks Custom Property Pages

Using Your Program to Edit Properties

You can set and read the properties of your controls programmatically in Visual Basic. Use the name of the control with the name of the property as you would with any other variable in Visual Basic. The syntax for setting a property in Visual Basic is `name.property` = `new value`.

For example, if you want to change the state of an LED control during program execution, change the `Value` property from `True` to `False` or `False` to `True`.

```
CWButton1.Value = False
```

Some properties of a control can be objects that have their own properties. In this case, specify the name of the control, sub-object, and property separated by periods. For example, consider the following code for the DAQ analog input (CWAI) control.

```
CWAI1.ScanClock.Frequency = 10000
```

In the above code, `ScanClock` is a property of the AI control and refers to a CWAIClock object. `Frequency` is a property of the CWAIClock object.

You can retrieve the value of control properties from your program in the same way. For example, you can print the value of an LED control.

```
Print CWButton1.Value
```

In Visual Basic most controls have a default property such as `Value` for the Knob, Button, and Slide controls. You can access the default property of a control by using its control name (without the property name attached).

```
CWSlide1 = 5.0
```

is programmatically equivalent to

```
CWSlide1.Value = 5.0
```

Consult the *Setting the Properties of an ActiveX Control* section of Chapter 1, *Introduction to ComponentWorks*, for information on setting properties programmatically, specifically with objects and properties in collections. The Object Browser, a Visual Basic tool that is helpful with using properties in your code, is described in detail in the *Using the Object Browser to Build Code in Visual Basic* section of this chapter.

Working with Control Methods

Calling the methods of an ActiveX control in Visual Basic is similar to working with the control properties. To call a method, add the name of the method after the name of the control (and sub-object if applicable). For example, you can call the `Start` method on the DAQ analog input control.

```
CWAI1.Start
```

Methods can have parameters that you pass to the method, and return values that pass information back to your program. For example, the `PlotY` method for the ComponentWorks Graph control has a required parameter—the array of data to be plotted—that you must include when you call the method. If you want to plot the data returned from an Analog Input control, use the following line of code.

```
CWGraph1.PlotY ScaledData
```

The `PlotY` method has some additional parameters that are optional. These are added after the `data` parameter, separated by commas, if desired.

In Visual Basic if you call a method without assigning a return variable, any parameters passed to the method are listed after the method name, separated by commas without parentheses.

```
CWAI1.AcquireData Voltages, BinaryCodes, 1.0
```

If you assign the return value of a method to a return variable, enclose the parameters in parentheses.

```
lErr = CWAI1.AcquireData(Voltages, BinaryCodes, 1.0)
```

You can use the Visual Basic Object Browser to add method calls to your program.

Developing Control Event Routines

After you configure your controls in the forms editor, write Visual Basic code to respond to events on the controls. The controls generate these events in response to user interactions with the controls or in response to some other occurrence in the control. To develop the event handler routine code for an ActiveX control in Visual Basic, double click on the control to open the code editor, which automatically generates a default event handler routine for the control. The event handler routine skeleton includes the control name, the default event, and any parameters that are passed to the event handler routine. The following code is an example of the event routine generated for the Slide control. This event routine (`PointerValueChanged`) is called when the value of the slide is changed by the user or by some other part of the program.

```
Private Sub CWSlide1_PointerValueChanged(ByVal
   Pointer As Long, Value As Variant)
End Sub
```

To generate an event handler for a different event of the same control, double click the control to generate the default handler, and select the desired event from the right pull-down menu in the code window, as shown in the following illustration.

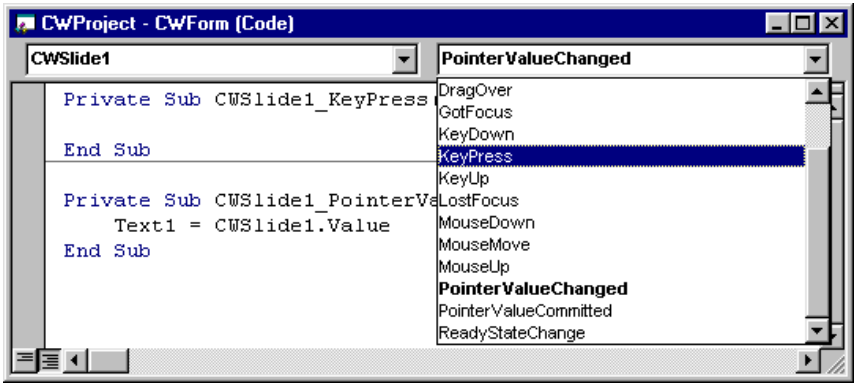

Figure 3-3. Selecting Events in the Code Window

Use the left pull-down menu in the code window to change to another control without going back to the form window.

Using the ComponentWorks Instrument Driver DLLs in Visual Basic

The ComponentWorks Full Development System comes with a library of instrument drivers and the Instrument Driver Factory. An instrument driver is software that handles the details of control and communications with a specific instrument.

An instrument driver consists of a set of high-level functions that controls a specific programmable instrument. Each function corresponds to a programmatic operation such as initialization, configuration, and measurement. If you did not purchase the ComponentWorks Full Development Kit and find that you need instrument drivers, contact National Instruments for information on ordering drivers.

The ComponentWorks instrument drivers are compiled to 32-bit DLLs in the Instrument Driver Factory. Before you can use any of the instrument driver functions in your program, you must add a reference to the instrument driver DLL to your project.

To add an instrument driver DLL to your project, select **References...** from the **Project** menu. In the References dialog window, press the **Browse** button and then move to the directory that contains the instrument driver DLL—the default is `..\ComponentWorks\Instrument Drivers\`.... Select the instrument driver DLL you wish to add to your project.

After you have added the reference to the DLL, you can use any of the functions in the instrument driver without having to do any more declarations. An example of a typical instrument driver function is the initialization function for the Fluke 45 multimeter.

```
lerr = fl45_init(2, 1, 1, InstrID)
```

All the functions in each instrument driver are listed and described in the corresponding help file that is installed with each driver. After you add the appropriate reference to your project for using an instrument driver DLL, use the object browser to view the functions and parameters available in each instrument driver. Refer to the *Using the Object Browser to Build Code in Visual Basic* section for more information on using the object browser to help you build your program.

For more information about the instrument drivers and related development tools, see the *Instrument Driver Factory* manual.

Using the Object Browser to Build Code in Visual Basic

Visual Basic includes a tool called the Object Browser that you can use to work with ActiveX controls and instrument driver DLLs while creating your program. The Object Browser displays a detailed list of the available properties, methods, and events for a particular control, as well as the functions of an instrument driver. It presents a three-step hierarchical view of controls or libraries and their properties, methods, functions, and events. To open the Object Browser, select **Object Browser** from the **View** menu, or press <F2>.

In the Object Browser, use the top left pull-down menu to select a particular ActiveX control file, library, or instrument driver. You can select any currently loaded control or driver. The Classes list on the left side of the object browser displays a list of controls, objects, and function classes available in the selected control file or driver.

Figure 3-4 shows the ComponentWorks User Interface (UI) control file selected in the Object Browser. The Classes list shows all the UI controls and associated object types. Each time you select an item from the Classes list in the Object Browser, the Members list on the right side displays the properties, methods, and events for the selected object or class.

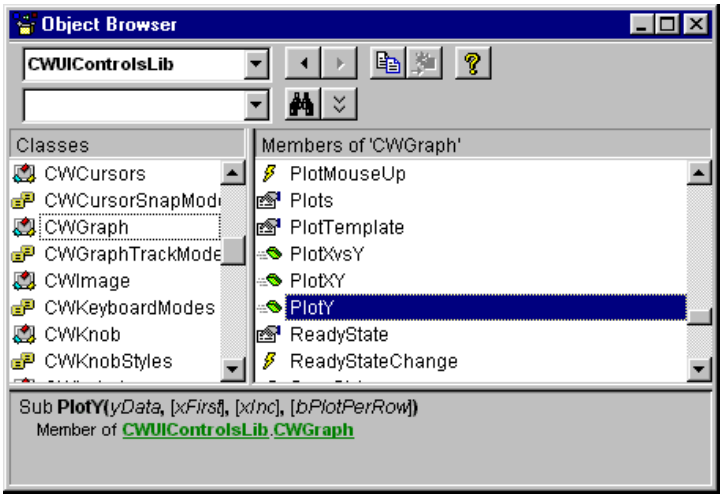

Figure 3-4. Viewing CWGraph in the Object Browser

When you select an item in the Members list, the prototype and description of the selected property, method, or function are displayed at the bottom of the Object Browser dialog box. In Figure 3-4, the CWGraph control is selected from the list of available UI objects. For this control, the `PlotY` method is selected and the prototype and description of the method appear in the dialog box. The prototype of a method or function lists all parameters, required and optional.

When you select a property of a control or object in the Members list which is an object in itself, the description of the property includes a reference to the object type of the property. For example, Figure 3-5 shows the Knob control (CWKnob) selected in the Classes list and its `Axis` property selected in the Members list.

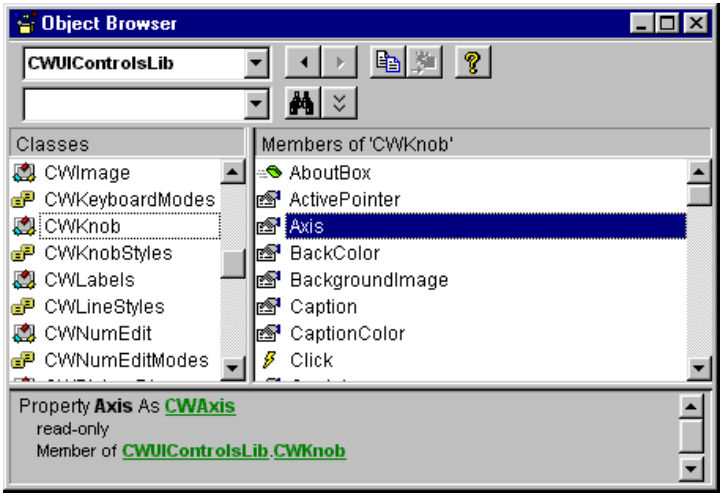

Figure 3-5. Viewing CWKnob in the Object Browser

The Axis on the Knob control is a separate object, so the description at the bottom of the dialog window lists the Axis property as CWAxis. CWAxis is the type name of the Axis object, and you can select CWAxis in the Classes list to see its properties and methods. Move from one level of the object hierarchy to the next level using the Object Browser to explore the structure of different controls.

The question mark (**?**) button at the top of the Object Browser opens the help file to a description of the currently selected item. To find more information about the CWGraph control, select the control in the window and press the **?** button.

Pasting Code into Your Program

If you open the Object Browser from the Visual Basic code editor, you can copy the name or prototype of a selected property, method, or function to the clipboard and then paste it into your program. To perform this task, select the desired Member item in the Object Browser. Press the **Copy to Clipboard** button at the top of the Object Browser or highlight the prototype at the bottom and press <Ctrl-C> to copy it to the clipboard. Paste

it into your code window by selecting **Paste** from the **Edit** menu or pressing <Ctrl-V>.

Use this method repeatedly to build a more complex reference to a property of a lower-level object in the object hierarchy. For example, you can create a reference to

```
CWGraph1.Axes.Item(1).ValuePairs.Item(3).Name
```

by typing in the name of the control (`CWGraph1`) and then using the Object Browser to add each section of the property reference. Refer to the *Item Method* section of Chapter 1, *Introduction to ComponentWorks*, for more information about the Item method and collections.

Adding Code Using Visual Basic Code Completion

Visual Basic 5 supports automatic code completion in the code editor. As you enter the name of a control, the code editor prompts you with the names of all appropriate properties and methods. Try placing a control on the form and then entering its name in the code editor. After typing the name, add a period as the delimiter to the property or method of the control. As soon as you type the period, Visual Basic drops down a menu of available properties and methods, as shown in Figure 3-6.

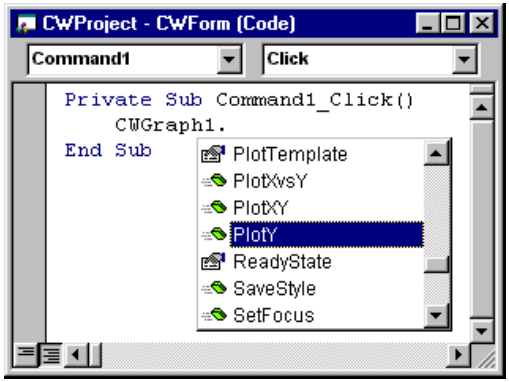

Figure 3-6. Visual Basic 5 Code Completion

You can select from the list or properties and events by scrolling through the list and selecting one or by typing in the first few letters of the desired item. Once you have selected the correct item, type the next logical character such as a period, space, equal sign, or carriage return to enter the selected item in your code and continue editing the code.

Learning to Use Specific ComponentWorks Controls

Each ComponentWorks control and its use are described in more detail in later chapters in this manual. However, these chapters do not discuss every property, method, and feature of every control. The ComponentWorks online reference contains detailed information about each control and all its associated properties, events, and methods. Refer to the online reference to find descriptions of the different features of a particular control. Remember that many of the ComponentWorks controls share properties. When you learn how to use one control, you are learning how to use others as well.

4

Building ComponentWorks Applications with Visual C++

This chapter describes how you can use ComponentWorks controls with Visual C++, explains how to insert the controls into the Visual C++ environment and create the necessary wrapper classes, shows you how to create an application compatible with the ComponentWorks controls using the Microsoft Foundation Classes Application Wizard (MFC AppWizard) and how to build your program using the ClassWizard with the controls, and discusses how to perform these operations using ActiveX controls in general.

At this point you should be familiar with the general structure of ActiveX controls described in Chapter 1, *Introduction to ComponentWorks*, as well as C++ programming and the Visual C++ environment. The individual ComponentWorks controls are described later in this manual.

Note ***The descriptions and figures in this chapter apply specifically to the Visual C++ 5 environment. Although most of this information applies to Visual C++ 4.x as well, some menu and option names and other environment components are different. See Appendix A, Using Previous Versions of Visual Basic, Visual C++, and Delphi with ComponentWorks, for a description of key differences between these versions of Visual C++ as they apply to ComponentWorks.***

Developing Visual C++ Applications

The following procedure explains how you can start developing Visual C++ applications with ComponentWorks.

1. Create a new workspace or project in Visual C++.
2. To create a project compatible with the ComponentWorks ActiveX controls, use the Visual C++ MFC AppWizard to create a skeleton project and program.
3. After building the skeleton project, add the ActiveX controls you need to the controls toolbar. From the toolbar, you can add the controls to the application itself.

4. After adding a control to your application, configure its properties using its property pages.
5. While developing your program code, use the control properties and methods and create event handlers to process different events generated by the control.

   Create the necessary code for these different operations using the ClassWizard in the Visual C++ environment.

Creating Your Application

When developing new applications, use the MFC AppWizard to create new project workspace so the project is compatible with ActiveX controls. The MFC AppWizard creates the project skeleton and adds the necessary code that enables you to add ActiveX controls to your program.

1. Create a new project by selecting **New...** from the **File** menu. The **New** dialog box opens (see Figure 4-1).

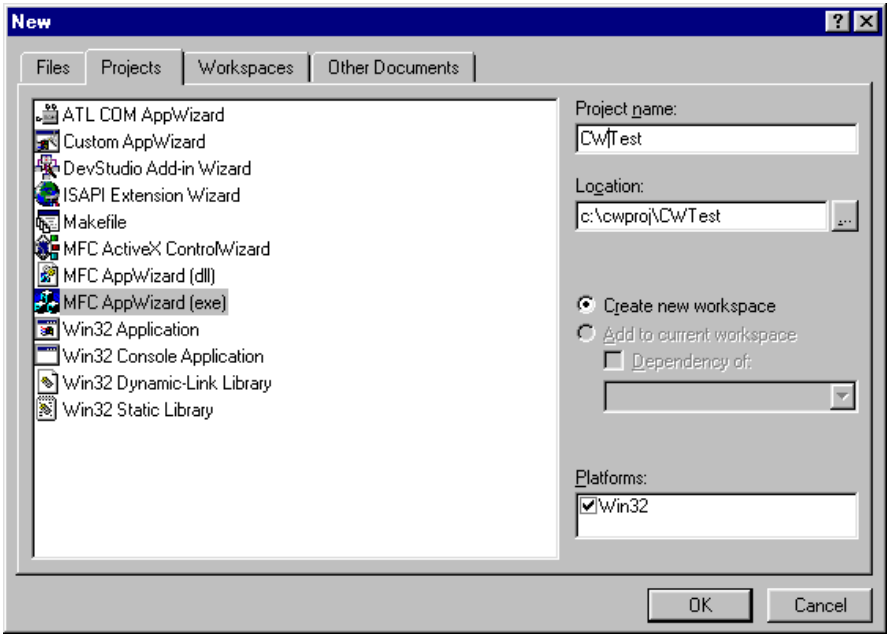

Figure 4-1. New Dialog Box

2. On the **Projects** tab, select the MFC AppWizard (exe) and enter the project name in the Project name field and the directory in the Location field.
3. Click on **OK** to setup your project.

Complete the next series of dialog windows in which the MFC AppWizard prompts you for different project options. If you are a new Visual C++ or the MFC AppWizard user, accept the default options unless otherwise stated in this documentation.

4. In the first step, select the type of application you want to build. For this example, select a Dialog based application, as shown in Figure 4-2, to make it easier to become familiar with the ComponentWorks controls.

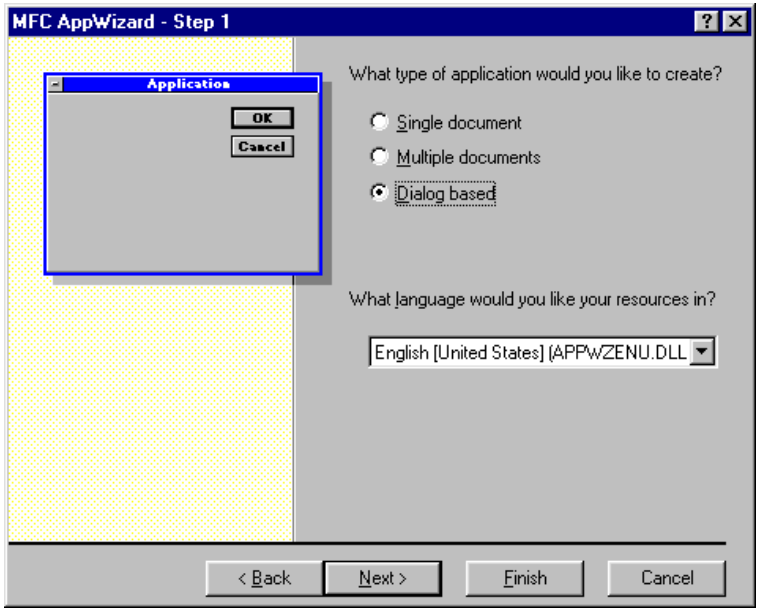

Figure 4-2. MFC AppWizard—Step 1

5. Click on the **Next>** button to continue.
6. Enable ActiveX controls support. If you have selected a Dialog based application, step two of the MFC AppWizard enables **ActiveX Controls** support by default.
7. Continue selecting desired options through the remainder of the MFC AppWizard. When you finish the MFC AppWizard, it builds a project and program skeleton according to the options you specified. The skeleton includes several classes, resources, and files, all of which can be accessed from the Visual C++ development environment.
8. Use the Workspace window, which you can select from the **View** menu, to see the different components in your project.

Adding ComponentWorks Controls to the Visual C++ Controls Toolbar

Before building an application using the ComponentWorks controls, you must load the controls into the Controls toolbar in Visual C++ from the Component Gallery in the Visual C++ environment. When you load the controls using the Component Gallery, a set of C++ wrapper classes automatically generate in your project. You must have wrapper classes to work with the ComponentWorks controls.

The Controls toolbar is visible in the Visual C++ environment only when the Visual C++ dialog editor is active. Use the following procedure to open the dialog editor.

1. Open the Workspace window by selecting **Workspace** from the **View** menu.
2. Select the Resource View (second tab along the bottom of the Workspace window).
3. Expand the resource tree and double click on one of the Dialog entries.
4. If necessary, right click on any existing toolbar and enable the Controls option.

By adding controls to your project, you create the necessary wrapper classes for the control in your project and add the control to the toolbox. Use the following procedure to add new controls to the toolbar.

1. Select **Project»Add To Project...»Components and Controls** and, in the following dialog, double click on Registered ActiveX Controls.
2. Select and insert registered ActiveX controls into your project and control toolbox.
3. Select the controls you need and click the **Insert** button. All ComponentWorks controls start with CW.
4. Click on **OK** in the following dialog windows.
5. When you have inserted all controls, click **Close** in the Components and Controls Gallery.

Building the User Interface Using ComponentWorks Controls

After adding the controls to the Controls toolbar, use the controls in the design of the application user interface. Place the controls on the dialog form using the dialog editor. You can size and move individual controls in the form to customize the interface. Use the custom property sheets to configure control representation on the user interface and control behavior at run time.

To add ComponentWorks controls to the form, open the dialog editor by selecting the dialog form from the Resource View of the Workspace window. If the Controls toolbar is not displayed in the dialog editor, open it by right clicking on any existing toolbar and enabling the Controls option.

To place a ComponentWorks control on the dialog form, select the desired control in the Controls toolbar and click and drag the mouse on the form to create the control. After placing the controls, move and resize them on the form as needed.

After you add a ComponentWorks control to a dialog form, configure the default properties of the control by right clicking the control and selecting **Properties** to display its custom property sheets. Figure 4-3 shows the CWGraph control property sheets.

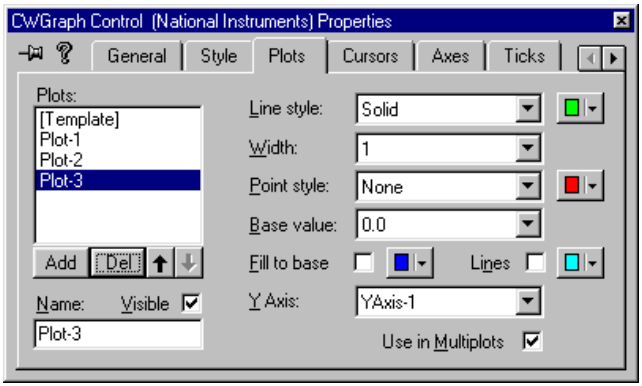

Figure 4-3. CWGraph Control Property Sheets

So you can see immediately how different properties affect the control, a separate window displays a sample copy of the control that reflects the property changes as you make them in the property sheets.

Programming with the ComponentWorks Controls

To program with ComponentWorks controls, use the properties, methods, and events of the controls as defined by the wrapper classes in Visual C++.

Note ***Later chapters in this manual provide more information on the most commonly used properties, methods, and events of the individual controls. All the properties, methods, and events of the different controls are described in detail in the ComponentWorks online reference, which you can access from* Programs»National Instruments ComponentWorks»ComponentWorks Reference *from the Windows* Start *menu.***

Before you can use the properties or methods of a control in your Visual C++ program, assign a member variable name to the control. This member variable becomes a variable of the application dialog class in your project.

To create a member variable for a control on the dialog form, right click on the control and select **ClassWizard**. In the **MFC Class Wizard** window, activate the **Member Variables** tab, as shown in Figure 4-4.

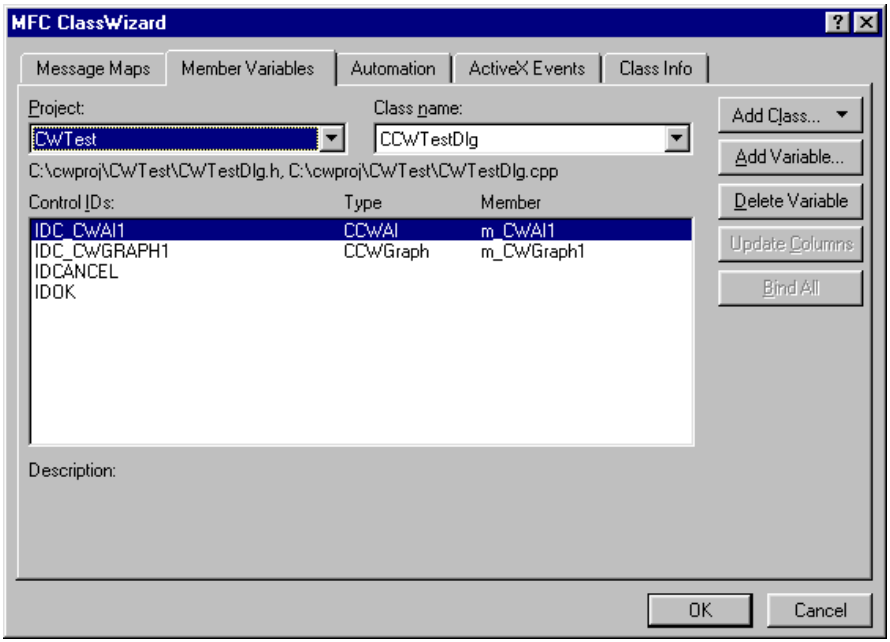

Figure 4-4. MFC ClassWizard—Member Variable Tab

Select the new control in the Control IDs field and press the **Add Variable...** button. In the dialog window that appears, complete the member variable name and press **OK**. Most member variable names start with `m_`, and you should adhere to this convention. After you create the member variable, use it to access a control from your source code. Figure 4-4 shows the MFC Class Wizard after member variables have been added for a graph and analog input control.

Using Properties

Unlike Visual Basic, you do not read or set the properties of ComponentWorks controls directly in Visual C++. Instead, the wrapper class of each control contains functions to read and write the value of each property. These functions are named starting with either `Get` or `Set`

followed by the name of the property. For example, to set the `Value` property of a slide, use the `SetValue` function of the wrapper class for the Slide control. In the source code, the function call is preceded by the member variable name of the control to which it applies.

```
m_Slide.SetValue(COleVariant(5.0));
```

All values passed to properties need to be of variant type. Convert the value passed to the Value property to a variant using `COleVariant()`.

Use the `GetValue()` function to read the value of a control or to pass a value of a control to another part of your program. For example, pass the value of a Slide control to a Meter control.

```
m_Meter.SetValue(m_Slide.GetValue());
```

Because the `GetValue` function returns its value as a variant in the previous line of code, conversion to a variant type is not necessary.

You can view the names of all the property functions (and other functions) for a given control in the ClassView of the Workspace window. In the Workspace window, select **ClassView** and then the control for which you want to view property functions and methods. Figure 4-5 shows the functions for the Slide object as listed in the Workspace. These are created automatically when you add a control to the Controls toolbar in you project.

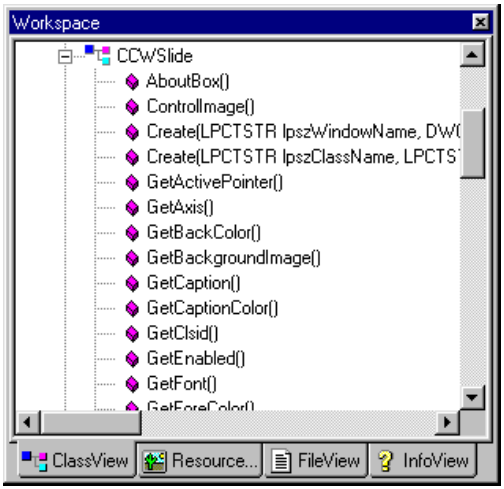

Figure 4-5. Viewing Property Functions and Methods in the Workspace Window

If you need to access a property of a control which is in itself another object, use the appropriate property function to return the sub-object of the control. Make a call to access the property of the sub-object. Include the header file in your program for any new objects. For example, use the following code to configure the Axis object of a Slide control.

```
#include cwaxis.h
CCWAxis Axis1;
Axis1 = m_Slide.GetAxis();
Axis1.SetMaximum(COleVariant(5.0));
```

You can chain this operation into one function call without having to declare another variable.

```
#include cwaxis.h
m_Slide.GetAxis().SetMaximum (COleVariant(5.0));
```

If you need to access an object in a collection property, use the `Item` method with the index of the object. Remember to include the header file for the collection object. For example, to set the maximum of the first y-axis on a graph, use the following code.

```
#include cwaxes.h
#include cwaxis.h
m_Graph.GetAxes().Item(COleVariant(2.0)).SetMaximum
  (COleVariant(5.0));
```

Using Methods

Use the control wrapper classes to extract all methods of the control. To call a method, append the method name to the member variable name and pass the appropriate parameters. If the method does not require parameters, use a pair of empty parentheses.

```
m_CWAI1.Start();
```

Most methods take some parameters as variants. You must convert any such parameter to a variant if you have not already done so. You can convert most scalar values to variants with `COleVariant()`. For example, the `PlotY` method of the graph control requires three scalar values as variants.

```
m_Graph.PlotY (*Voltages, COleVariant(0.0),
  COleVariant(1.0), COleVariant(1.0));
```

☞ **Note** ***Consult Visual C++ documentation for more information about variant data types.***

If you need to call a method on a sub-object of a control, follow the conventions outlined in the *Using Properties* section earlier in this chapter. For example, a single plot on a graph is an object in the Plots collection, which is an object itself in the graph control. To call `PlotY` on one particular plot of your graph, use the following line of code.

```
m_Graph.GetPlots().Item(COleVariant(2.0)).PlotY
  (*Voltages, COleVariant(0.0), COleVariant(1.0));
```

Using Events

After placing a control on your form, you can start defining event handler functions for the control in your code. Events generate automatically at run time when different controls respond to conditions, such as a user clicking a button on the form or the data acquisition process acquiring a specified number of points.

Use the following procedure to create an event handler.

1. Right click on a control and select **ClassWizard**.
2. Select the **Message Maps** tab and the desired control in the Object IDs field. The Messages field displays the available events for the selected control. (See Figure 4-6, *Event Handler for the PointerValueChanged Event of a Knob*).
3. Select the event and press the **Add Function...** button to add the event handler to your code.
4. To switch directly to the source code for the event handler, click on the **Edit Code** button. The cursor appears in the event handler, and you can add the functions to call when the event occurs. You can use the **Edit Code** button at any time by opening the class wizard and selecting the event for the specific control.

The following figure is an example of an event handler generated for the `PointerValueChanged` event of a knob. Insert your own code in the event handler:

```
void CTestDlg::OnPointerValueChangedCwknob1(long
Pointer, VARIANT FAR* Value)
{
// TODO: Add your control notification handler code here
}
```

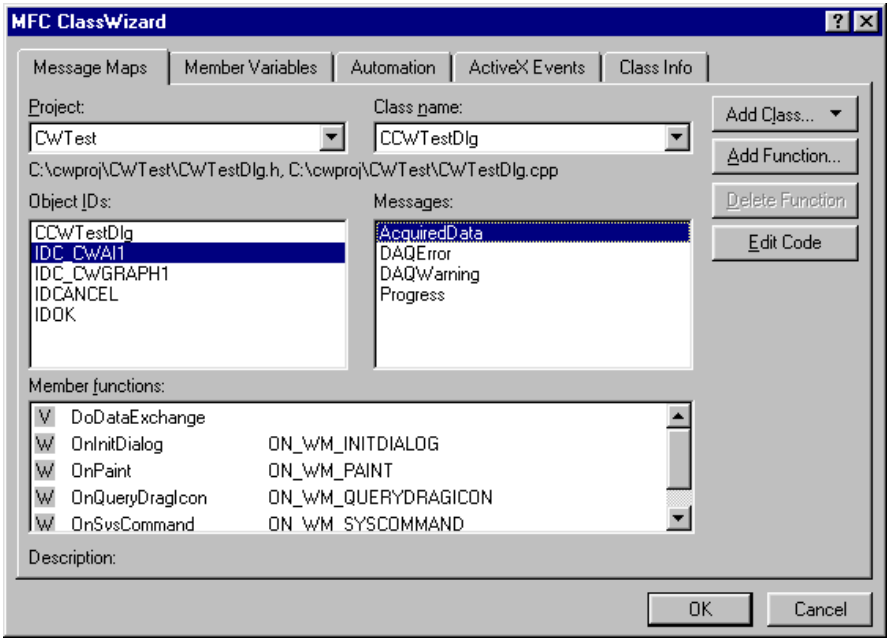

Figure 4-6. Event Handler for the PointerValueChanged Event of a Knob

Learning to Use Specific ComponentWorks Controls

Each ComponentWorks control and its use are described in more detail in later chapters in this manual. However, these chapters do not discuss every property, method, and feature of every control. The ComponentWorks online reference contains detailed information about each control and all its associated properties, events, and methods. Refer to the online reference to find descriptions of the different features of a particular control. Remember that many of the ComponentWorks controls share properties and sub-objects. When you learn how to use one control, you are learning how to use others.

5

Building ComponentWorks Applications with Delphi

This chapter describes how you can use ComponentWorks controls with Delphi; insert the controls into the Delphi environment, set their properties, and use their methods and events; and perform these operations using ActiveX controls. This chapter also outlines Delphi features that simplify working with ActiveX controls.

At this point you should be familiar with the general structure of ActiveX controls described in Chapter 1, *Introduction to ComponentWorks*. The individual ComponentWorks controls are described later in this manual.

Note ***The descriptions and figures in this chapter apply specifically to the Delphi 3 environment. Although most of this information applies to Delphi 2 as well, some menu and option names and other environment components are different. See Appendix A, Using Previous Versions of Visual Basic, Visual C++, and Delphi with ComponentWorks, for a description of key differences between these versions of Delphi as they apply to ComponentWorks.***

If you have the original release of Delphi 3, you might experience significant problems with ActiveX controls, but Borland offers a newer version of Delphi that corrects most of these problems. Before using ComponentWorks with Delphi 3, contact Borland to receive the Delphi 3 patch or a newer version.

Running Delphi Examples

To run the Delphi examples installed with ComponentWorks, you need to import the appropriate controls into the Delphi environment. See the section on *Loading the ComponentWorks Controls into the Component Palette* for more information about loading the controls.

Upgrading from a Previous Version of ComponentWorks

When you upgrade ComponentWorks, you must remove the current controls from the Delphi environment and reinsert the controls in the Delphi environment to update the support files.

1. From the **Component** menu select **Install Packages...**.
2. In the Design packages list, select **Delphi User's Components**.
3. Click on **Edit...** and **Yes** in the following dialog boxes to edit the user's component package. The package editor lists all the components currently installed in the user's components package, including the ComponentWorks controls.
4. Select each of the ComponentWorks entries and click on **Remove**.
5. Click on **Compile** to rebuild the package.
6. Close the package editor.

Developing Delphi Applications

You start developing applications in Delphi using a *form*. A form is a window or area on the screen on which you can place controls and indicators to create the user interface for your programs. The Component palette in Delphi contains all of the controls available for building applications. After placing each control on the form, configure the properties of the control with the default and custom property pages. Each control you place on a form has associated code (event handler routines) in the Delphi program that automatically executes when the user operates the control or the control generates an event.

Loading the ComponentWorks Controls into the Component Palette

Before you can use the ComponentWorks controls in your Delphi applications, you must add them to the Component palette in the Delphi environment. You need to add the controls to the palette only once because the controls remain in the Component palette until you explicitly remove them. When you add controls to the palette, you create Pascal import units (header files) that declare the properties, methods, and events of a control. When you use a control on a form, a reference to the corresponding import unit is automatically added to the program.

Note ***Before adding a new control to the Component palette, make sure to save all your work in Delphi, including files and projects. After loading the controls, Delphi closes any open projects and files to complete the loading process.***

Use the following procedure to add ActiveX controls to the Component palette.

1. Select **Import ActiveX Control...** from the **Component** menu in the Delphi environment. The Import ActiveX Control window displays a list of currently registered controls.

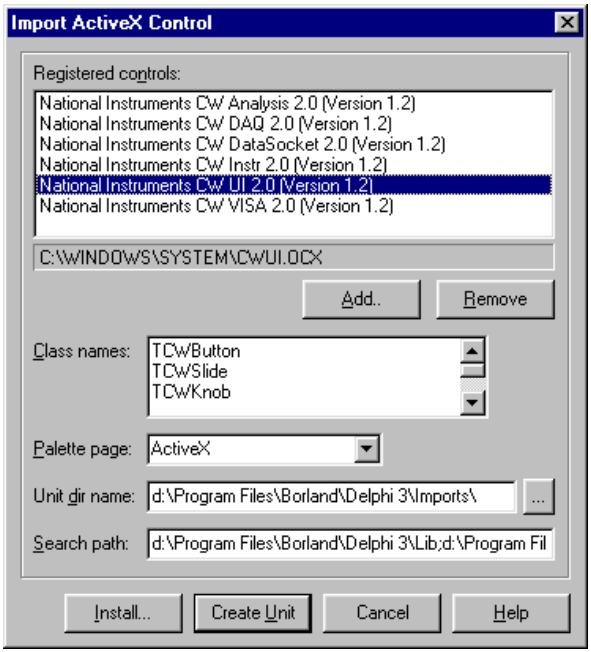

Figure 5-1. Delphi Import ActiveX Control Dialog Box

2. Select the control group you want to add to the Component palette. All ComponentWorks controls start with `National Instruments`.
3. After selecting the control group, click **Install...**.

   Delphi generates a Pascal import unit file for the selected `.OCX` file, which is stored in the Delphi `\Imports` directory. If you have installed the same `.OCX` file previously, Delphi prompts you to overwrite the existing import unit file.

4. In the **Install** dialog box, click on **OK** to add the controls to the Delphi user's components package.
5. In the following dialog, click on **Yes** to rebuild the user's components package with the added controls. Another dialog box acknowledges the changes you have made to the user's components package, and the package editor displays the components currently installed.

   At this point, you can add additional ActiveX controls with the following procedure.

   a. Click on the **Add** button.
   b. Select the **Import ActiveX** tab.
   c. Select the ActiveX control you want to add.
   d. Click on **OK**.
   e. After adding the ActiveX controls, compile the user's components package.

If your control does not appear in the list of registered controls, click the **Add...** button. To register a control with the operating system and add it to the list of registered controls, browse to and select the OCX file that contains the control. Most OCX files reside in the `\Windows\System(32)` directory.

New controls are added to the **ActiveX** tab in the Components palette. You can rearrange the controls or add a new tab to the Components palette by right clicking on the palette and selecting **Properties...**.

Building the User Interface

After you add the ComponentWorks controls to the Component palette, use them to create the user interface. Open a new project, and place different controls on the form. These controls, as part of the program user interface, add specific functionality to the application. After placing the controls on the form, configure their default property values through the stock and custom property sheets.

Placing Controls

To place a control on the form, select the control from the Component palette and click and drag the mouse on the form. Use the mouse to move and resize controls to customize the interface, as in the following figure. After you place the controls, you can change their default property values by using the default property sheet (Object Inspector) and custom property sheets.

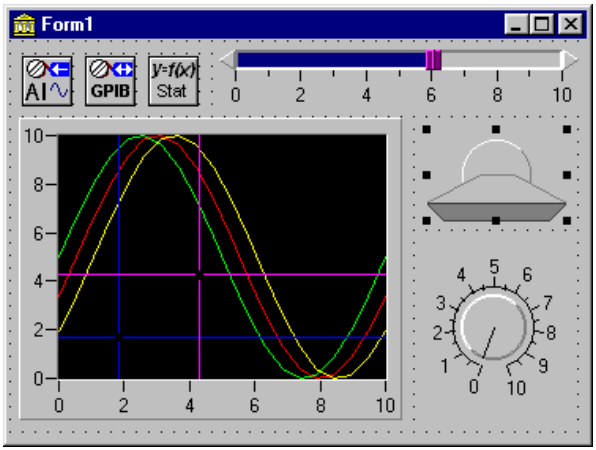

Figure 5-2. ComponentWorks Controls on a Delphi Form

Using Property Sheets

Set property values such as `Name` in the Object Inspector of Delphi. To open the Object Inspector, select **Object Inspector** from the **View** menu or press <F11>. Under the **Properties** tab of the Object Inspector, you can set different properties of the selected control.

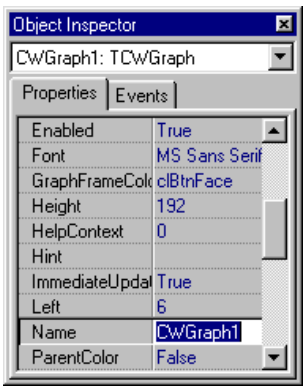

Figure 5-3. Delphi Object Inspector

To open the custom property pages of a control, double click on the control or right click on the control and select **Properties...**. You can edit most

control properties from the custom property pages. The following figure shows the ComponentWorks Graph control property page.

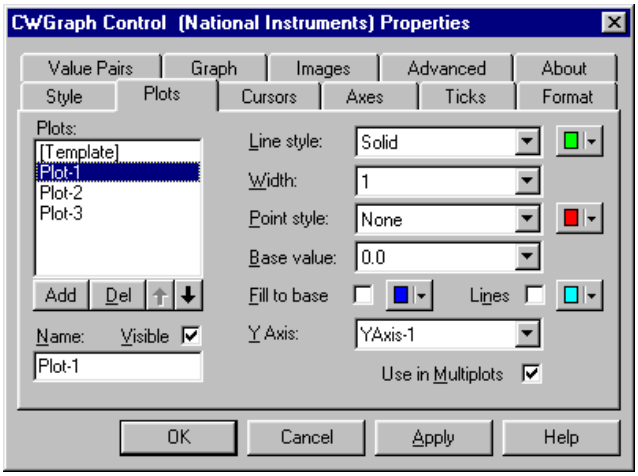

Figure 5-4. ComponentWorks Graph Control Property Page

For more information about specific control properties, see the ComponentWorks online reference.

Programming with ComponentWorks

The code for each form in Delphi is listed in the Associated Unit (code) window. You can toggle between the form and Associated Unit window by pressing <F12>. After placing controls on the form, use their methods in your code and create event handler routines to process events generated by the controls at run time.

Using Your Program to Edit Properties

You can set or read control properties programmatically by referencing the name of the control with the name of the property, as you would any variable name in Delphi. The name of the control is set in the Object Inspector.

If you want to change the state of an LED control during program execution, change the `Value` property from `True` to `False` or from `False` to `True`. The syntax for setting the `Value` property in Delphi is `name.property: = new_value`.

```
CWButton1.Value := True;
```

A property can be an object itself that has its own properties. To set properties in this case, combine the name of the control, sub-object, and property. For example, consider the following code for the DAQ CWAI control. `ScanClock` is both a property of the DAQ control and an object itself. `Frequency` is a property of the `ScanClock` object. As an object of the DAQ control, `ScanClock` itself has several additional properties.

```
CWAI1.ScanClock.Frequency := 10000;
```

You can retrieve the value of a control property from your program in the same way. For example, you can assign the scan rate of a CWAI control to a text box on the user interface.

```
Edit1.Text := CWAI1.ScanClock.Frequency;
```

To use the properties or methods of an object in a collection, use the `Item` method to extract the object from the collection. Once you extract the object, use its properties and methods as you usually would.

```
CWGraph1.Axes.Item(2).Maximum := 5;
```

In some cases, an object can be assigned as a property to another object. The following code assigns a Plot object of a graph to a Cursor object in order to specify the plot the cursor is tracking.

```
CWGraph1.Cursors.Item(1).Plot := CWGraph1.Plots.Item(2);
```

Consult the *Setting the Properties of an ActiveX Control* section in Chapter 1, *Introduction to ComponentWorks*, for more information about setting properties programmatically.

Using Methods

Each control has defined methods that you can use in your program. To call a method in your program, use the control name followed by the method name.

```
CWAI1.Start;
```

Some methods require parameters, as does the following method.

```
CWGraph1.PlotY (data, 0.0, 1.0, True);
```

In most cases, parameters passed to a method are of type variant. Simple scalar values can be automatically converted to variants and, therefore, might be passed to methods. Arrays, however, must be explicitly declared as variant arrays.

The following example plots data using the graph `PlotY` method. Consult your Delphi documentation for more information about the variant data type.

```
var
     vData:Variant;
begin
     // Create array in Variant
     vData := VarArrayCreate([0, 99], varDouble);
     for i := 0 to 99 do
     begin
          vData[i] := Random;
     end;
     // Plot Variant Array
     CWGraph1.PlotY (vData, 0.0, 1.0, True);
end;
```

Using Events

Use event handler routines in your source code to respond to and process events generated by the different ComponentWorks controls. Events are generated by user interaction with an object such as a knob or by other controls (such as the DAQ controls) in response to internal conditions (for example, completed acquisition or an error). You can create a skeleton for an event handler routine using the Object Inspector in the Delphi environment.

To open the Object Inspector, press <F11> or select **Object Inspector** from the **View** menu. In the Object Inspector, select the **Events** tab. This tab, as shown in the following figure, lists all the events for the selected control. To create a skeleton function in your code window, double click on the empty field next to the event name. Delphi generates the event handler routine in the code window using the default name for the event handler.

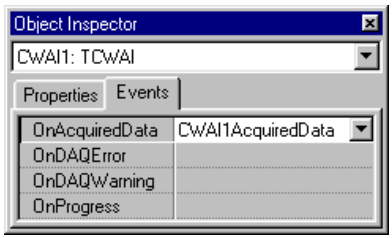

Figure 5-5. Delphi Object Inspector Events Tab

To specify your own event handler name, click in the empty field in the Object Inspector next to the event, and enter the function name. After the event handler function is created, insert the code in the event handler.

Learning to Use Specific ComponentWorks Controls

Each ComponentWorks control and its use is described in more detail in later chapters in this manual. However, these chapters do not discuss every property, method, and feature of every control. The ComponentWorks online reference contains detailed information about each control and all its associated properties, methods, and events. Refer to the online reference to find descriptions of the different features of a particular control. Remember that many of the ComponentWorks controls share properties. When you learn how to use one control, you are learning how to use others.

6

Using the User Interface Controls

This chapter describes how you can use the ComponentWorks User Interface (UI) controls to customize your application interface; explains the individual controls and their most commonly used properties, methods, and events; and includes tutorial exercises that give step-by-step instructions on using the controls in simple programs.

The UI controls include features commonly used in instrumentation and data acquisition. You can use them to create the front end for virtually any type of application, including finance, systems management, and many others.

In this chapter, all examples are presented in Visual Basic. Consult the appropriate chapter in this manual for information on using the ComponentWorks controls in another environment. The software includes solutions for the tutorials in Visual Basic, Visual C++, and Delphi. You can find additional information in the online reference, available by selecting **Programs»National Instruments ComponentWorks» ComponentWorks Reference** from the Windows **Start** menu.

What Are the UI Controls?

ComponentWorks includes five separate ActiveX controls with instrumentation-style interface objects. Each of the ActiveX controls represents a family of individual control styles. Table 6-1 lists the controls and their associated styles. You can set the individual control style from the property pages during design or through properties and methods at run time.

Table 6-1. User Interface Control Styles

Control	Control Style
CWKnob	Knob Dial Horizontal and Vertical Meter
CWSlide	Horizontal and Vertical Slide Horizontal and Vertical Fill ThermometerTank
CWNumEdit	Numeric Edit Box
CWButton	Slide Switch Toggle Switch Push Button Command Button Custom Bitmap Button LED
CWGraph	Graph Strip Chart Scope Chart

Object Hierarchy and Common Objects

Most of the ComponentWorks User Interface controls are made up of a hierarchy of less complex objects. Understanding the relationship among the objects in a control is the key to properly programming with the control. Dividing a control into individual objects makes it easier to work with because each individual component has fewer parts.

The Knob and Slide Controls

The Knob and Slide controls are similar to each other. The Knob control represents different types of circular displays, such as a knob, gauge, or different types of meters. The Slide control represents different types of linear displays, such as thermometers and tank displays. With the Knob and Slide controls, users can input or output (display) individual or multiple scalar values. A knob or slide can have multiple pointers on the control, each pointer representing one scalar value.

Like other controls, the Knob and Slide are made up of a hierarchy of objects, as shown in Figure 6-1.

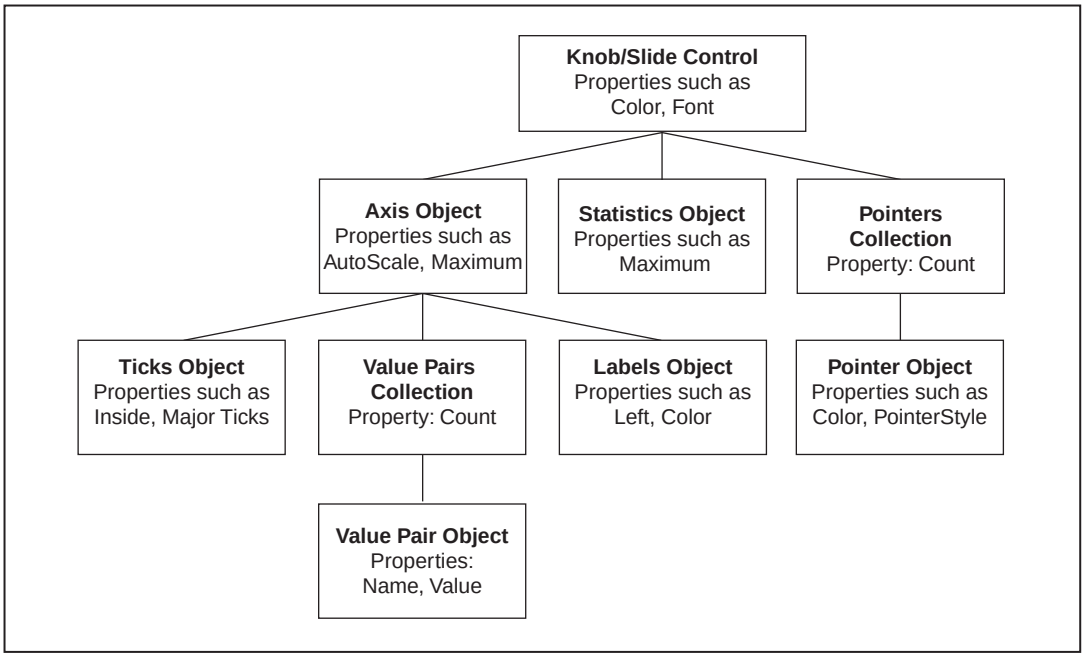

Figure 6-1. Knob/Slide Control Object Hierarchy

Knob and Slide Object

The Knob and Slide objects maintain the basic attributes of the control, such as background color and the caption. However, its most important property is `Value`, which contains the value of the currently active pointer. Although the control might have more than one pointer value (each stored in an individual Pointer object), the `Value` property can hold only the value of the selected pointer. You can select a pointer with the `ActivePointer`

property or with your mouse. You access the `Value` property with the following code.

```
CWKnob1.Value = 5.0
x = CWSlide1.Value
```

Pointers Collection

The Pointers collection contains the individual Pointer objects of the knob or slide object. It has one read-only property, `Count`, which returns the number of Pointer objects in the collection.

```
NumPointers = CWSlide1.Pointers.Count
```

Like all collections, the Pointers collection also has an `Item` method that you use to access a specific pointer in the collection. To retrieve a pointer, call the `Item` method and specify the (one-based) index of the pointer in the collection.

```
CWKnob1.Pointers.Item(2)
```

Because each pointer also has a name property, you can retrieve an individual pointer with its name rather than the index.

```
CWSlide1.Pointers.Item("BoilerPressure")
```

Pointer Object

The Pointer object, which is stored in the Pointers collection, represents a single value displayed on either a knob or a slide control. The Pointer object contains properties, such as `Style` and `FillStyle`, that affect the graphical representation of the pointer. Usually, these properties are set through the property pages during design, and they do not change during program execution. If the pointer is not currently active, use its `Value` property to read or set its value.

```
MaxLimit = CWKnob1.Pointers.Item(3).Value
CWSlide1.Pointers.Item("BoilerPressure").Value =
   AcquiredPressure
```

Axis Object

The Axis object contains information about the axis scale used for a knob (circular scale) or a slide (straight scale) and properties, such as `AutoScale`, `Maximum`, and `Minimum`, that you can set and read directly.

```
CWKnob1.Axis.AutoScale = True
```

```
MaxValue = CWKnob1.Axis.Maximum
```

Use the `SetMinMax` method to specify a new minimum and a new maximum for the axis in one function call.

```
CWSlide1.Axis.SetMinMax newMin, newMax
```

The Axis object contains three objects—`Ticks` object, `Labels` object, and the `ValuePairs` collection—described in the following sections.

Ticks Object

Use the Ticks object to specify how tick marks appear on a particular axis. You can set properties to specify the spacing between ticks as well as major and minor tick selection. The Tick object also controls any grid displayed for a particular axis on the graph. Usually, Tick properties are set during design though the property pages. If necessary, you can change them at run time with simple property calls.

```
CWSlide1.Axis.Ticks.AutoDivision = False
```

```
CWKnob1.Axis.Ticks.MinorUnitsInterval = 2.0
```

```
CWGraph1.Axes.Item(1).Ticks.MajorGrid = True
```

Labels Object

The Labels object determines how axis labels are drawn. Labels are the numbers displayed next to the ticks. The Label object properties specify where to draw the labels (`right`, `left`, `above`, or `below`) and the color of the labels.

```
CWSlide1.Axis.Labels.Color = vbBlue
```

```
CWKnob1.Axis.Lables.Radial = True
```

```
CWGraph1.Axes.Item(1).Labels.Above = True
```

ValuePairs Collection

Use the ValuePairs collection and ValuePair objects to mark specific points on any axis with a custom label. The ValuePairs collection contains a variable number of ValuePair objects on an axis. The `Count` property, along with several other properties, define how value pairs appear on the axis.

```
NumMarkers = CWSlide1.Axis.ValuePairs.Count
```

```
CWKnob1.Axis.ValuePairs.LabelType = cwVPLabelName
```

The ValuePairs collection has an `Item` method, which you can use to access a specific ValuePair in the collection, and several other methods (`Add`, `Remove`, `RemoveAll`) to dynamically manipulate the collection. The

`RemoveAll` method deletes all objects in the collection, and the `Add` and `Remove` methods add or remove only one value pair at a time. Specify the index of the value pair to be deleted on the `Remove` method.

```
CWSlide1.Axis.ValuePairs.Item(2)
```

```
CWKnob1.Axis.ValuePairs.RemoveAll
```

```
CWGraph1.Axes.Item(2).ValuePairs.Remove 2
```

ValuePair Object

A value pair associates a symbolic name with a value and marks a specific point on an axis. You can specify whether the value pair's value or the value pair's index in the collection determines the position of the value pair on the axis and whether the graphical representation of the value pair on the axis is its name or value.

```
CWSlide1.Axis.ValuePairs.Add
n = CWSlide1.Axis.ValuePairs.Count
CWSlide1.Axis.ValuePairs.Item(n).Name = "Max"
CWSlide1.Axis.ValuePairs.Item(n).Value = 7.0
```

Statistics Object

The Statistics object provides access to statistical values stored by the Knob and Slide controls. The three calculated statistics—minimum, maximum, and mean—are updated each time a pointer value changes graphically or programmatically. Use the `Reset` method to reset the minimum, maximum, and mean values. After a reset, minimum and maximum values are calculated using only those values collected since the last reset. The mean is the average of the last ten values.

```
AverageMeasurement = CWSlide1.Statistics.Mean
```

```
CWKnob1.Statistics.Reset
```

Use the property pages or the Pointer `mode` property to continuously display any of the statistics values for a specific pointer.

Events

When the value of a pointer on the control changes from the user interface or the program, the `PointerValueChanged` event is fired. Usually, this event updates values in the application in response to changes on the user interface. For example, the following event handler uses a numeric edit box to digitally display the value of a slide.

```
Private Sub CWSlide1_PointerValueChanged(ByVal Pointer
  As Long, Value As Variant)
    NumEdit1.Value = CWSlide1.Value
End Sub
```

The `Pointer` specifies the index of the pointer that has a changed value. You also can use the `PointerValueCommitted` event to process changed values after the user moves the mouse off the control.

Consult the online reference for more information about individual properties, methods, or events.

The Numeric Edit Box Control

Use the Numeric Edit Box control to display numbers as you would display text in a text box. Because the control includes increment and decrement buttons, you can change the value of the control with a mouse or touch screen. With range checking, you can preset a valid range for the control so the application is notified if the value is set outside of the limits. The Numeric Edit Box control has no other objects in its hierarchy. All properties and methods are contained in the control itself.

`Value`, the most commonly used property, reads and sets the value of the Numeric Edit Box control.

```
CWNumEdit1.Value = 5.0
x = CWNumEdit1.Value
```

With the `Minimum`, `Maximum`, and `RangeChecking` properties, you can configure the range checking process.

```
CWNumEdit1.Maximum = 5.0
CWNumEdit1.RangeChecking = True
```

Use the `SetMinMax` method to set the upper and lower limit of the range.

```
CWNumEdit1.SetMinMax -10,10
```

Events

The Numeric Edit Box control has three key events: `ValueChanged`, `ValueChanging`, and `IncDecButtonClicked`.

`ValueChanged` is fired every time the value of the control has been changed from the program or user interface.

`ValueChanging` is fired when the value of the control changes, before the new value is set in the control. The event returns parameters for the new value with range checking, the attempted value, and the previous value. `NewValue` is passed by reference so the code in the event routine can revise the value before it is set in the control. You only need to set `NewValue` if you want to change the value stored in the control.

```
Private Sub CWNumEdit1_ValueChanging(NewValue As
 Variant,ByVal AttemptedValue As Variant, ByVal
 PreviousValue As Variant,ByVal OutOfRange As Boolean)
   If NewValue > 100 Then NewValue = NewValue + 10
End Sub
```

`IncDecButtonClicked` is fired when a user presses either the increment or decrement button on the numeric edit box. The event returns a Boolean parameter indicating which button was pressed.

Consult the online reference for more information about individual properties, methods, or events.

Tutorial: Knob, Slide, and Numeric Edit Box Controls

This tutorial shows you how to use the Knob, Slide, and Numeric Edit Box controls in an application. Most often, these control are used to display information or to input simple data into an application. The tutorial lists the steps necessary to integrate the controls with the program.

The Knob and Slide controls each have several different display styles. For example, you can display a knob as a meter or dial, and the slide as a fill, thermometer, or tank. Although each style changes the display of the control, the programmatic functionality of the control remains constant. Continue to use property sheets, event functions, and the properties and methods the same way.

This tutorial uses Visual Basic syntax, but the discussion is in general terms so you can follow it in any compatible programming environment. Remember to adjust any code to your specific programming language. Refer to the *Building ComponentWorks Applications* chapters for information about implementing any step in different programming environments. You also can refer to the tutorial examples installed with ComponentWorks for a completed version of this example in several different programming environments.

Designing the Form

1. Open a new project and form. If you are working in Visual C++, select a dialog-based application and name your project `SimpleUI`.
2. Load the ComponentWorks user interface controls (specifically, the Numeric Edit Box, Knob, and Slide) into your programming environment.

3. From the toolbox or toolbar, place a CWKnob (knob) control on the form. Keep its default name, `CWKnob1`.

4. Place a CWSlide (slide) control on the form. Keep its default name, `CWSlide1`. Open its property page and select the `Vertical Fill` style. You also can change other properties, such as fill color.

5. Place a CWNumEdit (numerical edit box) control near the knob on the form. Keep its default name, `CWNumEdit1`. Keep its default property values.
6. Place another CWNumEdit (numerical edit box) control near the slide on the form. Change its name from `CWNumEdit2` to `CWSlideDisplay`. To change the name in Visual Basic, use the default property sheet (press <F4>). In Visual C++, open the custom property sheets. In Delphi, use the Object Inspector (press <F11>). Open its custom property sheet, under the **Style** tab, select the **Indicator** Control Mode, and unselect the **Visible** property of the Inc/Dec Button. You also can change other properties, such as the font used in the display.

Your form should look similar to the one shown below.

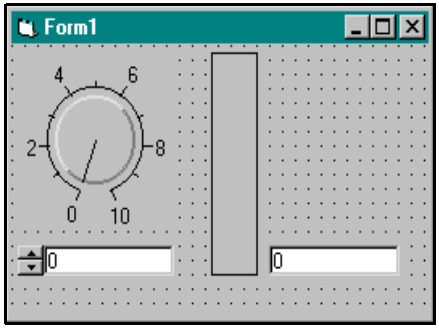

Figure 6-2. SimpleUI Form

Developing the Program Code

This program uses the numeric edit box next to the slide (without the increment or decrement arrows) to display the value of the slide control. The knob is used to change the value of the slide, and the other numeric edit box is used to change the value of the knob, thereby changing the value of the slide.

To have your program respond when the slide value changes, add the `PointerValueChanged` event for the slide. Use the `Value` property to retrieve or set the current value of the controls.

1. Create a skeleton event handler for the `PointerValueChanged` event of CWSlide1.
   - In Visual Basic, double click on the slide control on the form to create the `CWKnob1_PointerValueChanged` subroutine.
   - In Visual C++, use the MFC ClassWizard to create the event handler routine. Right click on the slide control and select **ClassWizard**.
   - In Delphi, use the Object Inspector to create the event handler routine. Select the slide control, press <F11> to open the Object Inspector, select the **Events** tab, and double click the empty field next to the `PointerValueChanged` event.
2. Add the following code inside the event handler routine. If you are working in Visual C++, first add a member variable for each control to the application dialog class.
   - Visual Basic:
     `CWSlideDisplay.Value = CWSlide1.Value`
   - Visual C++:
     `m_CWSlideDisplay.SetValue(m_CWSlide1.GetValue());`
   - Delphi:
     `CWSlideDisplay.Value:= CWSlide1.Value;`
3. Repeat step 1 for the knob control.
4. Add the following code to the `CWKnob1_PointerValueChanged` event routine, adjusting for your programming language:

   ```
   CWSlide1.Value = CWKnob1.Value
   ```
5. Repeat step 1 for the numeric edit box (`CWNumEdit1`) control.

6. Add the following to the `CWNumEdit1_PointerValueChanged` event routine, adjusting for your programming language:

   ```
   CWKnob1.Value = CWNumEdit1.Value
   ```

7. Save the project and associated files as `SimpleUI`.

Testing Your Program

Run the program. Notice that the slide display and associated numeric edit box change as you turn the knob. Notice that when you change the value of the other numeric edit box (with the increment or decrement arrows), both the knob and slide value change.

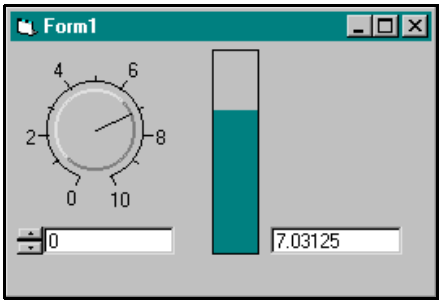

Figure 6-3. Testing SimpleUI

The program calls the `CWKnob1_PointerValueChanged` function and updates the slide control every time the value of the knob changes while the mouse button is pressed. Because the slide control has its own `PointerValueChanged` routine, the associated numeric edit box is always updated when the value of the slide control changes. Finally, when you change the value of the other numeric edit box, its `PointerValueChanged` routine updates the value of the knob, calling the `PointerValueChanged` routine of the knob, and so on.

To call the event handler routines after releasing the mouse button from the newly selected value, use the `PointerValueCommitted` or `MouseUp` event, rather than `PointerValueChanged`.

The Button Control

The Button control is a simple control you can use to input or output Boolean information or to initiate an action in your program. Because this control is simple, it is made of only one object. Like the Knob and Slide controls, the Button control has several different styles, including toggle switches, LEDs, push buttons, slides, on/off buttons and custom bitmap buttons.

The `Mode` property allows the button, regardless of its style, to act as a command button, switching state when you press the button with your mouse. Use this mode to initiate action in your program without changing the state of the button permanently.

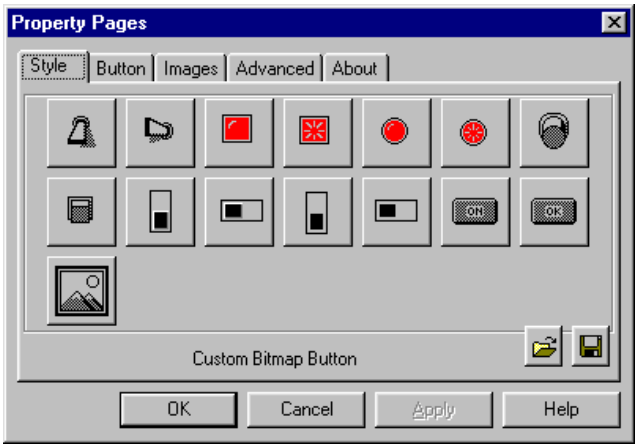

Figure 6-4. Button Control Modes

The most commonly used property on the button control is `Value`, which you can use to set the state of the control, such as for an LED, or to read the state of the control.

```
CWAlarmLED.Value = AlarmState
If (CWButton1.Value = True) Then...
```

Set other properties, such as `OnColor`, `OffColor`, `OnText` and `OffText`, in the property pages during development. In the property pages, you can select your own bitmaps to represent the on and off states of a custom Boolean control. For example, you can create representations of valves or heaters to depict industrial processes.

The Button control does not have any methods.

Events

The most important event generated by the Button control is `ValueChanged`, which notifies the application that the button value has changed. This event is generated if the button is in switch mode (switch value when clicked on) or in command mode (switch value until released).

```
Private Sub CWButton1_ValueChanged(ByVal Value As Boolean)
    'insert code to run when button is pressed
End Sub
```

The Graph Control

The Graph control is a flexible control used for plotting and charting data. It can display multiple traces and support multiple cursors and Y axes. *Plotting* data refers to the process of taking a large number of points and updating one or more plots on the graph with new data. The old plot is replaced with the new plot. *Charting* data appends new data points to an existing plot over time. Charting is used with slow processes where only few data points per second are added to the graph. When more data points are added than can be displayed on the graph, the graph scrolls so that new points are added to the right side of the graph while old points disappear to the left. You can use the same Graph control for both charting and plotting. Select between the two operations by using different methods for displaying the data.

The Graph control is made up of a hierarchy of objects, as illustrated in Figure 6-5, used to interact with the control programmatically. At design time, you can manipulate properties of the individual objects through the property pages.

The objects in the Graph control hierarchy represent the different parts displayed on the physical representation of the graph. The three main parts are the Axes collection and Axis objects, Plots collection and Plot objects, and Cursors collection and Cursor objects. Additionally, the PlotTemplate object serves as a template for new Plot objects created in the Plots collection.

The Graph object contains the basic properties of the control, such as name, graph frame color, plot area color, and track mode.

The Axes collection and Axis objects control the different axes on the graph. The graph contains one X axis and a varying number of Y axes, all of which are contained in the Axes collection.

The Cursors collection and Cursor objects control the cursors on the graph. Usually, cursors are created at design time with the property pages. You can use cursors to mark a specific point or region on the graph or highlight something programmatically.

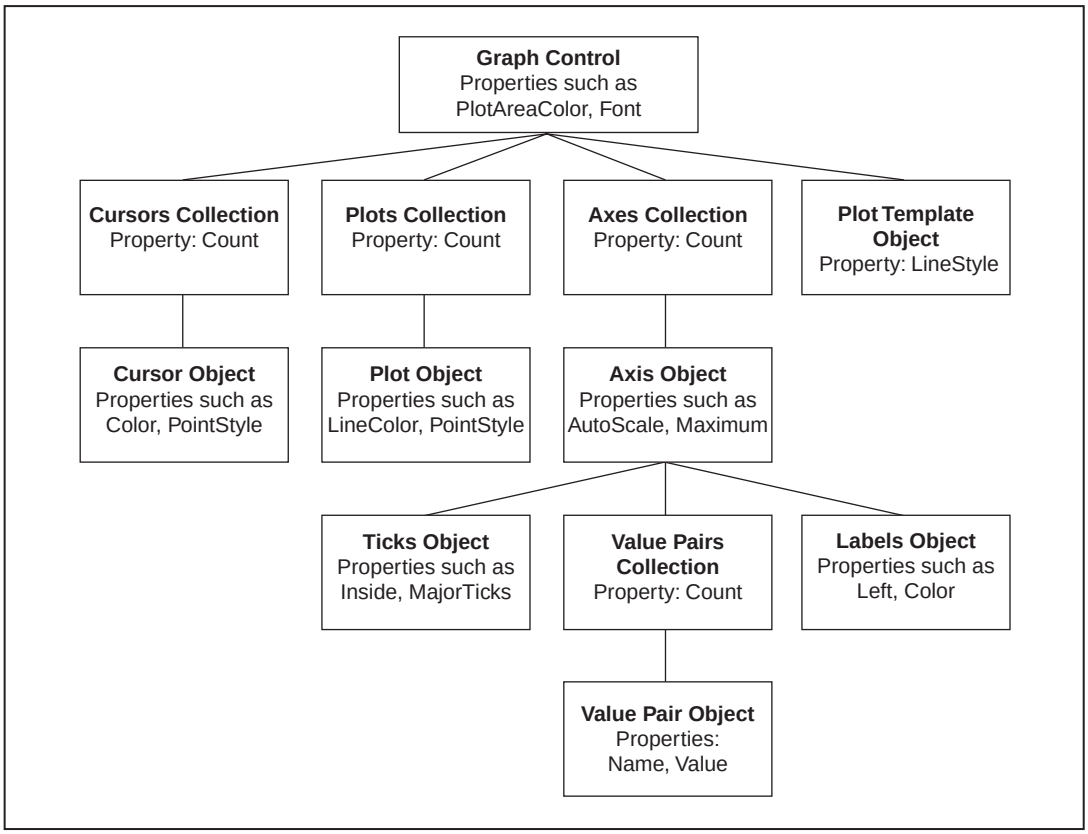

Figure 6-5. Graph Control Object Hierarchy

Graph Object

The Graph object has several simple properties, such as its name and colors, that are usually set in the property pages during design time. The Graph object also contains other properties that affect the behavior of the graph, generation of events, and some of its parts (such as the cursors), including `TrackMode`, `ChartStyle`, and `ChartLength`. The `TrackMode` property specifically determines how mouse interaction with the graph is interpreted, and it is used to implement cursors, zooming, and panning.

Several Graph control methods are called directly on the Graph object, including the `Plot` and `Chart` methods. These methods are called on the Graph object to send data to multiple plots at once and on individual Plot objects to send new data to one specific plot at a time. Use the `Plot` methods to update and replace all data on the plots, and use the `Chart` methods to append new data to the plots.

Plot Methods

The following three Plot methods accept data in slightly different formats:

- `PlotY` (`yData`, `xFirst`, `xInc`, `bPlotPerRow`) plots Y data evenly spaced on the X axis relative to the index in the array. Using the `xFirst` and `xInc` parameters, you can specify the X value at the first data point and the incremental X value between data points. `yData` can be a one-dimensional array that updates the first plot on the graph or a two-dimensional array that updates the first *n* plots on the graph. If `bPlotPerRow` is `true`, each row of the `yData` array is equivalent to one plot; if `bPlotPerRow` is `false`, each column of the `yData` array is equivalent to one plot.
- `PlotXY` (`xyData`, `bPlotPerRow`) plots a two-dimensional array of data. If `bPlotPerRow` is `true`, the first row in the data array contains the X data and subsequent rows contain plots of Y data. If `bPlotPerRow` is `false`, the first column in the data array contains the X data and subsequent columns contain plots of Y data.
- `PlotXvsY` (`xData`, `yData`, `bPlotPerRow`) plots a one-dimensional or two-dimensional array of Y data against a one-dimensional array of X data. If `bPlotPerRow` is `true`, each row of the `yData` array is equivalent to one plot; if `bPlotPerRow` is `false`, each column of the `yData` array is equivalent to one plot.

Depending on the programming environment, some of these parameters might be optional. If not explicitly specified, these parameters use a default value.

Visual Basic (Some parameters optional):

```
CWGraph1.PlotY Voltages
```

Visual C++ (All parameters required):

```
m_CWGraph1.PlotY (VariantArray, COleVariant(0.0),
COleVariant(1.0), COleVariant(1.0));
```

Delphi (All parameters required):

```
CWGraph1.PlotY (Voltages, 0.0, 1.0, True);
```

Chart Methods

The following three Chart methods accept data in slightly different formats:

- `ChartY` (`yData`, `xInc`, `bChartPerRow`) charts Y data on one or more plots relative to the index of the data. The `xInc` parameter determines the X spacing between points passed to a plot. `yData` can be a scalar value adding one point to the first plot, a one-dimensional array adding *n* points to the first plot or one point to *n* plots, or a two-dimensional array adding multiple points to multiple plots.

  If `bPlotPerRow` is `true` and the `yData` array is one-dimensional, all the values in the `yData` array are appended to a single plot; if `bPlotPerRow` is `false` and the `yData` array is one-dimensional, each value in the `yData` array is appended to its own plot. If `bPlotPerRow` is `true` and the `yData` array is two-dimensional, each *row* of the `yData` array is appended to its own plot; if `bPlotPerRow` is false and the `yData` array is two-dimensional, each *column* of the `yData` array is appended to its own plot.
- `ChartXY` (`xyData`, `bChartPerRow`) charts a two-dimensional array of data. If `bPlotPerRow` is `true`, the first row in the data array contains the X data and subsequent rows contain plots of Y data. If `bPlotPerRow` is `false`, the first column in the data array contains the X data and subsequent columns contain plots of Y data.
- `ChartXvsY` (`xData`, `yData`, `bChartPerRow`) charts a one- or two-dimensional array of Y data against a one-dimensional array of X data. If `bPlotPerRow` is `true`, each row of the `yData` array is equivalent to one plot; if `bPlotPerRow` is `false`, each column of the `yData` array is equivalent to one plot.

Depending on the programming environment, some of these parameters might be optional. If not explicitly specified, these parameters use a default value.

```
CWGraph1.ChartY VariantArray,,False
```

Plots Collection

The Plots collection is a standard collection containing Plot objects. The collection contains one property, `Count`, that returns the number of Plot objects in the collection.

```
NumPlots = CWGraph1.Plots.Count
```

Usually, all plots and their properties are defined during design in the property pages. You can use the `Add`, `Remove`, and `RemoveAll` methods to programmatically change the number of plots on the graph. When you add

a plot to the collection, the new plot assumes the properties of the PlotTemplate object (see *PlotTemplate Object* later in this chapter). The `Remove` method requires the index of the plot you are removing.

```
CWGraph1.Plots.Add
```

```
CWGraph1.Plots.Remove 3
```

Use the `Item` method of the Plots collection to access a particular Plot object in the collection.

```
Dim Plot1 as CWPlot
```

```
Set Plot1 = CWGraph1.Plots.Item(1)
```

Plot Object

The Plot object represents an individual plot on the graph. The object contains a number of different properties that determine the display of the plot, including `LineColor`, `LineStyle`, `PointColor`, and `FillToBase`. You can set these properties during design in the property pages and change them programmatically.

```
CWGraph1.Plots.Item(1).LineColor = vbBlue
```

```
CWGraph1.Plots.Item(1).PointStyle = cwPointAsterisk
```

The following code fills the space between the first and second plot on the graph red.

```
CWGraph1.Plots.Item(1).FillToBase = True
```

```
Set CWGraph1.Plots.Item(1).BasePlot =
  CWGraph1.Plots.Item(2)
```

```
CWGraph1.Plots.Item(1).FillColor = vbRed
```

Each Plot object has a set of `Plot` and `Chart` methods similar to those of the Graph object. Calling these methods directly on the Plot object allows you to update one individual plot on the graph without affecting the other plots. For the `PlotY`, `PlotXvsY`, `ChartY`, and `ChartXvsY` methods, the data arrays must be one-dimensional.

```
CWGraph1.Plots.Item(4).PlotY Voltages
```

```
CWGraph1.Plots.Item(2).ChartXvsY xData, yData
```

The protoypes for the `PlotXY` and `ChartXY` methods are as follows:

```
ChartXY (xyData, bXInFirstRow)
```

```
PlotXY (xyData, bXInFirstRow)
```

With these methods, use a two-dimensional data array with exactly two rows or two columns. If `bXInFirstRow` is `true`, the first row of the array contains the X data and the second row of the array contains a plot of Y data. If `bXInFirstRow` is `false`, the first column of the array contains the X data and the second column of the array contains a plot of Y data.

```
CWGraph1.Plots.Item(4).PlotXY xyData, True
```

For detailed descriptions of these methods, see *Plot Methods* and *Chart Methods* earlier in this chapter.

PlotTemplate Object

The PlotTemplate object is a special instance of a Plot object used to specify the default property values of new plots. The PlotTemplate object properties are the identical to those of the Plot object and are set through the property pages or programmatically.

The PlotTemplate property values are used as default property values for newly created plots when the `Add` method is called on the Plots collection. If you call the `Chart` or `Plot` method on CWGraph and pass in data for more plots than are defined in the Plots collection, the CWGraph automatically creates enough plots for all the data. For example, if you have defined only two plots and you call the `PlotY` method with data for five plots, the previously defined two plots receive data from the first and second rows of the array (or columns, if `bPlotPerRow` is `false`) and the three automatically created plots receive data from the remaining three rows of the array (or columns, if `bPlotPerRow` is `false`).

```
CWGraph1.PlotTemplate.LineColor = vbRed
```

Cursors Collection

The Cursors collection is a standard collection containing Cursor objects. To move the cursors with your mouse while running an application, the `TrackMode` property of the graph must be set to a value that supports this operation. You can find valid values in the online reference under the `TrackMode` topic.

The Cursors collection contains one property, `Count`, that returns the number of Cursor objects in the collection.

```
NumCursors = CWGraph1.Cursors.Count
```

Usually, you define all cursors and their properties during design time in the property pages. If necessary, you can use the `Add`, `Remove`, and `RemoveAll` methods to programmatically change the number of cursors on the graph. The `Remove` method requires the index of the cursor you are removing.

```
CWGraph1.Cursors.Add
```

```
CWGraph1.Cursors.Remove 3
```

Use the `Item` method of the Cursors collection to access a particular Cursor object in the collection.

```
Dim FirstCursor as CWCursor
```

```
Set FirstCursor = CWGraph1.Cursors.Item(1)
```

Cursor Object

The Cursor object controls the position and other attributes of the individual cursors on the graph. Two frequently used Cursor object properties are `XPosition` and `YPosition`, which return or set the position of the cursor on the graph.

```
x = CWGraph1.Cursors.Item(2).XPosition
```

```
CWGraph1.Cursors.Item(1).YPosition = YLimit
```

A cursor can be associated with a specific plot on a graph. Set this association in the property pages or programmatically using the `SnapMode` and `Plot` properties of the cursor. If a cursor is associated with a specific plot, you can use the `PointIndex` property to set the cursor at any specific index on the plot or to return the position of the cursor on the plot. Set the `SnapMode` property with a predefined ComponentWorks constant.

```
CWGraph1.Cursors.Item(1).SnapMode = cwCSnapPointsOnPlot
```

```
Set CWGraph1.Cursors.Item(1).Plot =
   CWGraph1.Plots.Item(1)
```

```
ptIndex = CWGraph1.Cursors.Item(1).PointIndex
```

In Visual C++ and Delphi, this constant is defined in a separate header file that you must include in your program.

Axes Collection

The Axes collection is a standard collection containing all the Axis objects of the graph. A graph has one X axis and a varying number of Y axes. You can determine the number of Y axes at design time and can change them programmatically at run time. These different Axis objects are contained in the Axes collection and can be referenced by index. Usually, the X axis is at index one, and the Y axes are at subsequent indices.

The Axes collection has the property `Count`, which returns the number of Axis objects in the collection.

```
NumAxes = CWGraph1.Axes.Count
```

Usually, you define all axes and their properties at design time in the property pages. If necessary you can use the `Add`, `Remove`, and `RemoveAll` methods to programmatically change the number of axes on the graph. The `Remove` method requires the index of the axis you are removing.

```
CWGraph1.Axes.Add
```

```
CWGraph1.Axes.Remove 3
```

Use the `Item` method of the Axes collection to access a particular Axis object in the collection.

```
Dim xAxis as CWAxis
```

```
Set xAxis = CWGraph1.Axes.Item(1)
```

Axis Object

The Axis object contains all the properties of the individual axes on the graph and is identical to the Axis object used on the Knob and Slide controls. The Axis object and its parts are described in detail in the *The Knob and Slide Controls* section earlier in this chapter.

Events

The graph generates a number of different events that enable your application to react to user interaction with the graph. The graph automatically processes certain mouse actions such as panning and zooming, for which you do not need to develop any event handler routines.

The `TrackMode` property, which you can set through the property pages or programmatically, determines the type of events generated and other automatic processing. Some common modes on the graph generate events for mouse interaction with cursors, plots, and the plot area, as well as moving cursors and panning and zooming the graph.

To move the cursors with the mouse during program execution, set the `TrackMode` property to a compatible value using either the property pages or your program.

```
CWGraph1.TrackMode = cwGTrackDragCursor
```

In track mode, the graph generates the `CursorChange` event when any cursor moves. This event can initiate a response action in your application.

```
Private Sub CWGraph1_CursorChange(CursorIndex As Long,
  XPos As Variant, YPos As Variant, bTracking As Boolean)
    xDisplay = XPos
    yDisplay = YPos
End Sub
```

Panning and Zooming

You can use the `TrackMode` property to specify panning and zooming modes on the graph. Panning is useful when the graph displays only a subset of the data that has been plotted. If you enable panning, the user can scroll through all data plotted on the graph, essentially shifting the graph's display to different portions of the plot. You can enable panning for both or either of the X and Y axes.

```
CWGraph1.TrackMode = cwGTrackPanPlotAreaXY
```

Users can use zooming to enlarge or diminish a portion of the plot displayed by the graph. For example, if the user zooms on a section of a plot, the graph displays a smaller portion of the plot in the same amount of display area, which enlarges the detail of that section.

```
CWGraph1.TrackMode = cwGTrackZoomRectXY
```

Tutorial: Graph and Button Controls

This tutorial shows you how to integrate the Button and Graph controls in a simple application.

The button control has several different display styles but maintains a single set of property sheets, event functions, and style of interaction with the program. You can use the Button control as an input or output. When using it as an input, create a push button or a switch to initiate an action or switch between actions. For an output, create an LED to indicate a Boolean condition.

The Graph is the most complex of the user interface objects. You can use it in two basic modes—Plot or Chart. You can select the mode by using different methods in your program to pass data to the graph.

This tutorial uses Visual Basic syntax, but the discussion is in general terms so you can follow it in any compatible programming environment. Remember to adjust any code to your specific programming language. Refer to the *Building ComponentWorks Applications* chapters for information about implementing any step in different programming environments. You also can refer to the tutorial examples installed with ComponentWorks for a completed version of this example in several different programming environments.

Designing the Form

1. Open a new project and form. If you are working in Visual C++, select a dialog-based application and name your project `ButtonGraphExample`.
2. Load the ComponentWorks user interface controls (specifically the Button and Graph) into your programming environment.

3. Place a ComponentWorks Graph control (shown at left) on the form. Keep its default name, `CWGraph1`. Open the Graph control property page and, in the **Axes** tab, disable autoscaling and change the X-axis range to `0` (minimum) and `20` (maximum). Examine other tabs.

   For more information about some of the advanced features, refer to Chapter 12, *Building Advanced Applications*, and the online reference.

4. Place two ComponentWorks buttons (shown at left) on the form. Change their name property to `Chart` and `Plot` in the default property sheets in Visual Basic and Delphi Object Inspector or the custom property pages in Visual C++.

   In the custom property pages, change each of their styles to `Command Button`. In the **Button** tab of the property sheets, change the On Text and Off Text to `Chart` for the first button and `Plot` for the second button.

5. Place another ComponentWorks button on the form. Change its name to `ChartSelect`. Leave its style as Vertical Toggle.
6. Place two Visual Basic labels next to the toggle button and change their caption properties so that the up state of the switch is labeled `Scope Chart` and the down state `Strip Chart`.

Your form should look similar to the one shown below.

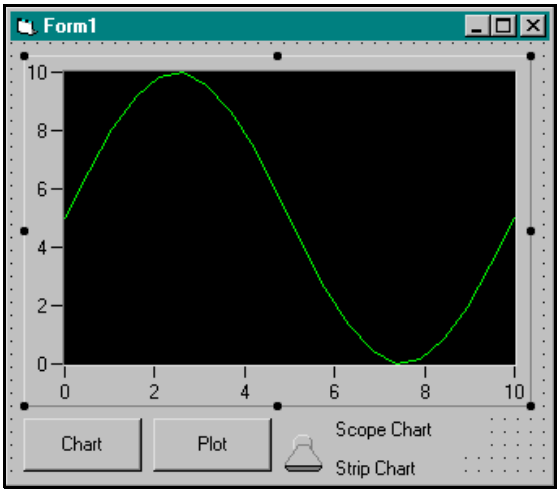

Figure 6-6. ButtonGraphExample Form

Developing the Code

Develop the code so that data is either plotted or charted on the graph in response to pressing the appropriate buttons.

1. Define an event handler routine for the **Plot** button to be called when the button is pressed. In the event handler the program creates an array of 20 points and plots it on the graph.

   Generate the event handler routine for the `Click` event of the **Plot** button. Add the following code to the `Plot_Click` subroutine. In Visual C++, remember to generate member variables for any controls referenced in the program. See the online tutorial programs for Visual C++ and Delphi code examples.

   ```
   Private Sub Plot_Click()
      Dim data(0 To 20) As Double
      CWGraph1.Axes.Item(1).Maximum = 20
      CWGraph1.Axes.Item(1).Minimum = 0
      For i = 0 To 20
         data(i) = Rnd * 10#
      Next i
      CWGraph1.PlotY data, 0, 1, True
   End Sub
   ```

   This code generates an array of 20 random numbers. The `PlotY` method then replaces any data on the plot and plots the new data starting at zero on the X axis. To ensure that the new data appears on the graph, the minimum and maximum values of the X axis (`CWGraph1.Axes.Item(1)`) are reset to 0 and 20. This routine uses a one-dimensional array with the `PlotY` method to generate one trace. You also can use a two-dimensional array to generate multiple traces.

2. Generate the event handler routine for the `Click` event of the **Chart** button, which charts some random data on the graph when pressed. Add the following code to the `Chart_Click` subroutine:

   ```
   Private Sub Chart_Click()
      Dim data As Double
      For i = 0 To 59
         data = Rnd * 10#
         CWGraph1.ChartY data, 1, True
      Next i
   End Sub
   ```

The `Chart_Click` subroutine performs a similar action to `Plot_Click`, except that `Chart_Click` generates random points and charts them individually. When the `ChartY` method is called, it appends the new data to the data already on the graph. If you add only one point at a time, you can use a scalar value or a one-dimensional array to pass multiple points to the trace. Use a two-dimensional array to chart multiple traces.

3. The `ChartStyle` property on the graph sets the charting style. This example uses the toggle switch to switch between the two charting styles. To have your program respond when the user switches styles, add the `ChartSelect_ValueChanged` event handler routine for the switch. Use the `Value` property of the switch to retrieve or set the current value of the control.

   Generate the event handler routine for the `ValueChanged` event of the switch. Add the following code. In Visual C++ and Delphi, include the appropriate header files to define the constant values.

```
Private Sub ChartSelect_ValueChanged(ByVal Value
  As Variant)
    If ChartSelect.Value = True Then
        CWGraph1.ChartStyle = cwChartScope
    Else
        CWGraph1.ChartStyle = cwChartStrip
    End If
End Sub
```

4. Save the project and form as `ButtonGraphExample`.

Testing Your Program

Run and test the program. Notice the difference between the Plot, Strip Chart, and Scope Chart options.

`PlotY` and `ChartY` are the two most common methods for passing data to the graph. However, there are two more Plot methods (`PlotXY` and `PlotXvsY`) and two more Chart methods (`ChartXY` and `ChartXvsY`) that affect how data is displayed on the graph. For more information about these methods, see *The Graph Control* earlier in this chapter or the online reference. Refer to Chapter 12, *Building Advanced Applications*, for more information about advanced graph features such as cursors and multiple axes.

7

Using the Data Acquisition Controls

This chapter describes how you can use the ComponentWorks Data Acquisition (DAQ) controls in your application to perform input and output operations using your DAQ hardware. It explains the individual controls and their most commonly used properties, methods, and events and includes tutorials that give step-by-step instructions on using the controls in simple programs.

Refer to the *Building ComponentWorks Applications* chapters for information about using the ComponentWorks controls in different programming environments. Use the online reference (available by selecting **Programs»National Instruments ComponentWorks» ComponentWorks Reference** from the Windows **Start** menu) to find information about each control and its properties, methods, and events.

What Are the Data Acquisition Controls?

Use the DAQ controls to program your DAQ hardware. ComponentWorks includes nine ActiveX controls for performing DAQ operations as well as a utility control that contains other DAQ support functions. Each control is

used for one specific type of operation such as analog input, analog output, and so on. The following is a list of the DAQ controls:

DAQ Control	Operation
CWAIPoint	Single Point Analog Input
CWAI	Waveform Analog Input
CWAOPoint	Single Point Analog Output
CWAO	Waveform Analog Output
CWDIO	Single Point Digital Input/Output
CWDI	Waveform Digital Input
CWDO	Waveform Digital Output
CWCounter	Data Acquisition Counter Functions
CWPulse	Data Acquisition Pulse Generation Functions
CWDAQTools	Data Acquisition Utilities

You can set most properties through property pages as you design your program. Property pages are the best place to become familiar with the different properties of a control. In certain cases, you might need to change the value of one or more properties in your program code. Throughout this chapter, examples demonstrate how to change property values programmatically.

Data Acquisition Configuration

Before you can use your National Instruments data acquisition hardware with the ComponentWorks DAQ controls you must configure your DAQ device using the NI-DAQ driver configuration utility. Make sure that you follow the directions in the NI-DAQ driver documentation (online) to properly configure the hardware. You also can test the hardware and perform simple input/output operations with the configuration utility. Once a data acquisition device is configured, it is assigned a device number that you use to reference the device in your application. Select the device and device number in the property pages of each control.

Object Hierarchy and Common Properties

Some of the ComponentWorks Data Acquisition controls are made up of a hierarchy of less complex individual objects. Understanding the relationship among the objects in a control is the key to properly programming with the control. Dividing a control into individual objects makes it easier to work with because each individual component has fewer parts.

The top-level object of each DAQ control shares common properties, as described in the following sections.

Device, DeviceName, and DeviceType

Each control has a `Device` property that you use to select the hardware device used by the control. You can set this property from a pulldown menu in the property pages of each control, or you can set it programmatically.

```
CWAI1.Device = 2
```

`DeviceName` and `DeviceType` are read-only properties that return the name and type number of the selected device. The name of a device is its descriptive name, such as `AT-MIO-64E-3`. The type number is a unique number assigned to each hardware device type in the NI-DAQ driver. You can use these properties to control the execution of your application.

```
If CWAI1.DeviceType = 16 Then…
```

Channel Strings

Most DAQ controls have a channel string property in their object hierarchy. Use the channel string to specify which channels on a data acquisition device are used by a particular operation. If you use only one channel, enter the channel number in the string.

```
CWAOPoint1.ChannelString = "1"
```

In many cases, you must specify more than one channel in a channel string. If you want to specify a series of consecutive channels, specify the first and last channel in your list separated by a colon.

```
CWAI1.Channels.Item(1).ChannelString = "1:4"
‘Specifies channel 1, 2, 3, and 4
```

You also can specify a reverse list of consecutive channels.

```
CWAI1.Channels.Item(1).ChannelString = "6:3"
'Specifies channels 6, 5, 4, and 3
```

Note ***Certain DAQ devices require that a multiple channel acquisition use a reverse list of consecutive channels ending with channel 0. These devices include all 500-, 700-, and 1200-series devices, as well as the Lab and LPM series cards.***

```
CWAI1.Channels.Item(1).ChannelString = "3:0"
```

You can specify non-consecutive channels in a channel string by listing each channel separated by commas.

```
CWAIPoint1.Channels.Item(1).ChannelString = "0,1,3,5"
```

SCXI Channel Strings

By using a different channel string to specify your channels, you can use the ComponentWorks DAQ controls with SCXI signal conditioning hardware. To configure channels on an SCXI module, set the `Device` property to the number of the DAQ board directly or indirectly connected to the desired SCXI module. The channel string(s) of your controls include information about the DAQ device channel, SCXI chassis number, SCXI module number, and SCXI channel number. The string has the following format.

```
oba!scx!mdy!z
```

In the SCXI channel string, `a` represents the DAQ device (onboard) channel used for the acquisition (with analog input channels only), `x` represents the chassis number, `y` the module number, and `z` the channel number on the SCXI module. The onboard channel number usually is one less than the chassis number.

```
CWAIPoint1.Channels.Item(1).ChannelString =
  "ob0!sc1!md1!0"
'Specifies channel 0 on module 1 in chassis 1
```

You can specify multiple channels on an SCXI module in a consecutive list.

```
CWAI1.Channels(1).ChannelString = "ob0!sc1!md3!1:5"
'Specifies channel 1 through 5 on module 3 in chassis 1
```

Other combinations are listed in the following table.

String Syntax	Description
`ob0!sc1!md2!5`	Channel 5 on module 2 of SCXI chassis 1 read through onboard Channel 0.
`ob0!sc1!md2!0:7`	Channels 0 through 7 on module 2 read through onboard Channel 0.
`ob1!sc2!md1!20:24`	Channels 20 through 24 of module 1 on chassis 2 read through onboard Channel 1.

Note ***All lines on a digital SCXI module are grouped into a single port. The SCXI channel string for digital lines refer to this port 0 only. For example,*** `sc1!md3!0`.

ExceptionOnError and ErrorEventMask

DAQ controls handle error checking in a number of different ways. By default, each DAQ control generates an exception that your programming environment handles when an error occurs. You can disable the generation of exceptions by setting the `ExceptionOnError` property to `false`. If exceptions are disabled, each call to a DAQ control method returns an error code. If the code is equal to zero, the operation completed normally. If the value is non-zero, either a warning or error occurred and the application should handle the condition.

Another type of error notification is the generation of error and warning events in response to error conditions. Each event calls a corresponding event handler routine that processes the error information. Use the `ErrorEventMask` property on each DAQ control to limit the error and warning event generation to specific operations (contexts) of the DAQ controls. For example, by default the AI control generates an error event only during the following contexts: `cwaiReadingData`, `cwaiReadingDataContinuous`, and `cwaiReadingDataSWAnalog`.

These contexts refer to asynchronous operations, which mean the AI control is in the process of acquiring data and returning it to the application. Other contexts, such as `cwaiStartingAcquisition` or `cwaiConfiguringChannels`, do not generate error events by default. To select which contexts generate error events, add the values of the `CWAIErrorContexts` constants and assign the sum to the `ErrorEventMask` property.

```
CWAI1.ErrorEventMask = cwaiReadingData +
cwaiReadingDataContinuous + cwaiReadingDataSWAnalog +
cwaiStartingAcquisition
```

Error handling is discussed in more detail in Chapter 12, *Building Advanced Applications*.

AIPoint Control—Single Point Analog Input

Use the AIPoint control to acquire one point of data from one or more analog input channels at a time to monitor slowly changing processes, such as temperature. After you set the properties of the control, the application can acquire a single scan of data using a simple method call to the AIPoint control. A *scan* is defined as an acquisition of one point from each channel in the channel list.

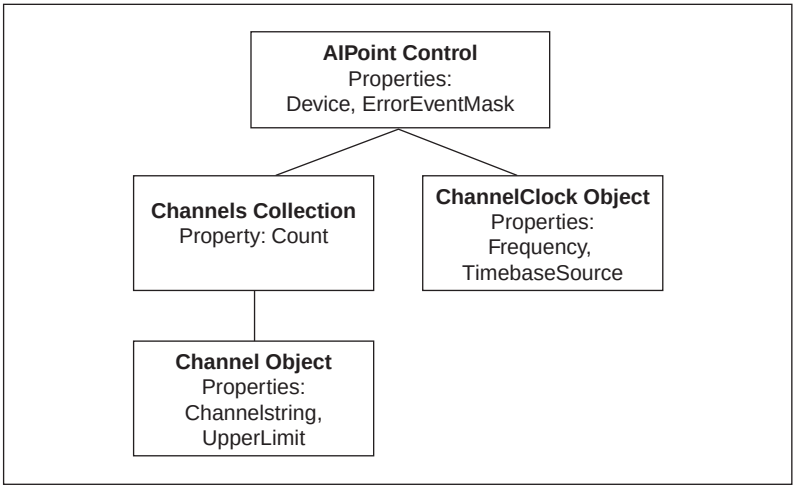

Figure 7-1. AIPoint Control Object Hierarchy (Single Point Analog Input)

The object hierarchy of the AIPoint control contains a Channels collection with Channel objects and a ChannelClock object.

AIPoint Object

In addition to the default properties of each DAQ control, the AIPoint object has one more property, `ReturnDataType`, that determines whether the acquired data is returned to the application as voltage data, binary values, or both.

The AIPoint object has two methods, `SingleRead` and `Reset`. The `SingleRead` method performs a single scan using the values set in the control properties. The `SingleRead` method has two variant parameters: `data` and an optional parameter, `TimeLimit`, which specifies the time limit for the acquisition.

```
Dim data as Variant
CWAIPoint1.SingleRead data, 1.0
```

The `data` variable must be a variant and is assigned the values read. The data is returned in the format specified by the `ReturnDataType` property, either as a scalar or one-dimensional array.

When you call the `SingleRead` method, the hardware is configured using the values set in the control's properties. This configuration is done only when necessary, such as calling `SingleRead` the first time or after changing any of the properties. You also can unconfigure the control manually using the `Reset` method. The control then configures the hardware on the next acquisition.

```
CWAIPoint1.Reset
```

Channels Collection

The Channels collection exists on all analog input and analog output DAQ controls, except the CWAOPoint control. This collection contains the individual Channel objects that determine which hardware channels are used by the control. The collection has a read-only property, `Count`, that returns the number of Channel objects in the collection.

```
NumChanObjects = CWAIPoint1.Channels.Count
```

You can get the value of the read-only property `NChannels` after a control has been configured. `NChannels` returns the total number of channels used by the control. The value returned from this property is valid only when the control is configured, which you can do using the `Configure` or `SingleRead` method of the respective control.

```
NumChannels = CWAIPoint1.Channels.NChannels
```

Like all collections, the Channels collection has an `Item` method you use to access a particular Channel object in the collection. To retrieve a Channel object, call the `Item` method and specify the (one-based) index of the channel in the collection.

```
CWAIPoint1.Channels.Item(2)
```

In this way, you can programmatically change properties of individual Channel objects. The Channels collection contains several methods that you can use to modify the number of Channel objects in the collection. The `RemoveAll` method clears the collection of all Channel objects. Use the `Remove` method to delete individual Channel objects. The `Add` method adds a new Channel object to the collection.

```
CWAIPoint1.Channels.RemoveAll
CWAIPoint1.Channels.Remove 1
CWAIPoint1.Channels.Add "1", 10, -10, cwaiDIFF, cwaiDC
```

Channel Object

Each Channel object contains information about one or more channels used by a DAQ control. The individual Channel object contains properties such as `ChannelString`, `InputMode`, `UpperLimit`, and `LowerLimit`. For example, the `ChannelString` property specifies which channels are affected by the Channel object, while the remaining properties determine how the channels are used. You can read and set these properties programmatically.

```
CWAIPoint1.Channels.Item(1).ChannelString = “0,1”
MaxVolts = CWAIPoint1.Channels.Item(1).UpperLimit
```

ChannelClock Object

The ChannelClock object determines the timing that the AIPoint control uses in the actual analog-to-digital conversions within a scan. Use it to increase the delay between the acquisitions of different channels or to synchronize conversions with an external signal. The ChannelClock object directly affects the conversions by allowing you to select either an internal source and frequency or an external source and exact description of the signal source, such as an I/O pin or RTSI pin.

By default, ComponentWorks chooses channel clock settings that should work for most applications. You usually configure the ChannelClock in the property pages, but you might need to change its settings from your program. For example, you can switch to frequency mode and change the frequency setting.

```
CWAIPoint1.ChannelClock.ClockSourceType =
  cwaiInternalCS
CWAIPoint1.ChannelClock.Frequency = 10000
```

AI Control—Waveform Analog Input

Use the AI control to perform waveform analog input operations including single shot and continuous acquisitions. With the AI control, you can acquire data from one or more channels at a time and configure many different modes, such as start and stop triggers, pause conditions, and different channel and scan clocks. Use this control for any application that requires fast acquisition of multiple points per channel, such as frequency analysis. After the properties of the control are set, the application can perform acquisitions using a number of simple method calls.

The object hierarchy of the AI control separates the functionality of the control into individual objects. The Channels collection and Channel objects specify the channels and channel attributes used for the acquisition. The condition objects specify when an acquisition starts, pauses, or stops. The clock objects specify the rate of the acquisition.

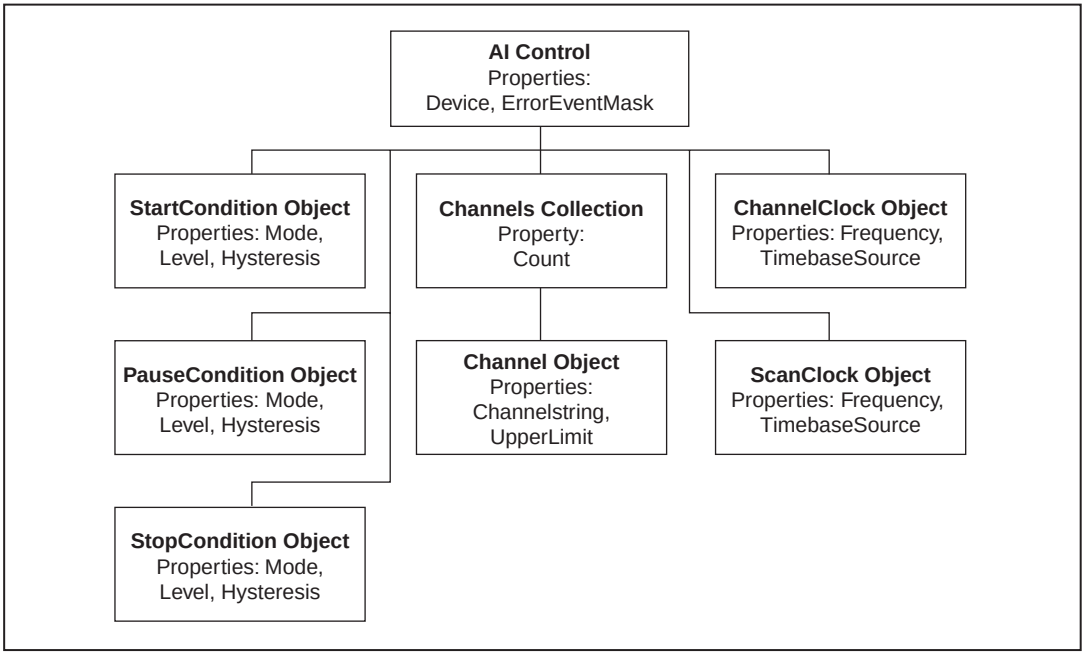

Figure 7-2. AI Control Object Hierarchy (Waveform Analog Input)

AI Object

Along with the default properties of the DAQ controls, the AI object has other properties. One property is `ReturnDataType`, which determines whether the acquired data is returned to the application as voltage data, binary values, or both. The `NScans` property (number of scans to acquire) specifies the number of scans acquired in a single-shot acquisition or the number of scans returned at a time in a continuous acquisition.

```
CWAI1.NScans = 5000
```

If the `UseDefaultBufferSize` property is set to `False`, the `NScansPerBuffer` property determines the size of the acquisition buffer; otherwise ComponentWorks automatically selects the buffer size.

The AI control uses the Channels collection and Channel object in the same manner as the AIPoint control. See the *AIPoint* section earlier in this chapter for more information about these objects. Consult the online reference for more information about the individual properties, methods, or events of the AI object or any of its underlying objects.

Methods and Events

The AI control has a number of simple methods for running the different acquisition processes. The normal and recommended acquisition type is an asynchronous acquisition, which is controlled using four different method calls and offers the most flexibility and control over the acquisition. Alternatively, in environments that do not support event handler routines, you can use an additional method to perform a synchronous acquisition.

Asynchronous Acquisition

The methods to perform an asynchronous acquisition are `Configure`, `Start`, `Stop`, and `Reset`. Use these methods to control the acquisition. These methods do not require any parameters because acquisition parameters are set through the control properties.

Use the `Configure` method to configure the DAQ driver and hardware with the acquisition parameters. `Configure` must be called before the `Start` method. Use the `Start` method to start the acquisition. Use the `Stop` method only during a continuous acquisition to stop such an acquisition. Use the `Reset` method to unconfigure the AI control and free any resources reserved during configuration. You also must call the `Configure` method after you change any of the control properties so that they can take effect and after you call the `Reset` method so that you can restart the acquisition.

```
Private Sub StartAcquisition_Click()
    CWAI1.Configure
    CWAI1.Start
End Sub

Private Sub StopAcquisition_Click()
    CWAI1.Stop
End Sub

Private Sub ResetAcquisition_Click()
    CWAI1.Reset
End Sub
```

The AI control returns data from an asynchronous acquisition by generating an `AcquiredData` event. The acquired data is returned as arrays, `ScaledData` and/or `BinaryCodes`, passed to the event handler. You can process the data inside the event handler by displaying it on a graph or writing it to a file.

```
Private Sub CWAI1_AcquiredData(ScaledData As Variant,
BinaryCodes As Variant)
    CWGraph1.PlotY ScaledData
End Sub
```

You also can use the `Read` method on the AI control to read data from an ongoing acquisition without events. Specify how many points to read from the acquisition and supply a variant variable to return the data. You also can use another variable of type `CWAIReadSpec` to specify and return additional information about the acquisition.

```
Private Sub Start_Click()
    CWAI1.Configure
    CWAI1.Start
End Sub

Private Sub Read_Click()
    Dim ReadSpec As New CWAIReadSpec
    Dim data As Variant
    CWAI1.Read 100, data, ReadSpec
End Sub
```

Synchronous Acquisition

Certain programming environments do not support event handler functions and are therefore not suited for running an asynchronous acquisition. In such cases, the AI control can perform a synchronous acquisition using the `AcquireData` method. The `AcquireData` method requires you to pass in

two variables for the scaled and binary data, which are used to return the acquired data at the completion of the acquisition.

You must call the `Configure` method before calling `AcquireData`, and you cannot run a continuous acquisition using this method. Because the `AcquireData` method takes control of the program until the acquisition is completed, you also can specify a timeout parameter in seconds that forces the method to return in the time limit specified.

```
Private Sub RunAcquisition_Click()
   Dim Voltages As Variant
   Dim BinaryCodes As Variant

   CWAI1.Configure
   CWAI1.AcquireData Voltages, BinaryCodes, 5
   'timeout is 5 seconds
   CWGraph1.PlotY Voltages
End Sub
```

Error Handling

The AI control also has `DAQError` and `DAQWarning` events that can be used for error handling.

```
Private Sub CWAI1_DAQError(ByVal StatusCode As Long,
  ByVal ContextID As Long, ByVal ContextDescription As
  String)
   MsgBox "DAQ Error: " + CStr(StatusCode)
End Sub
```

ScanClock and ChannelClock Objects

The AI control contains both a ScanClock and ChannelClock object to specify the scan rate and interchannel delay. These two settings apply if you acquire multiple points of data from more than one channel. In this type of operation, the data acquisition device performs repeated scans, in which one scan is an acquisition of one data point from each channel in the channel list. The timing within one scan is called the *interchannel delay*, and the ChannelClock object automatically selects it. The rate at which scans are acquired is called the *scan rate* and is set in the ScanClock object. The effective acquisition rate per channel (the rate at which points on one channel are acquired) is also the scan rate.

The ScanClock object is critical to the operation of the AI control and you must set it for most applications. In common operations, specify an internal frequency and set the acquisition (scan) rate in the property pages. More complex operations can include specifying an external source for the

ScanClock to synchronize the acquisition with another process. You can specify these settings in the property pages or programmatically. After the control is configured, you can read back the actual frequency or period used for the acquisition.

```
CWAI1.ScanClock.Frequency = 200
CWAI1.Configure
ScanPeriod = CWAI1.ScanClock.ActualPeriod
```

StartCondition, PauseCondition and StopCondition Objects

The StartCondition, PauseCondition, and StopCondition objects control when an acquisition starts, pauses, and stops. The main property of the Condition object is `Type`, which sets the overall operation of the object. The value of the `Type` property determines which of the remaining properties on the Condition object are used.

Certain condition types are supported only by specific hardware. Verify that your data acquisition device supports the desired operation. For example, all hardware analog conditions require specific analog trigger circuitry on the acquisition device.

The StartCondition object controls when an acquisition is started. By default (`Type` set to `cwaiNoActiveCondition`) the acquisition is started immediately after the corresponding method call.

```
CWAI1.StartCondition.Type = cwaiNoActiveCondition
```

You can set the StartCondition to start the acquisition on a digital or analog trigger. In such cases, the hardware is set to start on the corresponding software call, but actual conversions do not start until the digital or analog trigger arrives. You can set trigger conditions either in the property pages or programmatically.

```
CWAI1.StartCondition.Type = cwaiHWAnalog
CWAI1.StartCondition.Level = 5
CWAI1.StartCondition.Hysteresis = 0.1
CWAI1.StartCondition.Source = 1
CWAI1.StartCondition.Mode = cwaiRising
```

The PauseCondition object controls when an ongoing acquisition is paused, which might be done in response to an external digital or analog signal with a limited number of data acquisition devices. The `Type` property can have the following settings: `None`, `Hardware Digital Gate`, and `Hardware Analog Gate`. The remaining properties of the PauseCondition specify

specific pause conditions. You can pause the acquisition while above or below a specific analog level (high and low for digital) or while inside or outside a specific analog window.

```
CWAI1.PauseCondition.Type = cwaiHWAnalog
CWAI1.PauseCondition.Mode = cwaiInside
```

The StopCondition object controls when the acquisition is stopped. The default mode is to stop after the acquisition buffer, set by the `NScans` property on the AI control, has been filled once. You also can select to run a continuous acquisition so the acquisition stops only on a user command or when an error occurs. Other advanced options include stopping on a hardware digital or analog signal or a software analog condition (single shot or continuous). These last three types also support pretrigger scans, which means you can specify to acquire a number of points before the stop condition and the remainder after the stop conditions. The remaining properties are similar to the StartCondition and PauseCondition objects.

The software analog trigger type on the StopCondition supports analog triggering on devices that do not have an explicit hardware analog trigger circuit. In this mode, data is continuously acquired from the data acquisition device but returned only when it matches the specified conditions. This mode behaves similarly to a hardware analog start trigger, and you can run it, either as a one-shot or continuous acquisition. The continuous software analog trigger makes it easy for you to duplicate the operation of an oscilloscope in your application.

Tutorial: Using the AIPoint and AI DAQ Controls

This tutorial shows you how to use the AIPoint and AI controls in a simple program to acquire one scan of data from several channels using the AIPoint controls, how to perform a simple waveform acquisition using the AI control, and how to display the data on a graph.

This tutorial uses Visual Basic syntax, but the discussion is in general terms so you can follow it in any compatible programming environment. Remember to adjust any code to your specific programming language. Refer to the *Building ComponentWorks Applications* chapters for information about implementing any step in different programming environments. You also can refer to the tutorial examples installed with ComponentWorks for a completed version of this example in several different programming environments.

Designing the Form

1. Open a new project and form. If you are working in Visual C++, select a dialog-based application and name your project `AIExample`.
2. Load the ComponentWorks User Interface controls (specifically, the Graph, Button, and Numeric Edit Box) and the Data Acquisition controls (specifically, the AIPoint and AI controls) into your programming environment.

3. Place an AIPoint and AI control on your form. You configure their properties in the next section.

4. Place a ComponentWorks Graph control on the form. Keep its default name `CWGraph1`.

   For information about advanced features of the Graph control, refer to Chapter 12, *Building Advanced Applications*, and the online reference.

5. Place two ComponentWorks Numeric Edit Boxes on the form. Keep their default names, `CWNumEdit1` and `CWNumEdit2`.

6. Place two ComponentWorks buttons on the form. Change their `Name` property to `Acquire` and `Start` in the default property sheets in Visual Basic and Delphi or the custom property pages in Visual C++.

   In the custom property pages, change the style of both buttons to `Command Button`. Also, in the **Button** tab of the property pages, change the **On Text** and **Off Text** to `Acquire` for the first button and `Start` for the second button.

Your form should look similar to the one shown below.

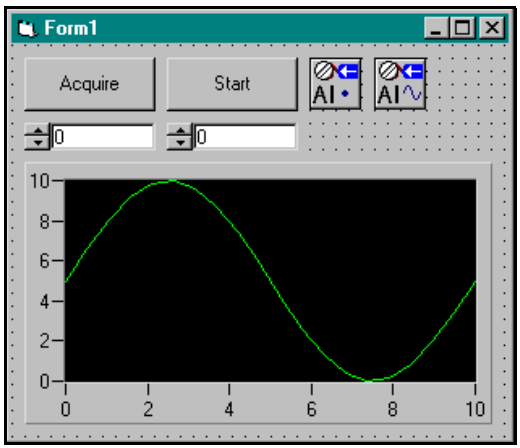

Figure 7-3. AIExample Form

Setting the DAQ Properties

You normally configure the default property values of the different controls before you develop your program code. When using the DAQ controls, most or all properties are set during design and do not change during program execution. Method calls in your program start and stop the acquisition processes. If necessary, you can set the properties of the DAQ controls at run time.

1. Open the custom property pages for the AIPoint control on the form by right clicking on the control and selecting **Properties…**. In the **Channels** tab, select your data acquisition device from the pulldown menu. Click the **New** button to add a new Channel object and set it to channels 0 and 1. Do this by entering `"0,1"` in the **Channels** field. The new channels are displayed in the channel list on the left side. Close the AIPoint property pages. If necessary, reverse the order of the channels for your DAQ device. See the *Channel Strings* section earlier in this chapter for more information.
2. Open the custom property pages for the AI control on the form. In the **Channels** tab, select your data acquisition device from the pulldown menu. Click the **New** button to add a new Channel object and set it to channel 1. Do this by entering `"1"` in the **Channels** field. The new channels are displayed in the channel list on the left side. In the **Clocks** tab, change the scan rate (Scans/second) to `1000`. Close the AI property pages.

Developing the Code

Next, you develop the code so that data is acquired and displayed when the buttons are pushed.

1. For the **Acquire** button, define an event handler routine to be called when the **Acquire** button is pressed. In the event handler, acquire one scan of data (one point each from Channel 0 and 1) and display the two points in the numeric edit boxes.

   Generate the event handler routine for the `Click` event of the **Acquire** button. In the event handler, declare a variable as a `Variant`. Pass this variable to the `SingleRead` method of the AIPoint control. Then display the data returned in the first variable in the numeric edit boxes. Add the following code to the `Acquire_Click` subroutine. In Visual C++, remember to generate member variables for any controls referenced in the program. See the `tutorials` folder for Visual C++ and Delphi code examples.

The following code acquires a scan from Channels 0 and 1 using the `SingleRead` method and returns the data in a one-dimensional array (`Volts`). The two values are copied to the numeric edit boxes to be displayed.

```
Private Sub Acquire_Click()
    Dim Volts As Variant
    CWAIPoint1.SingleRead Volts, 1
    CWNumEdit1.Value = Volts(0)
    CWNumEdit2.Value = Volts(1)
End Sub
```

2. For the **Start** button, define an event handler routine to be called when the **Start** button is pressed. In the event handler, start an asynchronous acquisition on the AI control. Because the acquisition is asynchronous, the program regains control immediately after the acquisition is started, while the acquisition continues to run in the background.

   Generate the event handler routine for the `Click` event of the **Start** button. In the event handler, call the `Configure` and `Start` methods of the AI control. Add the following code to the `Start_Click` subroutine. In Visual C++, remember to generate member variables for any controls referenced in the program. See the `Tutorial` folder for Visual C++ and Delphi code examples.

   The following code starts the acquisition and then returns control to the program.

```
Private Sub Start_Click
    CWAI1.Configure
    CWAI1.Start
End Sub
```

3. The AI control fires an `AcquiredData` event when it is ready to return the acquired data. You must generate the corresponding event handler to receive and process the data. In this example, plot the data on a graph. Use the `PlotY` method of the Graph control to display the returned `ScaledData` array.

   Generate the event handler for the `AcquiredData` event of the AI control and add the following line of code.

```
Private Sub CWAI1_AcquiredData(ScaledData As
 Variant, BinaryCodes As Variant)
    CWGraph1.PlotY ScaledData, 0, 1, 1
    CWAI1.Reset
End Sub
```

4. Save the project and form as `AIExample`.

Testing Your Program

Run and test the program. Click on the **Acquire** and **Start** buttons to perform the acquisitions. Your application should look similar to the following figure. The exact data displayed on the graph depends on the signal connected to your data acquisition board.

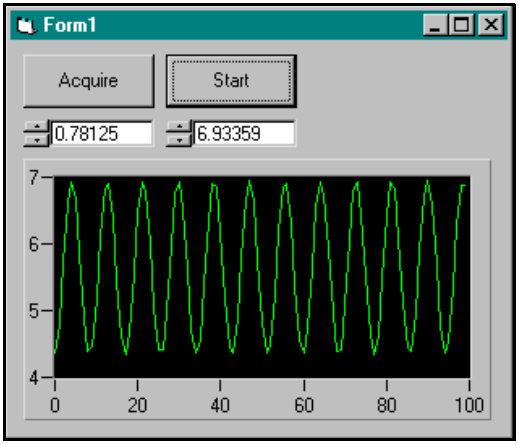

Figure 7-4. Testing AIExample

You can enhance the waveform acquisition performed in this example by defining more properties of the AI control. For example, you can perform a continuous acquisition by setting the `Type` property of the AI control StopCondition object to `Continuous` on the **Conditions** tab of the AI control property pages. In continuous mode, the acquisition continues and repeatedly returns the specified number of points in the `DataAcquired` event. The AI control has a `Stop` method you can call to stop a continuous acquisition.

AOPoint Control—Single Point Analog Output

Use the AOPoint control to update one or more analog output channels for slow process control systems, such as setting a control output (for example, the power of a heater or throughput of a valve). After you set the properties of the control, the application can update the configured channels using a simple method call to the AOPoint control.

Figure 7-5. AOPoint Control Object Hierarchy (Single Point Analog Output)

The AOPoint control contains no other objects. All properties of the control are part of the top-level object.

AOPoint Object

The AOPoint object has several unique properties that determine the operation of the control.

The `ChannelString` property, together with the `UpperLimit`, `LowerLimit`, `Reference Source`, and `ChannelType` properties, control the AOPoint output. You usually configure these properties through the property pages, but you also can set them programmatically.

```
CWAOPoint1.ChannelString = "0,1"
```

```
CWAOPoint1.UpperLimit = 10.0
```

Methods

The AOPoint object has two methods: `SingleWrite` and `Reset`. The `SingleWrite` method performs a single update on channels configured for the control. The `SingleWrite` method has two variant parameters, `Values` and `Scaled`.

Use the `Values` parameter to pass the analog values to the method to be generated by the analog output channels. Use the optional `Scaled` parameter to specify whether the analog values are passed as scaled or binary data. By default, the information is interpreted as voltage data.

```
'Update one channel
CWAOPoint1.ChannelString = "0"
CWAOPoint1.SingleWrite 5.0, True
```

```
'Update two channels
Dim VoltsArrayData(0 to 1)
CWAOPoint2.ChannelString = "0,1"
VoltsArrayData(0) = 3.2
VoltsArrayData(1) = 6.5
CWAOPoint2.SingleWrite VoltsArrayData, True
```

You can send the output data to the `SingleWrite` method as a scalar value for updating one channel or as a one-dimensional array of values for updating more than one channel.

When you call the `SingleWrite` method, the hardware is configured using the values set in the control properties. This configuration is done only when necessary, such as when `SingleWrite` is called the first time or after changing any of the properties. You also can unconfigure the control manually using the `Reset` method, which causes the control to configure the hardware on the next `SingleWrite` method call.

```
CWAOPoint1.Reset
```

AO Control—Waveform Analog Output

Use the AO control to perform waveform generation operations from one or more analog output channels on a data acquisition device. The waveform generation can be run in a continuous or finite mode. You can configure properties on the control such as the channels used for the waveform generation, the frequency (update clock), and the start condition or trigger. This control is used for applications that require dynamic analog signals, such as testing of analog devices. After you set the properties of the control, the application can perform output operations using a number of simple method calls.

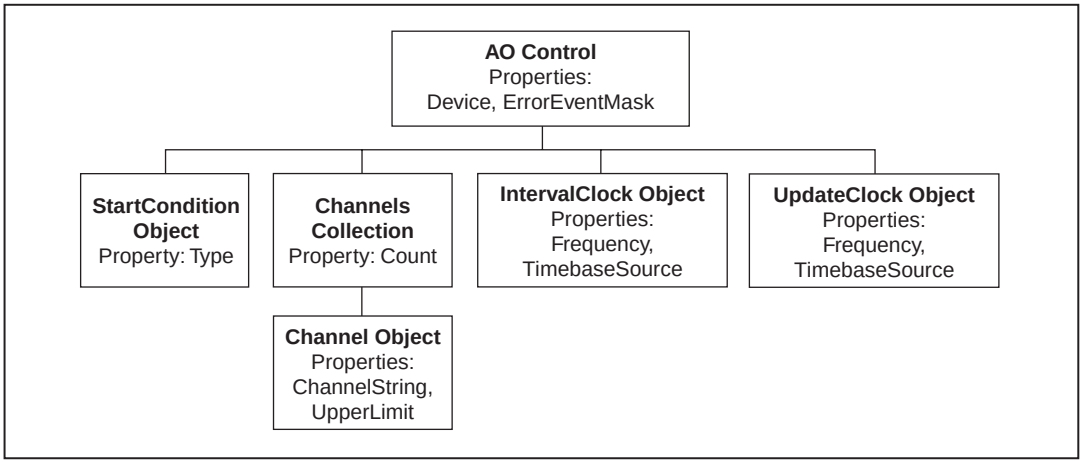

Figure 7-6. AOPoint Control Object Hierarchy (Waveform Analog Output)

The object hierarchy of the AO control separates the functionality of the control into individual objects. The Channels collection and Channel

objects specify the channels and channel attributes used for the signal generation. The StartCondition object specifies when a signal generation starts, and the Clock objects specify the rate of the output.

AO Object

Use the AO object to set default properties for the DAQ controls as well as other properties. The generated waveform data is stored in a buffer in memory. The `NUpdates` property specifies how many updates are stored in the buffer. The `Infinite` property is a Boolean that selects whether a waveform generation runs continuously or stops after a finite number of buffer outputs. In a finite generation, the `NIterations` property specifies how often the data in the buffer is generated.

```
CWAO1.Infinite = False
```

```
CWAO1.NIterations = 10
```

The data buffer usually is stored in computer memory. With certain data acquisition devices, you might be able to store the data in memory on the DAQ device itself to enable faster output of the waveform. You can select this type of operation using the `AllocationMode` property.

While a waveform generation is running, the AO control can generate events to notify you of the progress of the output operation. Use the `ProgressInterval` property to specify the frequency at which these events are generated in number of updates.

The AO control uses the Channels collection and Channel object the same way as the AIPoint control.

Methods and Events

The AO control has a number of methods for performing waveform generation operations, including `Configure`, `Write`, `Start`, and `Reset`. These methods control the output operations. Only the `Write` method requires parameters.

Use the `Configure` method to configure the DAQ driver and hardware with the operation parameters. Use the `Write` method to write an array of voltage data to the buffer in memory before the data can be generated from the analog output channels. Set all other parameters through the properties of the control.

You must call `Configure` before the `Write` and `Start` methods. Use the `Start` method to start the waveform generation that proceeds indefinitely

or stops after the specified number of buffer generations. Use the `Reset` method to stop a continuous generation, unconfigure the AO control, and free any resources reserved during configuration.

You must call the `Configure` method after any of the control properties are changed before they can take effect and after the `Reset` method is used before restarting a waveform generation.

```
Private Sub Start_Click()
   Dim WaveData(0 to 99) As Double
   For i = 0 To 99
      WaveData(i) = Sin(i / 100 * 6.28)
   Next i
   CWAO1.Configure
   CWAO1.Write WaveData
   CWAO1.Start
End Sub
```

```
Private Sub Stop_Click()
   CWAO1.Reset
End Sub
```

The AO control fires the `Progress` event while the output operation is running. The event notifies your application that a specific number of updates has been performed on the output. Set the frequency at which the event is generated in the `ProgressInterval` property. You can use the event to update your front panel to check the progress of the waveform generation.

```
Private Sub CWAO1_Progress(ByVal ScansGenerated As Long)
   Text1.Text = ScansGenerated
End Sub
```

The AO control also has `DAQError` and `DAQWarning` events you can use for error handling.

```
Private Sub CWAO1_DAQError(ByVal StatusCode As Long,
  ByVal ContextID As Long, ByVal ContextDescription
  As String)
   MsgBox "DAQ Error: " + CStr(StatusCode)
End Sub
```

UpdateClock and IntervalClock Objects

The AO control contains both an UpdateClock (commonly used) and IntervalClock (rarely used) object to specify the update rate and optional interval delay. The UpdateClock object specifies the rate at which data

points are generated by the boards. Because each output channel has its own digital-to-analog (DAC) converter, there is no delay between updates from different channels and all channels are updated simultaneously.

You can set the update clock to use an internal frequency, a signal from another component on the DAQ device, or an external signal. The choices of UpdateClock sources depend on the specific DAQ device.

```
CWAO1.UpdateClock.ClockSourceType = cwaoInternalCS
CWAO1.UpdateClock.Frequency = 10000
```

The IntervalClock object is used in a limited number of applications and is supported only on a small number of DAQ devices. When the data is stored completely in FIFO memory on the DAQ device, the IntervalClock object is used in a waveform generation to add a time delay in the output between generations of the waveform data stored in the buffer. With the IntervalClock object, you can store one cycle of a sine wave in the buffer and generate repeated cycles of a sine wave with delays between each. You can examine the properties of the IntervalClock object and their possible settings in the property page.

StartCondition Object

Use the StartCondition object to specify when the waveform generation is started. In most cases, the generation starts immediately after the `Start` method is called. You can use the StartCondition object on some DAQ devices to trigger the generation in response to an external signal or a signal coming from another component on the device. You can use this functionality to synchronize a waveform generation with an input operation or other external process.

The `Type` property selects the overall operation of the object. The value of the `Type` property determines which of the remaining properties on the condition object are used. If the StartCondition object is set to use another signal as the start trigger, the `Source` property specifies the source of the signal. You can use other properties to specify trigger parameters, such as the slope of the signal to trigger on or the conditions of an analog trigger.

You also can set the values of the different StartCondition properties programmatically.

```
CWAO1.StartCondition.Type = cwaoAIStartTrigger
CWAO1.StartCondition.Source = "PFI0"
CWAO1.StartCondition.Level = 5
CWAO1.StartCondition.Mode = cwaoRising
```

If you use a start trigger, the hardware is set to start the waveform generation after the `Start` method is called. The actual output conversions do not start until the specified trigger signal arrives.

Tutorial: Using the AOPoint Control

This tutorial shows how to develop a program using the AOPoint control. To complete this tutorial, your DAQ device must have one or more analog output channels. You can use an analog input channel of your DAQ device to read the output voltage. If your device does not have analog inputs, you can use an external voltmeter or oscilloscope.

This tutorial uses Visual Basic syntax, but the discussion is in general terms so you can follow it in any compatible programming environment. Remember to adjust any code to your specific programming language. Refer to the *Building ComponentWorks Applications* chapters for information about implementing any step in different programming environments. You also can refer to the tutorial examples installed with ComponentWorks for a completed version of this example in several different programming environments.

Designing the Form

1. Open a new project and form. If you are working in Visual C++, select a dialog-based application and name your project `AOPoint`.
2. Load the ComponentWorks User Interface controls (specifically, the Graph and Slide controls) and the Data Acquisition controls (specifically, the AOPoint and AI controls) into your programming environment.

3. Place a ComponentWorks button on the form. In the custom property page change the button style to `On/Off Toggle Button`, its `On Text` to `Stop`, and its `Off Text` to `Start`. In the default property page, change the `Name` property to `Start`.

4. Place a ComponentWorks Slide control on the form and change its name to `UpdateValue`. On the **Numeric** tab of the custom property page, change the **Min** and **Max** properties to `-10` and `10`.

5. Place a ComponentWorks graph on the form. On the **Axis** tab of the custom property page, set the X axis range from `0` to `30` and the Y axis range from `-10` to `10`.

6. If your DAQ device has analog input channels, place a DAQ AI control on the form. In the property page, select the device and channel you want to use. Set the **Number of scans to acquire** to `1`, the **Scan Clock** to `Internal 10 scans/second`, and the **Stop Condition** to `Continuous`.

7. Place a DAQ AOPoint control on the form. In the custom property page of the AOPoint control, select the device and output channel you want to use.

Your form should look similar to the one shown below.

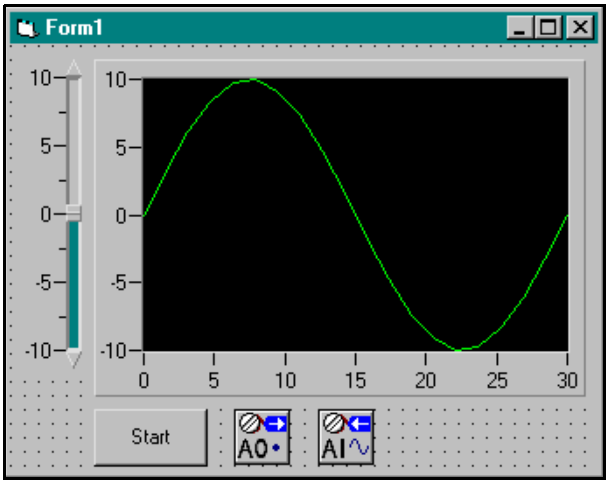

Figure 7-7. AOPoint Form

Developing the Code

The next step is to add the necessary code to generate the analog output and acquire the signal using your analog input channel. If you do not have an analog input channel on your DAQ device, exclude all calls relating to the analog input (starting with CWAI) and use an external voltmeter to measure your analog output.

The program updates the selected analog output channel with the value of the slide when the slide pointer is moved. Use the **Start**/**Stop** button to start and stop a continuous analog input operation that measures the generated voltage.

1. Use the `PointerValueChanged` event of the Slide control to detect any changes and update the analog output value. Use the `SingleWrite` method of the AOPoint control to update the output.

   Create the event handler routine for the `PointerValueChanged` event of the Slide control and add the following code to it. In Visual C++, you first need to create a member variable for the AOPoint control using the ClassWizard.

   ```
   Private Sub UpdateValue_PointerValueChanged(ByVal
     Pointer As Long, ByVal Value As Variant)
       lerr = AOPoint1.SingleWrite(Value, True)
   End Sub
   ```

2. To monitor the voltage being generated, run a continuous analog input operation and chart the acquired voltage on a graph. Start and stop the acquisition with the **Start**/**Stop** button in response to its `ValueChanged` event.

   Create the event handler routine for the `ValueChanged` event of the button and add the following code to it. In Visual C++ you first need to create a member variable for the AI control using the ClassWizard. Depending on the state (`Value`) of the button, the event handler either starts or stops the acquisition.

   ```
   Private Sub Start_ValueChanged(ByVal Value As
     Boolean)
       If Value Then
           CWAI1.Configure
           CWAI1.Start
       Else
           CWAI1.Stop
       End If
   End Sub
   ```

3. While running continuously, the acquisition returns data through the `AcquiredData` event of the AI control. In this event handler routine, chart the data on the graph. The `ChartY` method of the graph acts like the `PlotY` method, except that data is appended to the data already displayed on the graph.

   Create the event handler routine for the `AcquiredData` event of the AI control and add the following code to it. In Visual C++, you first need to create a member variable for the graph control using the ClassWizard.

   ```
   Private Sub CWAI1_AcquiredData(ScaledData As
   Variant, BinaryCodes As Variant)
      CWGraph1.ChartY ScaledData, 1, True
   End Sub
   ```

4. You usually reset hardware operations before quitting the application, which means stopping any ongoing acquisition and resetting the analog output to 0 volts.

   Add the following code to an event handler routine that you call when your application is terminated. The exact name of the event varies depending on your programming environment. In Visual Basic, you can use the `Terminate` event of the `Form` object.

   ```
   Private Sub Form_Terminate()
      CWAOPoint1.SingleWrite 0, True
   End Sub
   ```

5. Save your project and form as `AOPoint`.

Testing Your Program

1. Run the program. Remember to physically connect the analog output to your analog input. When you toggle the **Start** button, the continuous input operation starts and displays its measurement on the graph.
2. Change the value of the analog output by moving the slide. When you move the slide, the analog output is updated and you can see the change on the graph.

Your completed running program should look similar to the one shown below.

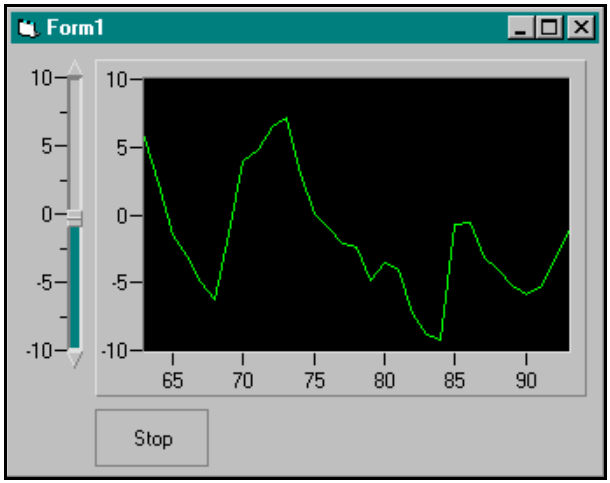

Figure 7-8. Testing AOPoint

When you end the program, the analog output is automatically reset to zero volts.

Digital Controls and Hardware

Three ComponentWorks DAQ controls perform digital input and output operations. Use the DIO control for both input and output single point operations to update the state of any output lines or read the state of any input lines. Use the DI control to perform buffered digital waveform input operations, and use the DO control to perform buffered digital waveform output (pattern generation) operations.

The digital I/O lines on each data acquisition device are grouped into logical units called *ports*. Although they can have as few as two and as many as 32 lines, most devices have eight lines per port. When referencing digital lines on different controls, always specify a port number (starting with zero per device) to select the lines you want to use. On the DIO control, you can select individual lines of a given port to update or read.

DIO Control—Single Point Digital Input and Output

Use the DIO control to perform single-point updates or reads on the digital lines of a data acquisition device. Typical applications using the DIO control include controlling the state of a physical device such as a valve, relay, or LED or reading the current state of a similar device, such as a switch or light gate. You also can use the DIO control to generate slow pulses to activate other parts of your system. After you set the properties of the DIO control, your program can perform the different operations using simple method calls to the DIO control.

The DIO control consists of a top-level object and a set of Ports and Lines collections and objects. The Ports collection contains a number of Port objects that represent the logical ports on the DAQ device selected in the DIO object. After you select a device in the DIO object, all ports of the device are represented by the control. The Lines collections contains a Line object for each physical digital line on the device. You usually access the Lines collection through one of the Port objects, but you also can access it directly from the DIO object.

Depending on the device you are using, you configure all lines in a given port for input or output operations or you might be able to configure individual lines of a digital port for input and output operations. Devices that allow for line configuration include all E-Series devices (except the extended ports on the MIO-16DE-10), the PC-TIO-10, and DIO-32HS devices.

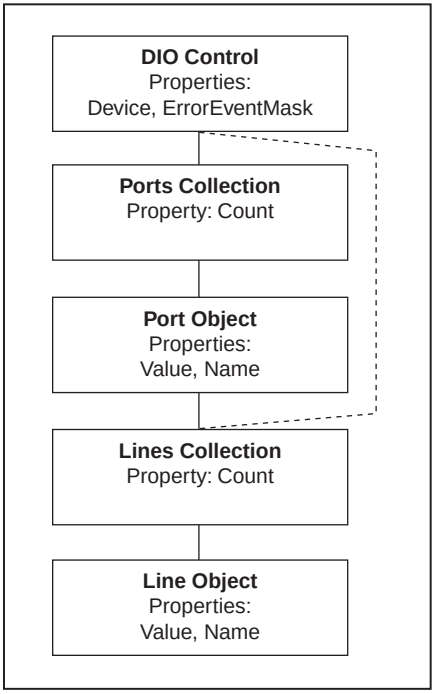

Figure 7-9. DIO Control Object Hierarchy

DIO Object

The DIO object contains an `SCXIChannelString` property. If you use the DIO control with one of the SCXI-116*x* digital modules, enter an SCXI channel string for this property to select the SCXI module. The string must follow the convention for SCXI channel strings as described in the *SCXI Channel Strings* section at the beginning of this chapter. Each SCXI digital module has only one logical port (port 0) that contains all the digital lines of the module. Therefore the port number is always zero.

```
CWDIO1.SCXIChannelString = “sc1!md3!0”
```

The DIO object also contains the Ports collection, which contains individual Port objects. Use the Port object to configure the individual ports on a device, including programmatic configuration of the direction of the digital lines. Each Port object contains a Lines collection with Line objects that you can use to access the individual digital lines of the DAQ device. You also can access the Lines collection directly from the DIO control. Configure the direction of individual ports and lines using the DIO control property pages.

The DIO, Port, and Line objects contain a set of common methods to perform operations using the DIO control.

Ports Collection and Port Object

The Ports collection contains the common properties and methods of a collection such as `Count` and `Item`. Use the `Item` method to access the individual Port objects.

```
CWDIO1.Ports.Item(1)
```

Use the Port object to configure the individual digital ports on a DAQ device. Port objects include properties such as `Assignment` and `LineDirection`. Use these two properties to configure the direction of the port or the lines in the port programmatically. The `Assignment` property specifies whether a port is configured for input or output operations or is line configured. In line configuration, use the `LineDirection` property to specify the direction of each individual line in the port. Each bit in the `LineDirection` property corresponds to a digital line. For example, bit 0 corresponds to line 0. A bit value of 1 indicates an output line and a 0 indicates an input line.

```
'Configure port 2 for output
CWDIO1.Ports.Item(2).Assignment = cwdioPortOutput
'Configure port 0, lines 0-3 for output, lines 4-7 for
input, binary 00001111 = decimal 15
CWDIO1.Ports.Item(0).Assignment =
  cwdioPortLineConfigured
CWDIO1.Ports.Item(0).LineDirection = 15
```

Note ***The MIO-16DE-10 is a hybrid DIO board:***

- ***Port 0 is 8 bits wide and line configurable.***
- ***Port 1 does not exist.***
- ***Ports 2, 3, and 4 are each 8 bits wide and port configurable.***

Each Port object also contains a reference to a Lines collection containing individual Line objects. Use these to configure and operate the individual digital lines of the data acquisition device.

Lines Collection and Line Object

The Lines collection has the same properties as the Ports collection and with it you can select individual Line objects. The Line object has a `Name` property that you can use to identify individual lines by name.

```
CWDIO1.Ports.Item(0).Lines.Item(0).Name = "Switch1"
```

The `Assignment` property of the Line object is read only. Use it to check the current direction of a particular line.

```
Line0isOutput =
  CWDIO1.Ports.Item(0).Lines.Item(0).Assignment
```

Common Properties and Methods

The DIO, Port, and Line objects have a set of common methods (`SingleRead`, `SingleWrite`, and `Update`) to perform input and output operations. In addition, the Port and Line objects include a `Value` property that you use in conjunction with the `Update` method.

Use either the `SingleRead` or `SingleWrite` method to read or write the current state of the digital I/O lines. `SingleRead` requires as a parameter a variant that returns the value or values from the read operation. Performing `SingleRead` on the DIO object returns an array of integers where each array element represents the state of one port. The integer represents the state of the digital lines in a port by bit, where the lowest bit in the integer represents the state of line 0 in the port and so on. For example an integer of value 25 (binary 00011001) indicates that the state of lines 0, 3, and 4 are high and all the remaining lines are low. Performing `SingleRead` on a Port object returns a single integer representing the state of the port and performing `SingleRead` on a Line object returns a Boolean value indicating the state of the line.

```
Dim vData As Variant
CWDIO1.SingleRead vData
‘Returns an array of integers
```

```
CWDIO1.Ports.Item(0).SingleRead vData
‘Returns an integer
```

```
CWDIO1.Ports.Item(0).Lines.Item(0).SingleRead vData
‘Returns a Boolean
```

You also can access the Lines collection directly from the DIO object. If you have assigned a name to a specific digital line, you can use it as a reference.

```
CWDIO1.Lines.Item("Switch1").SingleRead bState
```

```
CWDIO1.Lines.Item(14).SingleRead bState
```

To write to the digital output lines, use the `SingleWrite` method on the DIO, Port, or Line object. This method requires you to write a parameter containing the data to the digital lines. The form of the parameters is the same as it would be returned from a corresponding `SingleRead` call.

```
Dim vData As Variant
```

```
'An array of integers to write to the DIO object
vData = Array(0, 0, 0)
CWDIO1.SingleWrite vData
```

```
'A single integer to write to a port
vData = 0
CWDIO1.Ports.Item(0).SingleWrite vData
```

```
'A Boolean to write to a line
vData = False
CWDIO1.Ports.Item(0).Lines.Item(0).SingleWrite vData
```

As an alternative to reading and writing using the digital lines, use the `Value` property of the Port or Line object and the `Update` method on the DIO, Port, or Line object.

In the output direction, use the `Value` property to specify the value of a port or line that will be written to the hardware the next time you call the `Update` method. On the input side, the `Value` property represents the state of the hardware lines the last time the `Update` method was called.

Note ***The*** `Value` ***property does not represent the current state of the digital lines.***

The `Value` property represents a cached state of either the last input operation or the data that will be written on the next output operation. Use the `Update` method to synchronize the cached values with the hardware. This allows you to assign new values to individual output lines and update

all of them at once. You can cache the current state of a set of input lines once and then read the values from the individual lines.

```
'Input Operation
CWDIO1.Update
portVal = CWDIO1.Ports.Item(0).Value
lineVal = CWDIO1.Ports.Item(0).Lines.Item(4).Value

'Output Operation
CWDIO1.Ports.Item(0).Lines.Item(6).Value = False
CWDIO1.Ports.Item(0).Value = newPortVal
CWDIO1.Update
```

DI Control—Buffered Waveform Digital Input

Use the DI control to perform buffered waveform digital input operations. You can acquire data from your digital inputs quickly, at a rate specified by an external signal or internal frequency. Typical applications using the DI control include transferring digital data from an external device or monitoring a quickly changing system. Advanced applications might include network analyzers. After the properties of the DI control are set, your program can perform the digital acquisition using simple method calls to the DI control.

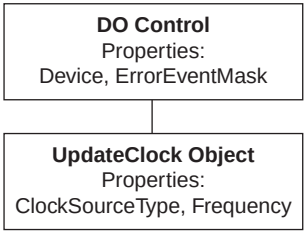

Figure 7-10. DI Control Object Hierarchy

The DI control consists of a top-level object and the UpdateClock object. Most properties that determine the actions of the DI control are set on the control itself, while the UpdateClock object determines the source for the rate of acquisition.

Many DAQ devices do not support buffered digital acquisition, and the DI control does not work with these devices. Other devices support only single buffered acquisition. For these devices, you cannot use the continuous mode. Most of these devices also require an external signal to set the acquisition rate, and you cannot just specify an internal frequency. Devices in the DIO-32 series support both continuous acquisitions and internal frequency sources and can take advantage of the full range of the DI control

features. Check the hardware manual of your data acquisition device for detailed information about its capabilities.

DI Object

After setting the default DAQ properties of the DI object (such as `Device`), configure the `ChannelString` property to specify which digital ports are used by the DI control. You can enter the number of the port to be used or list the ports separated by commas. Make sure the ports you specify support the desired operation. Not all ports on a device support buffered digital input. For example, the DIO-24 devices have three ports, but only ports 0 and 1 can be used with the DI control. Consult your hardware manual for specific information.

You can run the DI control in either continuous or single buffer mode. In continuous mode, data is acquired until the operation is explicitly stopped. The single buffered operation acquires a preset number of points and stops. Use the `Continuous` property to select between these two different modes.

```
CWDI1.Continuous = True
```

The `NPatterns` property specifies the size of the acquisition buffer, which also equals the number of patterns acquired per port in a single buffer acquisition. A *pattern* is a value acquired from a digital port that is a numeric representation of the state of the digital lines in the port. `NPatterns` is specified in number of patterns per port in the channel list. For example, if `NPatterns` is set to `1000` and there are two ports in `ChannelString`, the buffer contains 2000 pattern values. When the control is ready to return data to the application, it fires the `AcquiredData` event and returns the patterns in an array.

While the acquisition is running, you have the option to receive progress events from the DI control. Set the frequency of the `Progress` event in the `ProgressInterval` property as number of patterns acquired. `ProgressInterval` must be less than or equal to `NPatterns`. A zero value disables the `Progress` event generation. You can have the acquired patterns returned with the event using the `ProgressReturnData` Boolean property.

```
CWDI1.NPatterns = 1000

CWDI1.ProgressInterval = 100

CWDI1.ProgressReturnData = False
```

UpdateClock Object

Use the UpdateClock object to set the rate used to acquire the digital pattern into the buffer. Depending on the data acquisition device you are using, you might need to supply a clock signal to the device, or you might be able to select an internal rate by frequency or period. Check your hardware manual to see what clock sources are available for your device.

The `ClockSourceType` property sets the source for the clock signal, which might be the I/O connector of the device, the RTSI bus, or an internal clock. Depending on this value, you can use other properties to further specify the clock. If you are using an internal clock, set the acquisition rate with the `InternalClockMode` and `Frequency` or `Period` properties.

```
CWDI1.UpdateClock.ClockSourceType = cwdioCSInternalClock
```

```
CWDI1.UpdateClock.InternalClockMode = cwdioFrequency
```

```
CWDI1.UpdateClock.Frequency = 1000
```

If you use the I/O connector or RTSI bus, you do not need to set any other properties.

Methods and Events

You normally operate the DI control using its `Configure`, `Start`, and `Reset` methods. None of these methods require any parameters. After you set the properties of the control, call the `Configure` method to program the driver and hardware with the current property values. Next, call the `Start` method to initiate the acquisition.

Use the `Reset` method during a continuous acquisition to stop the acquisition and to release any resource assigned in the `Configure` call. If the control was not reset since the last acquisition, you can perform another acquisition without calling `Configure` again. If any property values are changed, you need to call `Configure` to implement those changes.

```
Private Sub DigStart_Click()
   CWDI1.Configure
   CWDI1.Start
End Sub
```

```
Private Sub DigStop_Click()
   CWDI1.Reset
End Sub
```

Note ***If the digital acquisition requires an external signal as a clock, you must physically connect this signal to the proper line of the I/O connector of your data acquisition card. Consult your hardware manual to determine where to connect this signal.***

The acquisition can generate two types of events. The `AcquiredData` event is generated at the completion of a single buffer acquisition or at `NPatterns/2` intervals of a continuous acquisition. The `AcquiredData` event is the main mechanism for returning the acquired data to the application. Data from a one-port acquisition is returned in a one-dimensional array; data from a multiport acquisition is returned in a two-dimensional array.

```
Private Sub CWDI1_AcquiredData(Waveform As Variant)
   CWGraph1.PlotY Waveform, 0, 1, True
End Sub
```

If the `ProgressInterval` property has a value other than zero and it returns data if you set the `ProgressReturnData` property to `True`, the `Progress` event is generated. Use the `Progress` event to retrieve data during an acquisition or to show the progress of an acquisition on the user interface.

```
Private Sub CWDI1_Progress(ByVal TotalPatterns As Long,
  Waveform As Variant)
   'Show percent complete
   CWSlide1.Value = TotalPatterns / CWDI1.NPatterns * 100
   Text1.Text = TotalPatterns
End Sub
```

In programming environments that do not support event handler routines, you can call the `AcquireData` method to perform a synchronous acquisition with single buffered operations.

```
Private Sub SynchAcq_Click()
   Dim waveform As Variant
   CWDI1.Configure
   CWDI1.AcquireData waveform, 5
   CWGraph1.PlotY waveform, 0, 1, True
End Sub
```

DO Control—Buffered Waveform Digital Output

Use the DO control to perform buffered waveform digital output operations. With the DO control, you can generate a digital stream from the digital outputs at a rate specified by an external signal or internal frequency. Typical applications using the DO control include generating digital test

signals for testing of electronic devices, systems, and networks. You also can use the DO control to transfer binary data from a computer to another computer or other device. By building your application after setting the properties of the DO control, your program can perform the digital waveform generation using method calls to the DO control.

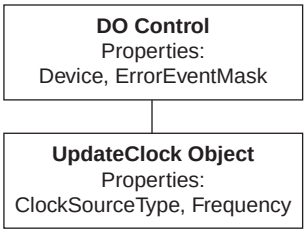

Figure 7-11. DO Control Object Hierarchy

The DO control consists of a top-level object and UpdateClock object. You can set most properties that determine the actions of the DO control in the control itself, while the UpdateClock object determines the source for the rate of generation.

Many DAQ devices do not support buffered digital generation and the DO control does not work with these devices. Other devices support only single buffered generation, and you cannot use the continuous mode in the DO control. Most of these devices also require an external signal to set the update rate, and you cannot simply specify an internal frequency. Devices in the DIO-32 series support both continuous acquisitions and internal frequency sources and can take advantage of the full range of the DO control features. Check the hardware manual of your data acquisition device for detailed information about its capabilities.

DO Object

After setting the default DAQ properties of the DO object, configure the `ChannelString` property, which specifies the digital ports used by the DO control. Enter the number of the port to be used or list the ports separated by commas. Make sure the ports you specify support the desired operation. Not all ports on a device support buffered digital input. For example, the DIO-24 devices have three ports, but only ports 0 and 1 can be used with the DO control. Port 2 is used to connect the timing signals required for these operations. Consult your hardware manual for specific information.

You can run the DO control in either continuous or single buffer mode. In the continuous mode, data is generated until you explicitly stop the

operation. The single buffer operation generates a preset number of data values stored in the control buffer once and stops. Use the `Continuous` property to select between these two different modes.

```
CWDO1.Continuous = True
```

The `NPatterns` property specifies the size of the control buffer used to store the output data. A *pattern* is a value generated from a digital port numerically representing the state of the digital lines in the port. `NPatterns` is specified in number of patterns per port in the channel list. For example, if `NPatterns` is set to `1000` and there are two ports in `ChannelString`, the buffer contains 2000 pattern values.

While the generation occurs, the control fires the `Progress` event to indicate the status of the operation. By default, the `Progress` event is only generated at the completion of a single buffer operation or at the end of every completed half-buffer in a continuous operation. You can set the frequency of the `Progress` event by disabling the `AutoSelectProgressInterval` property and specifying the `ProgressInterval` property in number of patterns acquired. `ProgressInterval` must be less than `NPatterns`, and a zero value disables the progress event generation.

```
CWDO1.NPatterns = 1000
CWDO1.AutoSelectProgressInterval = False
CWDO1.ProgressInterval = 100
```

UpdateClock Object

Use the UpdateClock object to set the update rate of the digital waveform generation. Depending on the data acquisition device, you might need to supply a clock signal to the device or you might be able to select an internal rate by frequency or period. Check your hardware manual to see what clock sources are available on your device.

The `ClockSourceType` property sets the source for the clock signal, which might be the I/O connector of the device, the RTSI bus, or an internal clock. Depending on this value, you can use other properties to further specify the clock. If you are using an internal clock, use the `InternalClockMode` and `Frequency`, `Period`, or `Timebase` properties to set the update rate.

```
CWDI1.UpdateClock.ClockSourceType = cwdioCSInternalClock
CWDI1.UpdateClock.InternalClockMode = cwdioFrequency
CWDI1.UpdateClock.Frequency = 1000
```

If you use the I/O connector or RTSI bus, you do not need to set any other properties.

Methods and Events

You normally operate the DO control using its `Configure`, `Write`, `Start`, and `Reset` methods. Only the `Write` method requires parameters. After you set the properties of the control, call the `Configure` method to program the driver and hardware with the current property values. Next, call the `Write` method to send data to the control buffer for output. Data should be passed as a one-dimensional array of pattern values for a single port output and a two-dimensional array of patterns for a multiport output. You then call the `Start` method to initiate the output operation.

In a single buffer output, the generation stops after it completes the buffer. In a continuous generation, use the `Reset` method to stop the output and release any resource assigned in the `Configure` call. If the control was not reset since the last generation, you can perform another output without calling `Configure` again. You must call `Configure` to implement any changes you make to the property values.

```
Private Sub DigStart_Click()
   Dim data(0 to 1, 0 to 99) '2-port output
   For i = 0 To 99
      data(0, i) = i 'Port 1 data
      data(1, i) = Int(Rnd * 256) 'Port 2 data
   Next i
   CWDO1.Configure
   CWDO1.Write data
   CWDO1.Start
End Sub
```

```
Private Sub DigStop_Click()
   CWDO1.Reset
End Sub
```

Note ***If your digital waveform generation requires an external signal as a clock, you must physically connect this signal to the proper line of the I/O connector of your data acquisition card. Consult your hardware manual to determine where to connect this signal.***

You also use the `Write` method during a continuous output to send new data to the buffer so that it can be output from the digital lines. This new data is usually written in half buffer sizes to the control. Use the `Progress` event to determine when to write new data to the DO control. In continuous mode, set the `AllowRegeneration` property to `False` to prevent the DO control from generating the same data in the buffer twice.

During the digital output operation, the DO control generates a `Progress` event to notify you of the completion of an output or to allow you to monitor its progress. In a continuous generation, the `Progress` event is set to fire after half the buffer is generated to prompt you to send new data to the control. Although only half the buffer is updated at a time, data is written to the control using the `Write` method.

```
Private Sub CWDO1_Progress(ByVal TotalPatterns As Long)
    Dim data(0 to 1, 0 to 49)
    For i = 0 To 49
        data(0, i) = i
        data(1, i) = Int(Rnd * 256)
    Next i
    CWDO1.Write data
End Sub
```

Tutorial: Using the DIO Control

This tutorial shows you how to develop a simple program that uses the DIO control to perform single point digital inputs and outputs. Your DAQ device must have one or more digital ports to use this example. You also need a way to apply signals to the digital lines if you want to perform an input or a way to display the state of any output lines. If you have two digital ports, you can connect all the lines from one to the other so that the input can read the state of the output lines.

This tutorial uses Visual Basic syntax, but the discussion is in general terms so you can follow it in any compatible programming environment. Remember to adjust any code to your specific programming language. Refer to the *Building ComponentWorks Applications* chapters for information about implementing any step in different programming environments. You also can refer to the tutorial examples installed with ComponentWorks for a completed version of this example in several different programming environments.

Designing the Form

1. Open a new project and form. If you are working in Visual C++, select a dialog-based application and name your project `DIO`.
2. Load the ComponentWorks User Interface controls (specifically, the Numeric Edit Box) and the Data Acquisition controls (specifically, the DIO control) into your programming environment.

3. Place two buttons on the form. Change their captions and names according to the following table.

Button	Name	Caption
CWButton1	Input	Read Port
CWButton2	Output	Write Port

4. Place two ComponentWorks Numeric Edit Box controls on the form and change their names to `InputNum` and `OutputNum`.

5. Place two DIO controls on the form. Use the first one (CWDIO1) for your input operations, and the second (CWDIO2) for any output operations. You configure the DIO controls in the next section of this tutorial.

Your form should look similar to the one shown below.

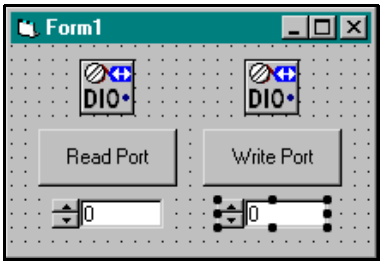

Figure 7-12. DIO Form

Developing the Code

Configure the DIO controls and add the necessary code to write an update to the output lines or read the state of the input lines. If you use only the input control or only the output control, follow the directions for the appropriate task. Although there are ports on some boards that can be configured only as one or the other, most digital ports can be configured for either input or output.

1. Configure one DIO control for input and one for output.

   Input: In the custom property page of the first DIO control, CWDIO1, select the appropriate DAQ device. Then select (highlight) one digital port in the list and set its direction (Port assignment) to `Input`. Set the port assignment for all other ports to `Unused`.

   Output: In the custom property page of the second DIO control, CWDIO2, select the appropriate DAQ device. Then select one digital port in the list and set its direction (Port assignment) to `Output`. Set the port assignment for all other ports to `Unused`.

   Depending on your device, you might be able to select the same port for both controls. You can write to the port using one control and read back a value with the other control without making any external connections.

2. Use the **Read Input** button to read the state of the digital input port and display the pattern value in the numeric edit control. You can read the state of all ports at once or you can read a specific port using the DIO control. In this example, read only the selected port to make the code more independent of the hardware you are using.

   Create an event handler for the `Click` event of the **Input** button and add the following code. In the `Item` method, use the number of the digital port you configured for input. In this example, port 1 is used. The first port on most boards is port 0.

```
Private Sub Input_Click()
    Dim data As Variant
    CWDIO1.Ports.Item(1).SingleRead data
    InputNum.Value = data
End Sub
```

Note ***If you read all ports at once, the data is returned as an array and you must access individual array elements for the information, as in the following code. Do not add this code to your example.***

```
Dim data As Variant
CWDIO1.SingleRead data
InputNum.Value = data(1)
```

3. Use the **Write Output** button to set the state of the digital output port and display the pattern value in the numeric edit control. You can write the state of all the ports at once or you can write to a specific port using the DIO control. In this example, write only to the selected port to make the code more independent of the hardware you are using.

Create an event handler for the `Click` event of the **Output** button and add the following code. In the `Item` method, use the number of the digital port you configured for output. In this example, port 0 is used, which is the first port on most boards.

```
Private Sub Output_Click()
   Dim data As Variant
   data = OutputNum.Value
   CWDIO2.Ports.Item(0).SingleWrite data
End Sub
```

Note ***If you want to write to all ports at once, you must pass in an array of data with a value for each port on the device, as in the following code. Do not add this code to your example.***

```
Dim data As Variant

'For a device with one port
data = OutputNum.Value

'For a device with three ports
data = Array(OutputNum.Value, 0, 0)
CWDIO2.Ports.Item(0).SingleWrite data
```

4. Save your project and form as `DIO`.

Testing Your Program

Before you can run your program, connect a signal source to your input port and a display or sensor to the output port. You can connect the output lines to the input lines and use the input to measure the output.

1. Run the program. Write a value to the output port by setting a value in the corresponding numeric edit box and pressing the **Write Output** button. The value in the numeric edit box represents the pattern written to the port. The pattern is a numeric (integer) representation of the bits in the digital port. A value of zero means all lines are written low. A value of 22 (binary 00010110) means lines 1, 2 and 4 are written high. Most ports have 8 digital lines, and a value of 255 corresponds to all lines high.
2. Read the value of the input port by pressing the **Read Input** button. The pattern value is read from the port and displayed in the corresponding numeric edit box. If you have connected all lines of the output port to the input port, you should read the same value you wrote out. If you read input lines that are not connected to anything (floating), their values are not necessarily low. An open port might read any value, not always 0.

3. Quit the program.

Instead of using the `SingleRead` and `SingleWrite` methods, you also can use the `Value` property and `Update` method of the DIO control to implement the functionality of this tutorial. The `Update` method can be called on the DIO control as a whole or on the individual port.

```
Private Sub Input_Click()
    Dim data As Variant
    CWDIO1.Update
    data = CWDIO1.Ports.Item(1).Value
    InputNum.Value = data
End Sub
```

```
Private Sub Output_Click()
    Dim data As Variant
    data = OutputNum.Value
    CWDIO2.Ports.Item(0).Value data
    CWDIO2.Ports.Item(0).Update
End Sub
```

Counter/Timer Hardware

The hardware component on the DAQ device used by the Counter and Pulse controls is called a counter/timer. Use this component to count or measure incoming digital pulses or to generate digital pulses and pulse trains on its output.

Each counter/timer has two inputs labeled *Source* and *Gate* and one output labeled *Out*. The Source is also referred to as *Clock*. The counter counts the number of digital pulses coming in on the source input. The counting operation can be gated by a digital signal applied to the gate input.
The output of the counter generates a pulse when the counter reaches its maximum count or zero, depending on whether it is counting up or down. Vary parameters, such as the signals applied to the source and gate, and the initial count of the counter and the mode used in gating to use this simple component in a variety of different applications.

The ComponentWorks Counter and Pulse controls allow you to select from a number of standard operations and specify applicable properties. The properties in the controls are grouped by the I/O point of the counter that they affect, such as source, gate, and output.

Counter Control—Counting and Measurement Operations

Use the Counter control to perform counting and other measurement operations using the counter/timer components on a data acquisition device. Typical operations include counting a number of events, measuring the period of an unknown pulse, or measuring the frequency of a signal. After the properties of the control are set, the application can perform the operation using a simple method call to the Counter control. The operation of the Counter control is measurement-oriented, and the behavior of the control depends on the type of measurement selected.

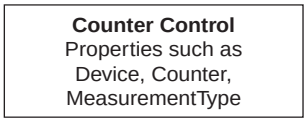

Figure 7-13. Counter Control Object Hierarchy

The Counter control consists of only one object—the Counter object—which contains all the properties and methods necessary to use the control.

Counter Object

The Counter object contains all the properties necessary to configure a counting or measurement operation.

The `Counter` and `MeasurementType` properties affect the Counter control as a whole. The `Counter` property specifies which counter/timer on the DAQ device will be used by the control. The available counter numbers depend on the DAQ device. The `MeasurementType` property selects the type of operation the control will perform and affects the interpretation of the remaining properties. The `InitialCount` property sets the value of the counter at the start of an operation.

Table 7-1 describes different measurement types.

Table 7-1. Measurement Types

Measurement Type	Description	Units	One Point	Buffered (DAQ-STC Only)
Event Count	Counts the number of pulses on the source input.	Count	X	X
Time	Measures time by counting a known clock input.	Sec	X	X
Frequency	Measures frequency on source input by counting the number of pulses in a known period of time.	Hertz	X	
Pulse Width, High	AM9513: Measures the length of the high pulse until the next falling edge. A partial measurement is possible if the pulse is initially high. DAQ-STC: Measure first pulse after next falling edge.	Sec	X	X
Pulse Width, Low	AM9513: Measures the length of the low pulse until the next rising edge. A partial measurement is possible if the pulse is initially low. DAQ-STC: Measures the first pulse after the next rising edge.	Sec	X	X
Pulse Width, High Single-Shot	AM9513: Measures the length of the high pulse until the next falling edge. A partial measurement is possible if the pulse is initially high. DAQ-STC: Measures the first complete pulse and generates an error if the pulse is in a high state at the start.	Sec	X	
Pulse Width, Low Single-Shot	AM9513: Measures the length of the low pulse until the next rising edge. A partial measurement is possible if the pulse is initially low. DAQ-STC: Measures the first complete pulse and generates an error if the pulse is in a low state at the start.	Sec	X	
Period, Rising Edge	Measures the period between two rising edges on gate input by counting a known clock.	Sec	X	X
Period, Falling Edge	Measures the period between two falling edges on gate input by counting a known clock.	Sec	X	X
Semi-Periods	Measures the length of consecutive high and low pulses.	Sec		X

Frequency and Single-Shot Pulse Width measurements can be performed as one point (unbuffered) operations only, but the Semi-Period measurement must be buffered.

The remaining Counter object properties directly affect one of the inputs or outputs of the control. Properties affecting the source input include `TimebaseSource`, `TimebaseSignal`, `SourceEdge` and `CountDirection`.

- `TimebaseSignal`—Selects either a specific input pin of your board or the internal frequency used for the source of the counter, depending on the `TimebaseSource` property.
- `SourceEdge`—Selects whether to count rising or falling edges on the source.
- `CountDirection`—Determines whether the counter counts up or down.

Gate properties are `GateMode, GateSource`, `GateSignal`, and `GateWidth`.

- `GateMode`—Selects the type of gating used in an operation such as No Gating, High Level Gating, and so on.
- `GateSource` and `GateSignal`—Select the source of the gate signal.
- `GateWidth`—Sets the sample width of the measurement in Frequency measurements.

Output properties are `OutputMode` and `Polarity`.

- `OutputMode`—Determines whether the counter output pulses or toggles when the counter reaches its limit.
- `Polarity`—Determines if the output is high (active high) or low (active low) polarity.

To become familiar with different types of measurements and properties, browse through the custom property pages and study the different settings. Some properties are disabled depending on the state of other properties.

Methods and Events

The Counter control has a set of simple methods to control the operation of the counter. Call the `Configure` method to configure the data acquisition driver and hardware with the current properties before starting any operation. Call the `Start` method to start the measurement. If needed, you can call the `Stop` method to stop the counter and the `Reset` method to unconfigure the counter and release its resources for other operations. After stopping a counter, you can restart it using just the `Start` method if you

have not called the `Reset` method. After resetting the counter, you must call `Configure` again before restarting.

```
Private Sub CounterStart_Click()
    CWCounter1.Configure
    CWCounter1.Start
End Sub
```

```
Private Sub CounterStop_Click()
    CWCounter1.Stop
    CWCounter1.Reset
End Sub
```

Depending on the measurement, the control might fire an `AcquiredData` event to indicate the completion of a measurement and return the data. The data is returned in the `Measurement` parameter that is passed to the `AcquiredData` event and scaled to the units indicated in the previous table. Measurements that fire an `AcquireData` event are Frequency, Pulse Width, Period, and Semi-Period.

```
Private Sub CWCounter1_AcquiredData(Measurement As
  Variant, ByVal Overflow As Boolean)
    txtPulseWidth.Text = Measurement
End
```

Event Count and Time measurements do not fire an event to return data. Therefore, use the `ReadCounter` and `ReadMeasurement` methods to read the value of your measurement. `ReadCounter` returns the actual count value of the counter and usually is used in event counting operations. `ReadMeasurement` returns the value scaled to the appropriate units, such as time. Both methods require one parameter to return the measurement and another to indicate an overflow condition in the counter.

```
Private Sub Timer1_Timer()
    Dim lVal As Long
    Dim vVal As Variant
    Dim bOverflow As Boolean
    CWCounter1.ReadCounter lVal, bOverflow
    txtCount.Text = lVal
    CWCounter1.ReadMeasurement vVal, bOverflow
    txtMeas.Text = vVal
End Sub
```

Buffered Measurements

All measurements acquire and return a single value of the selected type. On specific data acquisition devices such as the E-Series devices, you can perform a buffered measurement for most of the measurement types. A buffered measurement acquires multiple values and stores the individual values in a buffer for later analysis and processing. After they are started, measurements are stored in the buffer asynchronously. When all data points have been acquired, the `AcquiredData` event is fired and the buffer is returned to the application. Measurements are stored in the buffer at the completion of a period or pulse for Period and Pulse measurements or at a conversion signal on the gate input for Event Count and Time measurements.

The Counter object has two properties to enable buffered measurements. `UseBuffering` is a Boolean property that enables the buffered mode. `NMeasurements` specifies the number of measurements. While a buffered measurement is running, no data can be read from the counter or the buffer, and all data is returned when the measurement is complete. The same methods used for a one-point operation also are used to control buffered measurements. When data is returned in the `Measurement` parameter of the `AcquiredData` event, it is formatted as an array.

```
Private Sub BufferedStart_Click()
    CWCounter1.MeasurementType = cwctrHiPulseWidth
    CWCounter1.UseBuffering = True
    CWCounter1.NMeasurements = 10
    CWCounter1.Configure
    CWCounter1.Start
End Sub
```

```
Private Sub CWCounter1_AcquiredData(Measurement As
  Variant, ByVal Overflow As Boolean)
    CWGraph1.PlotY Measurement, 0, 1, True
End Sub
```

Pulse Control—Digital Pulse and Pulsetrain Generation

Use the Pulse control to generate individual pulses and pulse trains with the counter/timer component on a data acquisition device. Typical operations include generating a digital test signal or driving a stepper motor. After you set the properties of the control, the application can perform operations using simple method calls to the Pulse control. The operation of the Pulse control is task-oriented, and the behavior of the control depends on the type of task selected in the properties.

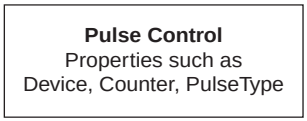

Figure 7-14. Pulse Control Object Hierarchy

The Pulse control consists of only one object—the Pulse object—which contains all the properties and methods necessary to use the control.

Pulse Object

The Pulse object contains the properties necessary to configure the pulse operations.

The `Counter` and `PulseType` properties affect the Pulse control as a whole. The `Counter` property specifies the counter/timer on the DAQ device used by the control. The available counter numbers depend on the type of DAQ device. The `PulseType` property selects the type of operation the control performs and affects the interpretation of the remaining properties.

Table 7-2 describes the different pulse type operations.

Table 7-2. Pulse Type Operations

Pulse Type	Description
Single Pulse	Generates one pulse according to the specifications.
Continuous Pulse Chain	Generates a continuous pulse train according to the specifications.
Finite Pulse Chain	Generates a finite set of pulses. This requires two counters.
FSK Pulse	Frequency Shift Keying, generates one of two different pulse trains dependent on a digital input. The selector is applied to the gate of counter.
Incremental Delay Pulse (ETS)	Equivalent Time Sampling pulse. Generates a series of individual pulses with increasing offset from the digital trigger. The trigger is applied to the gate of counter.
Retriggerable Pulse, Rising Edge	Retriggered single pulse. The trigger is applied to the gate, triggered off the rising edge.
Retriggerable Pulse, Falling Edge	Retriggered single pulse. The trigger is applied to the gate, triggered off the falling edge.

The remaining properties of the Pulse object directly affect the pulse generation or the gate of the counter. Properties affecting pulse generation include `ClockMode`, `CountDirection`, `DutyCycle`, `Frequency`, `Period`, `Period2`, `Phase1`, `Phase2`, `Frequency2`, `Count`, `Phase1Inc`, `PulseDelay`, `PulseWidth`, `SourceEdge`, `TimebaseSource`, and `TimebaseSignal`.

The parameters that are used for the actual pulse generation depend on the settings of the `PulseType` and `ClockMode` properties. The `PulseType` settings select the general operation of the Pulse control, such as single pulse or continuous pulse generation, while the `ClockMode` settings determine how the output is characterized. Settings for `ClockMode` include `Frequency` and `Period`, which means the output is characterized by its frequency or by the period of the pulse.

Gate properties include `GateMode`, `GateSource`, `GateSignal`. `GateMode` selects the type of gating used in an operation such as No Gating, High Level Gating, and so on. `GateSource` and `GateSignal` select the source of the gate signal.

To become familiar with different types of pulse generation and their corresponding properties, browse through the custom property pages and study the different settings. Some properties are disabled depending on the state of other properties.

Methods

The Pulse control has a set of simple methods to control the operation of the pulse generation. Use the `Configure` method to configure the data acquisition driver and hardware with the current properties before starting any operation. Use the `Start` method to start the generation. If needed, you can call the `Stop` method to stop the counter and the `Reset` method to unconfigure the counter and release its resources for other operations. After stopping a pulse generation, you can restart it using just the `Start` method if you have not called the `Reset` method. After resetting the counter, you must call `Configure` again before restarting.

```
Private Sub PulseStart_Click()
    CWCounter1.Configure
    CWCounter1.Start
End Sub
```

```
Private Sub PulseStop_Click()
    CWCounter1.Stop
    CWCounter1.Reset
End Sub
```

While running a pulse generation operation, you can change some parameters without stopping the operation. For example, you can change the frequency of continuous pulse generation in mid-stream by updating the relevant property of the pulse control and calling the `Reconfigure` method.

`Reconfigure` can be called only when an operation is running because it updates all properties that have changed since the last call to `Configure` or `Reconfigure`. Use this method with the `PointerValueChanged` event of the Slide control to set the pulse frequency to the value of the slide.

```
Private Sub CWSlide1_PointerValueChanged(ByVal Pointer
  As Long, Value As Variant)
    CWPulse1.Frequency = Value
    If OutputRunning Then CWPulse1.Reconfigure
End Sub
```

FSK and ETS Pulse Generation

On data acquisition devices that use the DAQ-STC counter/timer, the Pulse control supports two specialized pulse generation modes, which you can select in the `PulseType` property. Use these modes in conjunction with analog input operations to perform special acquisitions.

Use FSK (frequency shift keying) to generate a continuous pulse with one of two different frequencies, depending on a separate digital signal. Apply the digital signal that selects between the two different frequencies to the gate input of the counter used by the pulse control. Using the `Frequency` and `Frequency2`, `Period` and `Period2`, or `Phase1` and `Phase2` properties, you can select the two different frequencies to generate. You can combine this type of pulse generation with an analog input to build a Rate-Change-on-the-Fly acquisition.

To change the acquisition rate dynamically with a digital signal, use the output of the counter as the scan clock of the acquisition. If you have a hardware analog trigger on your device, you can enhance this application by using the analog trigger to convert an analog signal, such as the one being acquired, into a digital signal, which you then use to control the pulse generation. This means you can use two different acquisition rates on your analog signal and the acquisition automatically changes rates above and below the specified analog voltage value.

ETS (equivalent time sampling) is an acquisition technique that combines an analog input operation, hardware analog trigger, and pulse generation on a repetitive signal to achieve an effective acquisition rate that is significantly higher than the actual acquisition rate of the device.

The analog trigger is used to generate a repeating digital trigger signal from the repetitive analog signal. This digital signal is routed to the gate of a counter that generates another digital pulse with an increasing amount of delay in response to the trigger. This pulse is used as the scan clock of the acquisition. By triggering off the same point of the analog signal with an exact and varying delay, you can sample the entire signal waveform using many cycles. Because the change in the delay is much smaller than the minimum acquisition period of the device, the effective acquisition rate is significantly increased.

In the Pulse control, specify the input of the gate to be `AITrigger` and set the change in delay using the `ETSIncrement` property. The ETS increment is always specified in number of cycles of the clock source and can range between 0 and 255.

Tutorial: Using the Counter and Pulse Controls

This tutorial shows you how to develop a program that uses the Counter and Pulse controls. The example generates a continuous pulse of varying frequency using the Pulse control and counts the number of pulses generated using the Counter control. To complete this tutorial, your DAQ device must have two available counters. If you use just one counter with either the Counter or Pulse control, you need an external mechanism to display the output from the Pulse control or apply an input to the Counter control.

This tutorial uses Visual Basic syntax, but the discussion is in general terms so you can follow it in any compatible programming environment. Remember to adjust any code to your specific programming language. Refer to the *Building ComponentWorks Applications* chapters for information about implementing any step in different programming environments. You also can refer to the tutorial examples installed with ComponentWorks for a completed version of this example in several different programming environments.

Designing the Form

1. Open a new project and form. If you are working in Visual C++, select a dialog-based application and name your project `Counters`.
2. Load the ComponentWorks User Interface controls (specifically the Slide controls) and the Data Acquisition controls (specifically the Counter and Pulse controls) into your programming environment.

3. Place five buttons on the form. Change their captions and names to the ones listed in the following table.

Button	Name	Caption
CWButton1	ConfigurePulse	Configure Pulse
CWButton2	StartPulse	Start Pulse
CWButton3	StopPulse	Stop Pulse
CWButton4	StartCounter	Start Counter
CWButton5	ReadCounter	Read Counter

4. Place a ComponentWorks Slide control on the form and change its name to `Frequency`. In the **Numeric** tab of the custom property page, change the **Min** and **Max** properties to `1` and `100` and set the Log scale option.

5. Place a text box on the form.

6. Place a DAQ Pulse control on the form. In the custom property page of the Pulse control, select the device and counter you want to use. If you have an E-Series device, use counter 0. If you have an older MIO device, use counter 5. Otherwise, use an available counter. You set the remaining properties in the following section.

7. Place a DAQ Counter control on the form. In the property page, select the device and counter you want to use. If you have an E-Series or older MIO device, use counter 1. Otherwise, use an available counter.

8. If your environment has a timer control, place one on the form. In the property page, set its **Enabled** property to `False` and its **Interval** property to `100` (ms).

Your form should look similar to the one shown below.

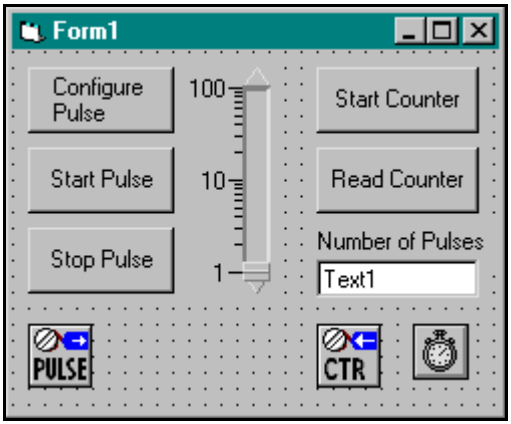

Figure 7-15. Counters Form

Developing the Code

Configure the DAQ controls and add the necessary code to generate the continuous pulse train and count the pulses. If you are using one counter, follow the directions for either the Counter or Pulse control. The next section describes how you can connect your signals in either case.

1. Configure the Pulse control to generate a continuous pulse train. On the **General** tab of the property page, set **Pulse Mode** to `Continuous pulse train`. On the **Clock** tab, set **Clock Mode** to `Use frequency settings`, **Frequency** to `1.0`, and **duty cycle** to `0.50` (50%). On the **Gate** tab, set **Gate Mode** to `Ungated`.
2. Three buttons on the form control the operation of the Pulse control. Generate event handlers for the `Click` event of each button and add the following code. You also need to declare a Boolean variable, `Running,` which is global to the module containing the code for the form. This declaration might vary among different programming languages. In Visual C++, you must create a member variable for the Pulse control.

   ```
   Dim Running as Boolean
   ```

   ```
   Private Sub ConfigurePulse_Click()
      CWPulse1.Configure
   End Sub
   ```

   ```
   Private Sub StartPulse_Click()
      CWPulse1.Start
      Running = True
   End Sub
   ```

   ```
   Private Sub StopPulse_Click()
      CWPulse1.Stop
      Running = False
   End Sub
   ```

   Use the **Stop** and **Start** buttons to halt the pulse generation and restart it without having to reconfigure the Pulse control.
3. The Slide control changes the frequency of the pulse train. If the pulse generation is running, the output updates immediately. Create an event handler for the `PointerValueChanged` event of the slide, and add the following code.

   ```
   Private Sub Frequency_PointerValueChanged(ByVal
   Pointer As Long, Value As Variant)
      CWPulse1.Frequency = Value
      If Running Then
         CWPulse1.Reconfigure
      Else
         CWPulse1.Configure
      End If
   End Sub
   ```

If the pulse generation is running, the `Reconfigure` method updates the output immediately. Otherwise, the `Configure` method reprograms the hardware for the next start.

4. Configure the Counter control to count events, which are the pulses generated by the pulse control. On the **General** tab of the property page, set **Measurement Type** to `Event count`. On the **Clock** tab, set **Timebase source** to `Counter's source`. On the **Gate** tab, set the **Gate Mode** to `Ungated`.

5. Two buttons on the form control the operation of the Pulse control. Generate event handlers for the `Click` event of each button and add the following code. In Visual C++, you must create a member variable for the Counter control. Add the following code to the two event handlers.

```
Private Sub StartCounter_Click()
   CWCounter1.Configure
   CWCounter1.Start
End Sub
```

```
Private Sub ReadCounter_Click()
   Dim CountValue As Long
   Dim Overflow As Boolean
   CWCounter1.ReadCounter CountValue, Overflow
   Text1.Text = CountValue
End Sub
```

After the counter starts, you can use the **Read** button to interactively read the value of the counter and display it in the textbox.

6. If you have a timer control on your form, you can program it to automatically read the value of the counter once it has been started. If you are working in Visual C++, create a member variable for the timer control.

To activate the timer, add the following line to the end of the code in the `StartCounter_Click` subroutine.

```
Timer1.Enabled = True
```

Create an event handler for the `Timer` event of the timer control and add a call to the `ReadCounter_Click()` subroutine.

```
Private Sub Timer1_Timer()
   ReadCounter_Click
End Sub
```

7. Save your project and form as `Counters`.

Testing Your Program

1. Before you can run your program, you must connect the output of the counter running the pulse generation to the source input of the counter counting events, which is an external connection. If you are using only one counter, you need other external hardware. If you are using the DAQ Pulse control, you should have a device to measure the output of the counter, such as an oscilloscope or external counter. If you are using the DAQ Counter control, you must connect an external pulse source to the source input of the counter.
2. Run the program. Start the pulse generation by clicking on the **Configure Pulse** and **Start Pulse** buttons. Start the counter. If you have the timer control enabled, the textbox displays the value of the event count. Without the timer control, you can use the **Read Counter** button to update the textbox.
3. Change the value of the slide to modify the frequency of the pulse generation. Notice that the count value in the textbox changes faster or slower accordingly. You also can stop and restart the pulse generation, which affects the count value.

Your completed running program should look similar to the one shown below.

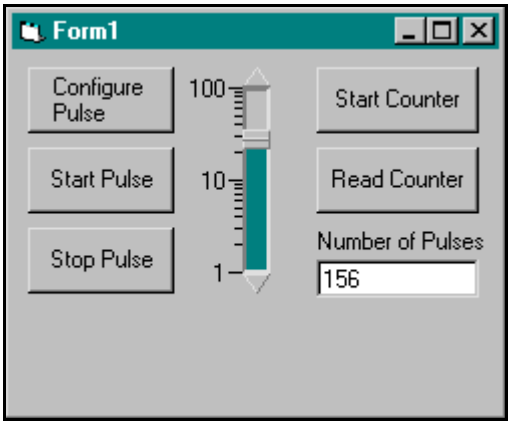

Figure 7-16. Testing Counters

4. Quit the program (click on the **X** button in the window title bar).

DAQTools—Data Acquisition Utility Functions

The ComponentWorks DAQ controls include a DAQTools control that contains a set of utility functions for data acquisition. Although the control itself has no properties or events, these utility functions are all methods of the DAQTools control. Use these functions with different data acquisition devices to implement functionality that is not part of any other DAQ control. The function groups in the DAQ Tools control include the following:

- `GetErrorText` function—Converts a ComponentWorks error number to a descriptive string. Use this in error handling to convert the return code from a DAQ method call to an error message.
- `Configure` functions—Configure specific devices or parts of devices such as the hardware analog trigger circuit on specific E-Series devices.
- `Conversion` functions—Convert measurement units to physical units for certain transducers, such as thermocouples, thermistors, RTDs, and strain gauges.
- `Get` and `Set` functions—Read and set different properties of data acquisition and SCXI (signal conditioning) devices.
- `Reset` functions—Reset DAQ and SCXI devices.
- `ICtr` functions—Perform operations using the interval counter (ICounter) on some data acquisition devices. Devices that include this counter are the 500, 700, 1200, LPM-16, and Lab-PC series devices. The functions are `ResetICtr`, `StartICtr`, and `ReadICtr`.
- `FOUT` functions—Generate a simple continuous pulse train from the FOUT pin of different DAQ devices. The functions are `StartFOUT` and `ResetFOUT`.
- `Calibration` functions—Perform software calibration on different data acquisition devices.

Note ***Because all devices are calibrated before shipping and do not need to be calibrated before their first use, you probably do not need to calibrate your device. Read all hardware documentation describing calibration and calibration functions before attempting to perform any calibration.***

Using DAQ Tools Functions

You must place the DAQTools control on your form to use the DAQTools functions. To call a function, use the standard convention for calling any method of a control; that is, prepend the name of the control to the function name.

The following example converts an error code number to a text description and displays it in a message box.

```
MsgBox "DAQ Error: " +
  CWDAQTools1.GetErrorText(ErrorCode)
```

To configure the hardware analog trigger circuit, you can use the `ConfigureATCOut` function.

```
'Prototype: ConfigureATCOut(Device:=, Enable:=,
  TriggerMode:=, Level:=, Hysteresis:=, strSource:=,
  ActualLevel:=, ActualHysteresis:=)

Dim Level as Variant, Hysteresis as Variant

CWDAQTools1.ConfigureATCOut 1, True, 1, 0#, 0.1, "0",
  Level, Hysteresis
```

8

Using the GPIB and Serial Controls

This chapter describes how you can use the ComponentWorks Instrument controls in your application to perform input and output operations using GPIB and serial hardware; explains the individual controls and their most commonly used properties, methods, and events; and includes tutorial exercises that give step-by-step instructions on using the controls in simple programs.

Refer to the *Building ComponentWorks Applications* chapters for information about using the ComponentWorks controls in different programming environments. You can find additional information in the online reference, available by selecting **Programs»National Instruments ComponentWorks»ComponentWorks Reference** from the Windows **Start** menu.

What Are the GPIB and Serial Controls?

Use the GPIB and Serial controls to perform instrument communication and control and integrate these operations into your application. ComponentWorks includes two ActiveX controls for performing GPIB and Serial (RS-232) operations, CWGPIB and CWSerial.

You can set most properties for these controls through property pages as you design your program. In certain cases, you might need to change the value of one or more properties in your program code. Throughout this chapter, examples demonstrate how to change property values programmatically.

Object Hierarchy and Common Features

The ComponentWorks Instrument controls are made up of a hierarchy of objects. Understanding the relationship among the objects in a control is the key to properly programming with the control. Dividing a control into individual objects makes it easier to work with because each individual component has fewer parts.

The Instrument controls share a very similar object hierarchy (compare Figure 8-1, *GPIB Control Object Hierarchy*, and Figure 8-3, *Serial Control Object Hierarchy*). The hierarchy consists of a simple series of objects from the top-level control through three sets of collections and associated objects. A collection is the property of an object that stores multiple instances of the same type of object. For example, a Tasks collection object contains multiple Task objects.

Common Properties

The top-level object of each Instrument control share some common properties.

`DataAsString`—Use the `DataAsString` property to specify whether data is returned in a string or byte array format. If this property is `True`, data is returned as a string. If this property is `False`, data is returned as an array of bytes. You can set this property from the property pages of each control or programmatically as in the following example.

```
CWGPIB1.DataAsString = True
```

`SwapBytes`—Some instruments return data in which the least significant byte appears first. This format is known as Little Endian. Other instruments return data starting with the most significant byte of the word, or in Big Endian format. Check your instrument documentation to determine if the instrument returns data in Big or Little Endian format. If your instrument returns data in Big Endian format, set `SwapBytes` to `True`.

`ExceptionOnError`—The Instrument controls handle error checking in two different ways. When an error occurs, each Instrument control generates an exception that your programming environment handles (default action). You can disable the generation of exceptions using the `ExceptionOnError` property of each Instrument control. If you disable exceptions on errors, the control fires an `OnError` event in response to an error condition and passes the error information to the `OnError` event handler. For some error conditions, such as those that occur during asynchronous operations, the control fires `OnError` events regardless of the value of `ExceptionOnError`.

Parsing

With the Instrument control parsing features, you can parse data into formats more easily used by your application. Both Instrument controls support multiple parsing tasks. Use tasks to specify the way in which the data is parsed, and you can invoke different tasks at different times in your application.

Each task contains a group of patterns. These patterns are made up of one or more tokens, which are the basic building blocks of parsing. For example, a token might be a number, a 4-byte word, or a comma. A pattern might consist of a number token followed by a comma token. A parsing task might consist of this particular pattern repeated several times.

Parsed data is returned as an array of tokens. The control returns parsed data as an array of tokens. You can configure the control to ignore specific types of tokens and not return them in the array. For example, you might not want the commas returned in certain patterns. If you specify commas as ignored, only the numbers are returned in the array. Use the built-in task `Number Parser` to automatically extract all numbers from the data and return them in an array. If only one number is present, it returns the number as a scalar (simple non-array variable).

Advanced Parsing Features

This section describes advanced parsing features. Refer to *Tutorial: Using the Serial Control* later in this chapter to learn how to use simple parsing features.

You perform a parsing task by calling the task's `Read` method. The `ReadAsynch` method performs an asynchronous read and returns the parsed data in the `DataReady` event. The task number is also passed to the event handler. The `Run` method of the Task object performs an automated asynchronous task, writing the output string specified for the task (either programmatically or on the **Parsing** property page) and then performs an asynchronous read. The `Parse` method parses an argument passed to the method rather than performing a read and then parsing the read data.

```
CWGPIB1.Tasks.Item(1).Read
CWGPIB1.Tasks.Item(1).Run
```

You can build a parsing task through property pages at design time. By creating and manipulating CWTask objects programmatically, you also can build parsing tasks at run time.

CWTask Object

To add a parsing task to a control, call the `Add` method of the Tasks collection property of the control, as in the following example.

```
CWGPIB1.Tasks.Add
```

The new CWTask object is added to the end of the Tasks collection. You can reference it by using the `Count` property of the Tasks collection. In Visual Basic, use the `Set` statement to assign an object to a variable.

```
Dim NewTask as CWTask ‘create a CWTask variable

CWGPIB1.Tasks.Add
Set NewTask = CWGPIB1.Tasks(CWGPIB1.Tasks.Count)
```

In environments other than Visual Basic, use the `Item` method on the Tasks collection to reference a specific Task object in the collection, as in the following block of code.

```
Dim NewTask as CWTask ‘create a CWTask variable

CWGPIB1.Tasks.Add
Set NewTask = CWGPIB1.Tasks.Item(CWGPIB1.Tasks.Count)
```

The CWTask object has two properties previously mentioned. `Name` is an identifier used to select a specific task in the Tasks collection. To programmatically set the `Name` property of the task added above use the following syntax.

```
NewTask.Name = “Waveform”
```

`OutputString` contains a string that is written to the instrument when the `Run` method is invoked.

```
NewTask.OutputString = “val1?”
```

Methods and Events

The CWTask object has four methods, `Read`, `ReadAsynch`, `Run`, and `Parse`, which are described earlier in this parsing section. The Task object also contains a Patterns collection which contains the individual Pattern object that defines the parsing task.

CWPattern Object

You add and manipulate pattern objects in the same manner that you add Task objects. The CWPattern object has several properties that define its parsing description. To access a Pattern object, reference it through the Task object in which it is contained.

```
NewTask.Patterns.Add
```

`Name` is an identifier used to select the pattern in the Patterns collection.

```
NewTask.Patterns(NewTask.Patterns.Count).Name = "Header"
```

`RepetitionFactor` is the number of times the pattern is to be repeated. Set this property to `-1` if the pattern is to be repeated an undefined number of times.

If you are programming in environments other than Visual Basic, use the `Item` method.

```
NewTask.Patterns.Item(NewTask.Patterns.Count).Name = "Header"
```

You also can use the complete syntax to access the Pattern object from the Instrumentation control.

```
CWGPIB1.Tasks.Item(CWGPIB1.Tasks.Count).Patterns(CWGPIB1.
Tasks.Item(CWGPIB1.Tasks.Count).Patterns.Count).Name = "Header"
```

The `Type` property of the Pattern object is set to one of two values—`cwNumberParser` or `cwUserDefined`—which are constants defined by ComponentWorks. Set the pattern type to `cwNumberParser` (only built-in pattern type) to pull all of the numbers from the input string and return them. Use the `cwUserDefined` type to define a pattern using tokens.

```
Dim NewPattern as CWPattern
```

```
Set NewPattern = NewTask.Patterns.Item(NewTask.Patterns.Count)
NewPattern.Type = cwUserDefined
```

The Pattern object contains a Tokens collection and associated Token objects that define the individual pattern.

CWToken Object

You add and manipulate Token objects in the same manner that you work with Task and Pattern objects. The CWToken object has several properties that define its parsing description.

`Value` is a string defining the type of token. For example, the value can be `"<number>"`, `"<stringX>"` (where `X` is the number of bytes that make up the token), or a literal string to match, such as `"FLUKE"` or `","`.

`RepetitionFactor` is the number of times the token is to be repeated. Set this property to `-1` if the token is to be repeated an undefined number of times. `Ignore` is a boolean property that determines whether the token should be ignored or returned with the data. For example, you might set the `Ignore` property to `True` so that commas are not returned to the application.

```
NewPattern.Tokens.RemoveAll
NewPattern.Tokens.Add
NewPattern.Tokens.Item(1).InputType = cwString
NewPattern.Tokens.Item(1).Value = ","
NewPattern.Tokens.Item(1).Ignore = True
```

The `RemoveAll` method used in the code segment above is available on all collections such as Tasks, Patterns, and Tokens. It removes all objects from the collection. A subsequent call to the `Add` method adds the first object in the collection; therefore, you know the exact index of any objects you add. Calling `RemoveAll` on the Tasks collection also removes the predefined number parsing task. You can recreate this task by adding a new task and new pattern and setting the `Type` property of the new pattern to `cwNumberParser`.

The GPIB Control

Use the GPIB control to control GPIB instruments. After setting the properties of the control, you can acquire data using method calls to the GPIB control and perform automated parsing of the returned data. To configure the properties of the control during design, right click on the control and select **Properties...** from the popup menu to open the property pages for the control.

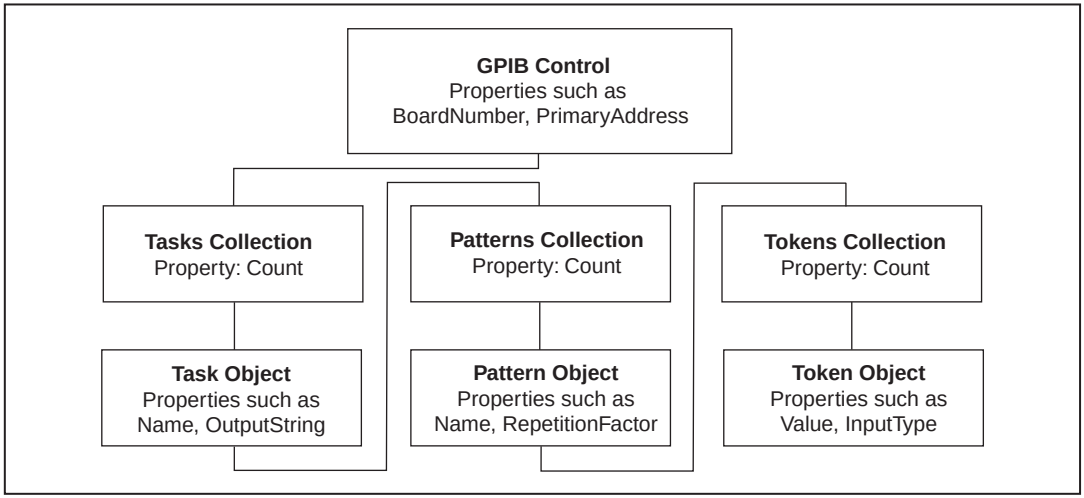

Figure 8-1. GPIB Control Object Hierarchy

The object hierarchy of the GPIB control contains a Tasks collection and objects for parsing. See *Advanced Parsing Features* earlier in this chapter for more information about accessing Pattern and Token objects.

CWGPIB Object

The GPIB object has GPIB-specific properties, such as `BoardNumber`, `PrimaryAddress`, and `SecondaryAddress`, that you use to specify the instrument. You can set these properties through the property pages or in your program, as in the following code.

```
CWGPIB1.BoardNumber = 0
CWGPIB1.PrimaryAddress = 2
CWGPIB1.SecondaryAddress = 0
```

Other properties include `Timeout`, `EOTMode`, `EOSCharacter`, `Compare8Bits`, `EOSEndsRead`, `EOIWithEOS`, `Unaddressing`, and `NotifyMask`, although not all instruments require you to set these properties. See the online reference for all properties and their uses.

Methods and Events

To control a GPIB instrument, use the `Configure` method. This method initializes the GPIB instrument and configures it with the settings you specified for the control. You must call the `Configure` method in the following cases:

- Before calling any other control methods
- After changing any properties (for new properties to take effect)
- After calling the `Reset` method, before performing any GPIB I/O

```
Private Sub ConfigureDevice_Click()
   CWGPIB1.Configure
End Sub

Private Sub ResetDevice_Click()
   CWGPIB1.Reset
End Sub
```

Synchronous I/O

You perform synchronous I/O using the `Read` and `Write` methods of the CWGPIB control. `Read` accepts an optional parameter that specifies the buffer size. Unlike the `Read` method on CWTask, the `Read` method on CWGPIB does not perform any parsing of the data.

The following example displays the identification string returned by many GPIB instruments. The control first writes the command `*IDN?` to the instrument and then reads back and displays the response in a text box in the program.

```
CWGPIB1.Write "*IDN?"
Text1.Text = CWGPIB1.Read
```

Asynchronous I/O

You use the `ReadAsynch` and `WriteAsynch` methods to perform asynchronous I/O operations. These methods start a read or write operation, respectively, and then return immediately. When an asynchronous read is complete, the control generates a `DataReady` event and passes the data to your event handler. When an asynchronous write is complete, the control generates a `WriteComplete` event.

```
CWGPIB1.ReadAsynch
```

```
Private Sub CWGPIB1_DataReady(taskNumber as Short,
 data As Variant)
   Text1.Text = data
End Sub
```

Unlike the `ReadAsynch` and `WriteAsynch` methods on CWTask, the `ReadAsynch` and `WriteAsynch` methods on CWGPIB do not perform any parsing of the data.

Other GPIB Operations

The GPIB control has methods to perform basic operations such as serial polling, parallel polling, triggering, and clearing the instrument. You can perform these methods with the following syntax.

```
CWGPIB1.SerialPoll
```

```
CWGPIB1.ParallelPoll
```

```
CWGPIB1.Trigger
```

```
CWGPIB1.Clear
```

Tutorial: Using the GPIB Control

This tutorial shows you an example of using the GPIB control in a simple program to control a Fluke 45 multimeter. While the Fluke 45 and this example use a specific command set, you can control most other GPIB instruments in a similar manner. Substitute commands for your particular instrument in place of the Fluke 45-specific commands.

This tutorial uses Visual Basic syntax, but the discussion is in general terms so you can follow it in any compatible programming environment. Remember to adjust any code to your specific programming language. Refer to the *Building ComponentWorks Applications* chapters for information about implementing any step in different programming environments. You also can refer to the tutorial examples installed with ComponentWorks for a completed version of this example in several different programming environments.

Designing the Form

1. Open a new project and form. If you are working in Visual C++, select a dialog-based application and name your project `GPIBExample`.
2. Load the ComponentWorks Instrument controls (specifically, the GPIB control) into your programming environment.

3. Place a GPIB control on the form. You configure its properties in the next section.

4. Place a text edit box control on the screen. Leave the `Name` property for the text edit box set to `Text1`.

5. Place a button control on the form. Change the `Name` and `Caption` properties of the button to `FREQ`.

Your form should look similar to the one show below.

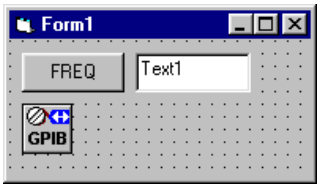

Figure 8-2. GPIBExample Form

Setting the GPIB Control Properties

You normally configure the default property values of the Instrument controls before you develop your program code. Most or all properties are set during design and do not change during program execution; however, you can edit the properties of the Instrument controls at run time, if necessary.

1. Open the custom property pages for the GPIB control on the form by right clicking on the control and selecting **Properties...**.
2. In the General tab, select your GPIB instrument from the **Device** combobox. If it is not present, make sure that the GPIB driver software is properly installed and configuration software recognizes the instrument. Alternatively, you can manually select the GPIB board number and instrument address.
3. Select other communication parameters in the **Advanced** tab.
4. Open the **Test** property page for the GPIB control. To ensure that the control is properly communicating with the instrument, enter `"*IDN?"` in the Send String box and press the **Execute** button. If your instrument

receives this command a string similar to "FLUKE, 45, 5005161, 1.6 D1.0" appears in the output window. You can send any string to the instrument from this property page. For example, for the Fluke 45 enter "val1?" in the Send String box and press the **Execute** button. A value appears in the output window. The **Execute** button performs both send and read operations.

Developing the Code

Develop the code so that data is acquired and displayed in response to a user pressing the button. In the following steps, you define an event handler routine to be called when the **FREQ** button is pressed. In the event handler, you send a command to the Fluke 45 to read the frequency and display it in the text box on the user interface.

1. Generate the event handler routine for the `Click` event of the **FREQ** button. The following code configures the GPIB control and sends a command to the Fluke 45 to switch it into frequency mode, reads the current value from the multimeter, and finally displays the value in the text edit box.

   Add the following code to the `FREQ_Click` subroutine. In Visual C++, remember to generate member variables for any controls referenced in the program. See the `\Tutorial` folder for Visual C++ and Delphi code examples.

   ```
   Private Sub FREQ_Click()
       CWGPIB1.Configure
       CWGPIB1.Write "FREQ"
       CWGPIB1.Write "val1?"
       Text1.Text = CWGPIB1.Read
   End Sub
   ```

2. Save the project as `GPIBExample`.

Testing Your Program

Run and test the program. Click on the **FREQ** button. The value displayed in the text edit box should match the value displayed on the Fluke. By selecting a different mode of the instrument, you can perform other types of measurements using the Fluke 45. Change the "FREQ" command sent to the instrument with a corresponding command.

The Serial Control

Use the Serial control to control serial instruments, such as scales and calipers, or to communicate with other devices connected to your serial port, including modems and fieldbus networks, using a serial connection and other computers. After setting the properties of the control, you can acquire data or send commands using method calls to the serial control and perform automated parsing of the returned data. To configure the properties of the control during design, right click on the control and select **Properties...** from the popup menu to open the property pages for the control.

The object hierarchy of the Serial control contains a Tasks collection and objects for parsing. Refer to *Advanced Parsing Features* earlier in this chapter for more information about accessing Pattern and Token objects.

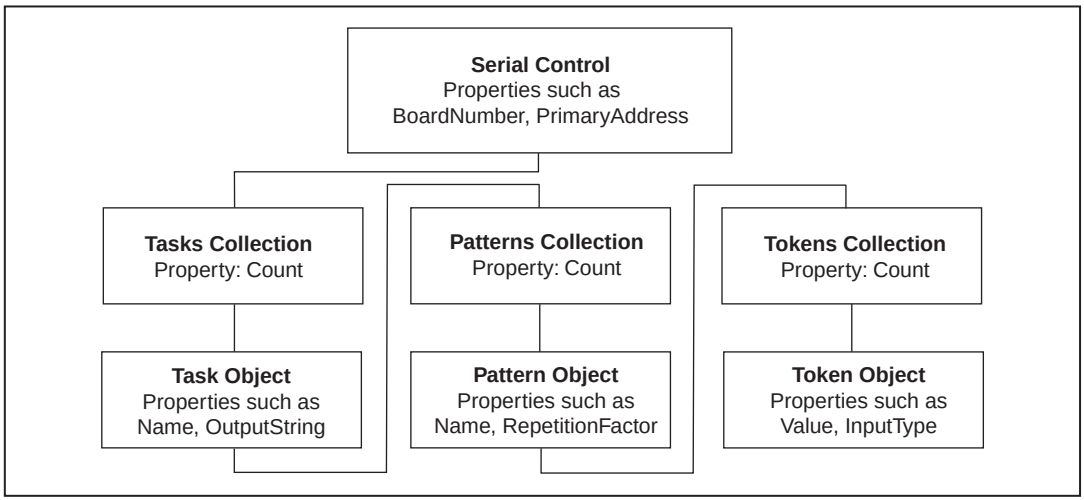

Figure 8-3. Serial Control Object Hierarchy

CWSerial Object

The Serial object has serial-specific properties—`ComPort`, `Parity`, `StopBits`, `DataBits`, `BaudRate`, and `EOSChar`—that you use to specify and configure the communications port.

```
CWSerial1.ComPort = 2 'Instrument connected to COM port 2
```

The following settings must match the settings used by your instrument.

```
CWSerial1.Parity = cwParityEven
CWSerial1.BaudRate = 9600
CWSerial1.StopBits = cwStopBitsTwo
CWSerial1.DataBits = cwDataBitsEight
CWSerial1.EOSChar = "\10"
```

Other communication properties you can set include `FlowControl`, `XonChar`, and `DTR`.

Methods and Events

To begin communication with a serial instrument, use the `Configure` method. This method initializes a communication process with the serial instrument and configures it with the settings you specified for the control. You must call the `Configure` method in the following cases:

- Before calling any other control methods
- After changing any properties (for new properties to take effect)
- After calling the `Reset` method, before performing any serial I/O

```
Private Sub ConfigureDevice_Click()
   CWSerial1.Configure
End Sub
```

```
Private Sub ResetDevice_Click()
   CWSerial1.Reset
End Sub
```

Synchronous I/O

You perform synchronous I/O using the `Read` and `Write` methods of the CWSerial control. Read accepts an optional parameter that specifies the buffer size. Unlike the `Read` method on CWTask, the `Read` method on CWSerial does not perform any parsing of the data.

The following example displays a string returned by a serial scale.

```
CWSerial1.Write “P\13”
Text1.Text = CWSerial1.Read
```

Many serial instruments require some termination character or characters, such as the carriage return or line feeds. In Visual Basic you can enter these unprintable characters using the backslash symbol (\) followed by the ASCII value of the character you want to send.

Asynchronous I/O

You use the `ReadAsynch` and `WriteAsynch` methods to perform asynchronous I/O operations. These methods start a read or write operation, respectively, and then return immediately. When an asynchronous read is complete, the control generates a `DataReady` event and passes the data to your event handler. When an asynchronous write is complete, the control generates a `WriteComplete` event.

```
CWSerial1.ReadAsynch
```

```
Private Sub CWSerial1_DataReady(taskNumber as Short,
  data As Variant)
   Text1.Text = data
End Sub
```

Unlike the `ReadAsynch` and `WriteAsynch` methods on CWTask, the `ReadAsynch` and `WriteAsynch` methods on CWSerial do not perform any parsing of the data.

Tutorial: Using the Serial Control

This tutorial shows you an example of using the Serial control in a simple program to control a serial scale. Other serial instruments behave differently and use different commands, but this tutorial provides a good illustration of how to use the Serial control and its parsing ability.

This tutorial uses Visual Basic syntax, but the discussion is in general terms so you can follow it in any compatible programming environment. Remember to adjust any code to your specific programming language. Refer to the *Building ComponentWorks Applications* chapters for information about implementing any step in different programming environments. You also can refer to the tutorial examples installed with ComponentWorks for a completed version of this example in several different programming environments.

Designing the Form

1. Open a new project and form. If you are working in Visual C++, select a dialog-based application and name your project `SerialExample`.
2. Load the ComponentWorks instrument controls (specifically, the Serial control) into your programming environment.

3. Place a Serial control on the form. You configure its properties in the next section.

4. Place a text edit box control on the screen. Leave the `Name` property for the text edit box set to `Text1`.

5. Place a button control on the form. Change the `Name` and `Caption` properties of the button to `Weigh`.

Your form should look similar to the one show below.

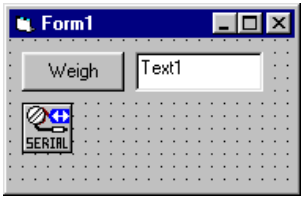

Figure 8-4. Weigh Form

Setting the Serial Control Properties

You normally configure the default property values of the Instrument controls before you develop your program code. Most or all properties are set during design and do not change during program execution; however, you can edit the properties of the Instrument controls at run time.

1. Open the custom property pages for the Serial control on the form by right clicking on the control and selecting **Properties...**.
2. In the General tab, select the communications port to which your serial instrument is connected.
3. On the General and the Flow Control property pages, adjust the properties of the control to match the setting of your instrument. These settings vary by instrument. Refer to your instrument documentation for details on specific instrument settings.
4. Open the **Test** property page for the Serial control. To ensure that the control is properly communicating with the instrument, enter `"P\13"` or another string compatible with your instrument in the Send String box and press the **Execute** button. A value, similar to `"1.0040 Lbs 02"` appears in the output window. The **Execute** button performs both send and read operations. Each instrument has a different command interface, which should be documented in your instrument manual.

Add a custom parsing task to the control in order to extract the number from the string.

5. Open the **Parsing** property page and press the **Task** button.
6. Click on the string `New Task1` and change its name to `Weight`.

7. Add a pattern to the new parsing task by clicking the **Pattern** button.

Add the tokens that make up the parsing tasks.

8. Add a token to the pattern by clicking the **Token** button.
9. Change its value to a number by selecting <number> in the **Token** combobox at the top right.

Add a token to grab the rest of the string.

10. Click the **Token** button.
11. Change the value of the token to <string1>.
12. Select **Many** from the **Token Repeat** options and select the **Ignore Token** box. This token matches the rest of the string but is not returned as data.

Your **Parsing** page should look like the following figure.

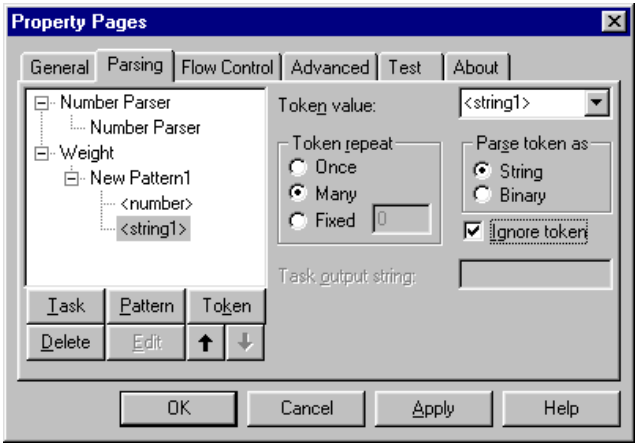

Figure 8-5. Serial Property Pages—Parsing Page

13. Click on **Apply** and **OK**.

Developing the Code

Develop the code so that data is acquired and displayed in response to the button. In the following steps, you define an event handler routine to be called when the **Weigh** button is pressed. In the event handler, you send a command to the scale to read the weight and display the parsed and returned data in the text edit box.

1. Generate the event handler routine for the `Click` event of the **Weigh** button. The following code calls the `Configure` method of the Serial control, sends the command `"P\13"` to prompt the scale to send back its current reading, and reads back the value from the scale using the `Read` method of the defined task. The returned information is automatically parsed by the corresponding Task object. Finally, it displays the returned value in the text edit box.

   Add the following code to the `Weigh_Click` subroutine. In Visual C++, remember to generate member variables for any controls referenced in the program. See the `\tutorials` folder for Visual C++ and Delphi code examples.

   ```
   Private Sub Weigh_Click()
      CWSerial1.Configure
      CWSerial1.Write "P\13"
      Text1.Text = CWSerial1.Tasks.Item("Weight").Read
   End Sub
   ```

Note ***Because the weight task is the second task in the collection (after the built-in number parser task at index 1), you can specify the value*** `2` ***as an index to the Tasks collection instead of*** `Weight`**.**

2. Save the project as `SerialExample`.

Testing Your Program

Run and test the program. Click on the **Weigh** button. The value displayed in the text edit box should match the value displayed on the scale. You can further enhance this example by making the read asynchronous. Change the `Read` method call to a `ReadAsynch` call, and display the value returned in the `CWSerial1_DataReady` event handler.

9

Using the VISA Control

This chapter describes the basic structure of the VISA API; shows you how to use the ComponentWorks VISA control in your application to perform input and output operations using GPIB, Serial, and VXI hardware; explains the control and its most commonly used properties, methods, and events; and includes tutorial exercises that give step-by-step instructions on using the control in simple programs.

Refer to the *Building ComponentWorks Applications* chapters for information about using the ComponentWorks controls in different programming environments. You can find additional information in the online reference, available by selecting **Programs»National Instruments ComponentWorks»ComponentWorks Online Reference** from the Windows **Start** menu.

Overview of the VISA API

VISA is a standard I/O API for instrumentation programming. By itself, VISA does not provide instrumentation programming capability.

VISA Structure

VISA is a high-level driver that calls lower-level drivers. VISA can control VXI, GPIB, and serial instruments, as the hierarchy in Figure 9-1indicates.

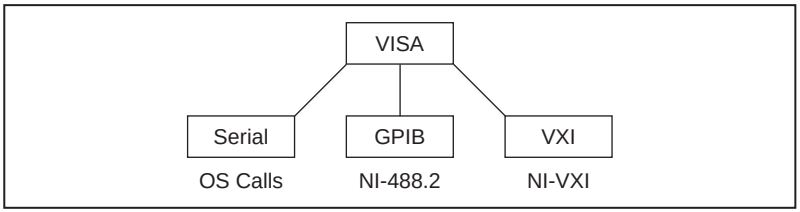

Figure 9-1. VISA Structure

When debugging VISA problems, remember that this hierarchy exists. Although VISA appears to be causing your problem, you might have a bug or installation problem with one of the drivers into which VISA is calling.

VISA Advantages

VISA provides interface independence. Regardless of the interface type, VISA uses many of the same methods to communicate with instruments. For example, the VISA command to write an ASCII string to a message-based instrument is the same for Serial, GPIB, and VXI. This feature allows you to switch interfaces and use a single language to work with different interfaces for instruments.

VISA is an object-oriented language that can easily adapt to new instrumentation interfaces as they are developed in the future, which will enable programmers to easily move to new interfaces.

As an object-oriented language, VISA and its operations are intuitive. VISA provides the most frequently used functionality for instrumentation programming in a very compact command set.

What is the VISA Control?

Use the VISA control to program your instruments and integrate these operations with the rest of the application. The VISA control, CWVISA, is an implementation of the VISA API. After setting the properties of the control, you can acquire data using simple method calls to the VISA control and perform automated parsing of the returned data. To configure the properties of a control during design, right click on the control and select **Properties...** from the popup menu to open the property pages for the control. From the property pages, you can intuitively set property values.

You can set most properties through the property pages as you design and create the program. In certain cases, you might need to change the value of one or more properties in your program code. Throughout this chapter, examples demonstrate how to change property values programmatically.

Object Hierarchy and Common Properties

The ComponentWorks VISA control is made up of a hierarchy of simple objects. Understanding the relationship among the objects in a control is the key to properly programming with the control. Dividing a control into individual objects makes it easier to work with because each individual component has fewer parts.

Figure 9-2 shows the object hierarchy of the CWVISA control.

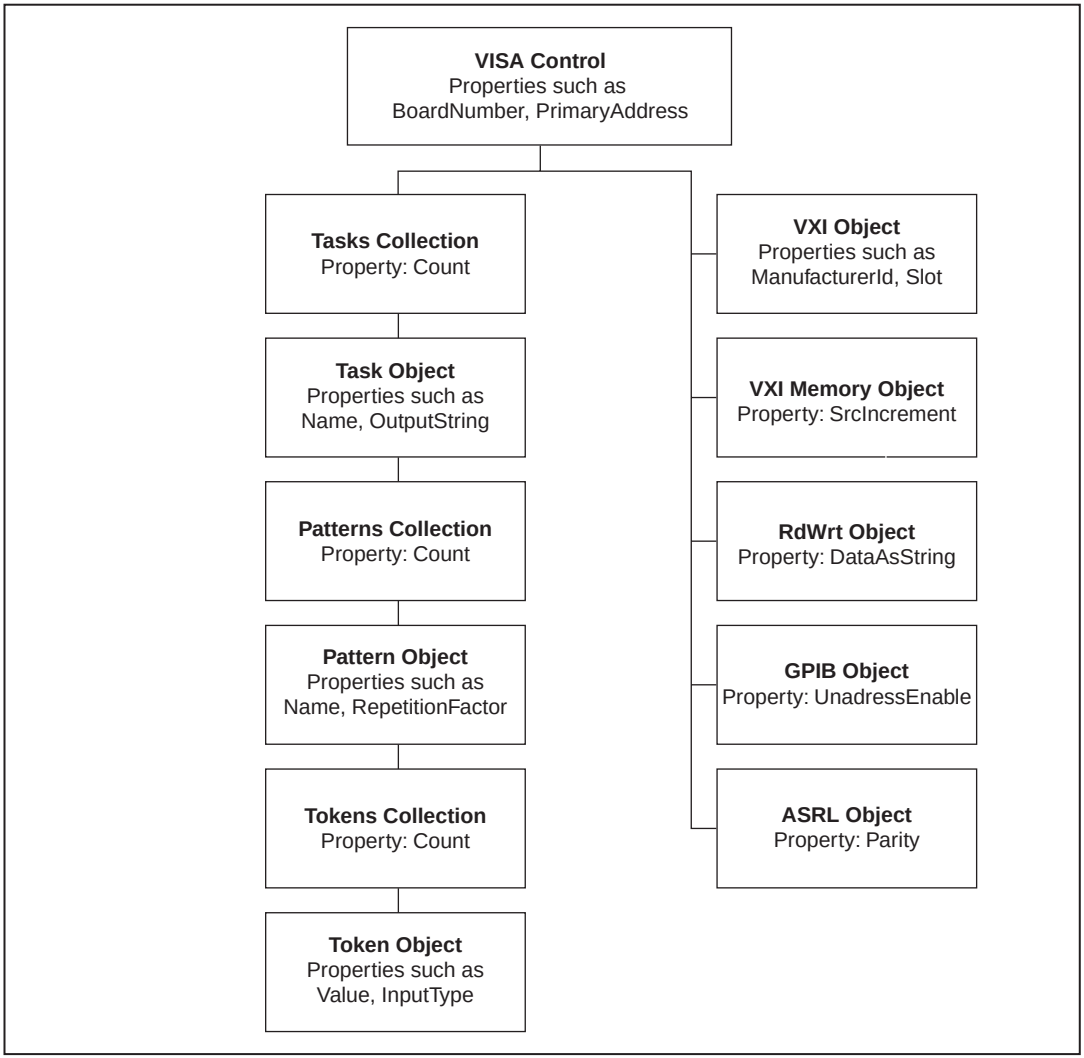

Figure 9-2. VISA Control Object Hierarchy

The VISA control contains a Tasks collection and underlying objects for parsing, an ASRL (Serial) object for serial properties, a RdWrt object for message-based I/O properties, a VXI Memory object for VXI register-access properties, a GPIB object for GPIB properties, and a VXI object for VXI specific properties.

A *collection* is a property of an object, which stores multiple instances of the same type of object. For example, the VISA control contains a Tasks collection object that contains multiple Task objects.

Common Instrument Control Features

The VISA control uses the same parsing sub-objects as the GPIB and Serial controls. The Instrument controls share certain common properties.

`SwapBytes`—Some instruments return data in which the least significant byte appears first. This format is known as Little Endian format. Other instruments return data in which the address points to the most significant byte of the word, or in Big Endian format. Check your instrument documentation to determine if the instrument returns data in Big or Little Endian format. If your instrument returns data in Big Endian format, set the `SwapBytes` property of the RdWrt object to `True`, as in the following example:

```
CWVisa1.RdWrt.SwapBytes = True
```

`ExceptionOnError`—The Instrument controls handle error checking in two different ways. When an error occurs, each Instrument control generates an exception that your programming environment handles (default action). You can disable the generation of exceptions using the `ExceptionOnError` property of each Instrument control. If you disable exceptions on errors, the control fires an `OnError` event in response to an error condition and passes the error information to the `OnError` event handler. For some error conditions, such as those that occur during asynchronous operations, the control fires `OnError` events regardless of the value of `ExceptionOnError`.

`DataAsString`—Use the `DataAsString` property to specify if data is returned as a string or an array of bytes. If you set this property to `True`, the data is returned as a string. If you set this property to `False`, the data is returned as an array of bytes. You can set this property from the RdWrt page of the VISA property pages or in your program:

```
CWVisa1.RdWrt.DataAsString=True
```

Parsing

As with the GPIB and Serial controls, you can use the VISA control to parse data into formats more easily used by your application. The VISA control supports multiple parsing tasks. Use these tasks to specify the way in which the data is parsed, and you can invoke different tasks at different times in your application.

Each task contains a group of patterns. These patterns are made up of one or more tokens, which are the basic building blocks for parsing. For example, a token might be a number, a 4-byte word, or a comma. A pattern might consist of a number token followed by a comma token. A parsing task might consist of this particular pattern repeated several times.

Parsed data is returned as an array of tokens. The control returns parsed data as an array of tokens. You can configure the control to ignore specific types of tokens and not return them in the array. For example, you might not want the commas returned in certain patterns. If you specify commas as ignored, only the numbers are returned in the array. Use the built-in pattern type `Number Parser` to automatically extract all numbers from the data and return them in an array. If only one number is present, it returns the number as a scalar (simple non-array variable).

For more detailed information about parsing objects, see *Advanced Parsing Features* in Chapter 8, *Using the GPIB and Serial Controls*. To use the parsing features in a tutorial, see *Tutorial: Using the Serial Control* in Chapter 8, *Using the GPIB and Serial Controls*.

VISA Object

The VISA object has several properties for performing I/O with any interface (GPIB, Serial, or VXI). The most important is the resource name (`RsrcName`) property, which you can use to select the instrument being addressed by the VISA control. `RsrcName` specifies the exact name and location of a VISA resource, with the following format.

```
Interface Type[Board Index]::Address::VISA Class
```

- `Interface Type`—Specifies the I/O interface to which the device is connected (GPIB, ASRL (Serial), or VXI).
- `Board Index`—Specifies the index of the board. Use `Board Index` only if the system has more than one interface type. For example, if the system contains two GPIB plug-in boards, you can refer to one as `GPIB0` and the other as `GPIB1`.

- `Address`—For VXI instruments, the `Address` parameter is the logical address of the instrument. For GPIB instruments, `Address` is the GPIB primary address.

  For GPIB communication with secondary addressing, add the secondary address to the resource name. For example, to communicate with a GPIB instrument with primary address 4 and secondary address 2, use the resource name `GPIB::4::2::INSTR`.

  Do not use `Address` for serial instruments. For example, `ASRL1::INSTR` is the descriptor for the COM 1 serial port on a personal computer.

- `VISA Class`—Specifies the grouping that encapsulates some or all of the VISA methods. INSTR is the general class that encompasses all VISA methods. In the future, other classes might be added to the VISA specification. Currently, you are not required to include `VISA Class`, but you should include it to ensure future compatibility. Most applications use the INSTR class.

The VISA control can search the system for available resources. From the **General** property page, you can view all available resources in the **Resource Name** list box, as shown in Figure 9-3.

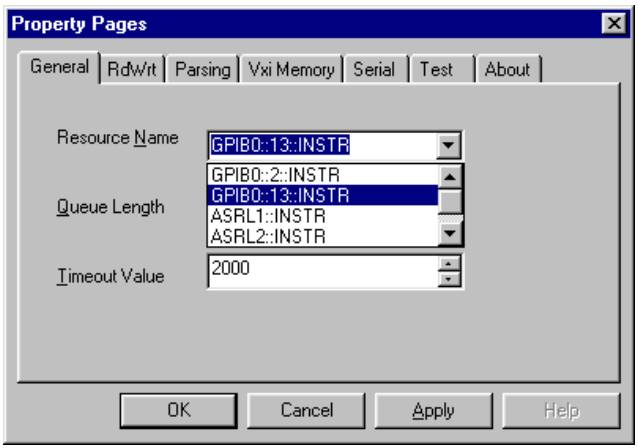

Figure 9-3. VISA Property Pages—General Page

Other general attributes include **Queue Length** and the **Timeout Value**.

- **Queue Length**—The length of the queue, used only with a synchronous method of event handling.
- **Timeout Value**—The value in milliseconds used for Input/Output (IO) operations.

You also can set these properties programmatically.

```
CWVisa1.RsrcName = "GPIB::2::INSTR"
CWVisa1.MaxQueueLength = 100
CWVisa1.Timeout = 1000
```

Other properties include `InterfaceInstanceName`, `InterfaceType`, `InterfaceNumber`, `ResourceImplVersion`, `ResourceSpecVersion`, and `ResourceManufacturerName`.

RdWrt Object

The RdWrt object contains properties that apply to general message-based communication for any type of interface—GPIB, Serial, or VXI. Figure 9-4 shows the RdWrt object properties.

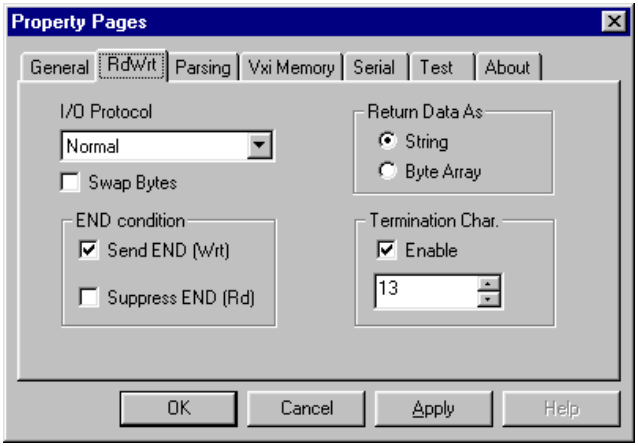

Figure 9-4. VISA Property Pages—RdWrt Page

- **I/O Protocol**—Usually left at `Normal`, but other options are available depending on the interface.
- **END condition**—Specifies whether the end condition is sent at the end of write operations.
- **Return Data As**—Specifies whether to return data as a string or byte array. You also can set this property programmatically.

  Use the `DataAsString` property to specify the format of returned data. If you set this property to `True`, the data returns as a string. If you set this property to `False`, the data returns as an array of bytes. You can set this property from the **RdWrt** tab of VISA property page or programmatically.

  ```
  CWVisa1.RdWrt.DataAsString = True
  ```

- **Termination Character**—Use the **Enable** feature to set a termination character for terminating read operations.

Serial (ASRL) Object

The Serial object contains properties pertaining to VISA communication with a serial device. Use the settings to configure the serial port for serial communication, to check the state of serial hardware lines, and to check the number of bytes available in the serial input buffer. These properties include `BaudRate`, `DataBits`, `StopBits`, and `FlowControl`. You can set most of these properties on the **Serial** page of the VISA control property pages, as shown in the following figure.

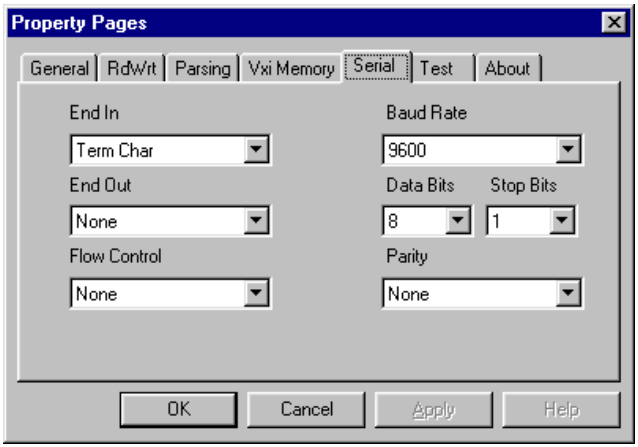

Figure 9-5. VISA Property Pages—Serial Page

GPIB Object

The GPIB object contains properties pertaining to VISA communication with a GPIB resource. Two of these properties, `PrimaryAddess` and `SecondaryAddress`, are read only. Use the other two properties, `ReaddressEnable` and `UnaddressEnable`, to enable readressing or unaddressing after GPIB read and write operations. By default, `ReaddressEnable` is enabled and `UnaddressEnable` is disabled.

VXI Object

The VXI object contains properties pertaining to VISA communication with a VXI device. All properties, including `ManufacturerId`, `Slot`, and `ModelCode`, are read only. Use the properties to access information obtained by the VXI Resource Manager utility when it configures the VXI system.

Methods and Events

To begin controlling a resource with the VISA control, use the `Open` method. This method initializes the resource and configures it with the settings that you specified for the control. You must call `Open` before you can call any other methods of the control. If you call the `Close` method at any point in the program, you must call `Open` again before performing any I/O.

```
Private Sub OpenDevice_Click( )
   CWVisa1.Open
End Sub
```

```
Private Sub CloseDevice_Click( )
   CWVisa1.Close
End Sub
```

Message-Based Communication

All serial and GPIB devices, and many VXI devices, recognize a variety of message-based command strings. With the VISA control, the actual protocol used to send a command string to an instrument is transparent. You need to specify only whether you want to write a message or read a message from a message-based device.

Note ***The same methods are used to write message-based commands to GPIB, serial, and message-based VXI instrument. VISA automatically knows which functions to call based on the type of resource being used.***

Synchronous I/O

You perform synchronous message-based I/O using the `Read` and `Write` methods of the CWVisa control. `Read` accepts an optional parameter that specifies the buffer size. If you omit this parameter, the buffer size is specified by the `DefaultBufferSize` property. Unlike the `Read` method on CWTask, the `Read` method on CWVisa does not perform any parsing of the data.

The following example displays the identification string returned by many message-based instruments. The control first writes the command `*IDN?` to the instrument and then reads back and displays the response in a text box in the program.

```
CWVisa1.Write "*IDN?"
Text1.Text = CWVisa1.Read
```

Asynchronous I/O

You use the `ReadAsynch` and `WriteAsynch` methods to perform asynchronous message-based I/O operations. These methods start a read or write operation, respectively, and then return immediately. When an asynchronous read is complete, the control generates a `DataReady` event and passes the data to your event handler. When an asynchronous write is complete, the control generates a `WriteComplete` event.

```
Private Sub CWVisa2_DataReady(ByVal taskNumber As
 Integer, ByVal Data As Variant)
   Text1.Text = Data
End Sub
```

Both of these operations also generate the `IOCompletion` event when they finish. You can use these events when you want your program to continue executing other tasks during a long write or read operation.

Unlike the `ReadAsynch` and `WriteAsynch` methods on CWTask, the `ReadAsynch` and `WriteAsynch` methods on CWVisa do not perform any parsing of the data.

Register-Based Communication

Some VXI instruments do not support message-based commands. To communicate with these instruments, use register accesses. The VISA control contains a set of register access methods that you can use with VXI instruments.

Because all VXI instruments have configuration registers in the upper 16 kilobytes of A16 memory space, you can use register access functions to read from and write to the configuration registers for message-based devices. To read a value from a register, use the basic VISA `In` method. Three different versions of the `In` operation exist so you can read 8-, 16-, or 32-bit values.

You can set the properties related to VXI register accesses in the **VXI Memory** page of the VISA property pages, as shown in the following figure.

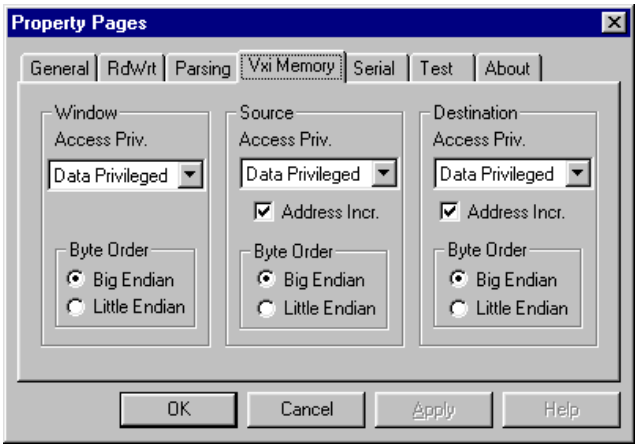

Figure 9-6. VISA Property Pages—VxiMemory Page

The `Access Privilege` property specifies the VME access privilege. The `Byte Order` property specifies the byte order used in performing register accesses. The `Source` and `Destination` attributes apply to the high-level register access operations. The `Address Increment` properties apply to the `MoveIn` and `MoveOut` methods only. The `Window` attributes apply to lower-level register access methods only.

High-Level Register Accesses with the In and Out Methods

The parameters for the `In16` method, `space` and `offset`, specify the address space and the offset. The address space is an enumeration indicating whether the A16, A24, or A32 space is being accessed. The following list includes the constants for the different address spaces.

Constant Name	Description
`cwVxiA16Space`	A16
`cwVxiA24Space`	A24
`cwVxiA32Space`	A32

The **offset** parameter is a long integer specifying the address in the indicated address space. Remember that VISA keeps track of the base memory address that a device requests in each address space. The **offset** input is relative to this base address. For example, suppose that you have a device at logical address 1 and want to use the `In16` method to read its ID/Logical Address configuration register. This register is at absolute address 0xC040 in A16 space and the configuration registers for the device at logical address 1, range from 0xC040 to 0xC07F. Because VISA also holds this information, you only need to specify the offset in the region you want to access. In this case, that offset is zero.

The `Out16` method uses the address space and offset parameters in addition to a third integer parameter, which specifies the value to be written to the register. Do not use the VISA `Out` methods on read-only registers.

Moving Blocks of Data with the MoveIn and MoveOut Methods

Use the `MoveIn` and `MoveOut` methods to move large blocks of 8-, 16-, or 32-bit data from a VXI address space to local memory or from local memory to a VXI address space. The `Move` methods are efficient operations, and you should use them for block transfers of data. If you do not select the `Address Increment` property, the block of data moves to a single location (FIFO) rather than to a corresponding block.

In addition to the `MoveIn` and `MoveOut` methods, you can use the `Move` and `MoveAsync` methods with a special resource of the MEMACC class (instead of INSTR class). This resource has access to all VXI address spaces via absolute addresses (instead of relative offsets for a particular resource). `Move` and `MoveAsync` can move data from one VXI memory range to another as well as to or from local memory.

Low-Level Register Accesses with the Peek and Poke Methods

If the high-level `In` and `Out` methods cannot sufficiently perform register accesses, you can use the lower-level methods. Low-level register access methods use the User Window specified in the controller's VXI Configuration utility to access the VXI/VMEbus. The `Peek`*XX* method reads registers, and the `Poke`*XX* method writes to registers, where *XX* specifies the corresponding size in bits of the access (8, 16, or 32). The `Peek` and `Poke` methods are generally faster than the corresponding `In` and `Out` methods.

Before you can use these methods, map a part of the VXI memory space to local memory with the `MapAddress` method. The `MapAddress` method takes the following input: address space, base VXI address, size of memory (in bytes), and a suggested local address to which the memory can be mapped. `MapAddress` returns the actual local address to which the specified base VXI address was mapped. If the `MapAddress` method completes without an error, check the `WinBaseAddr` and `WinSize` properties under the VxiMemory object to verify the actual size of the window that was mapped.

Note ***Before attempting to use the*** `Peek` ***and*** `Poke` ***methods, verify that the*** `MapAddress` ***method executed successfully. If the operation did not successfully execute, the system can crash.***

Use the following steps as a guideline for developing a program that performs low-level register accesses.

1. Set the property pages for the resource that you want to access.
2. Call the `Open` method.
3. Use the `MapAddress` method to map a region of a VXI address space to local memory.
4. Check the mapped window size and window base to make sure you do not access an area outside this window.
5. Perform `Peek` and `Poke` operations.
6. Call the `UnMapAddress` method to free the window.
7. Call the `Close` method.

If you want to access another VXI address space using low-level register access functions, unmap your current window and remap it to the region that you want to peek or poke.

Events

In VISA, events provide communication between VISA resources and a VISA application. Events are generated when certain conditions occur in the system. When the specified conditions occur, the corresponding event handler routine is called. To respond to an event, place the code that you want to execute into the event handler routine. Some events return additional information in the form of a VisaEvent object, but most events simply indicate that a specific condition occurred.

When should you use events? Consider the following situation. Suppose that you have a sophisticated message-based VXI device in your chassis that can perform a variety of different measurement operations, but some measurements require several minutes for the device to obtain a stable reading. After writing the command to the device to get the measurement, you cannot be sure when the measurement is complete. After performing the write operation, you can continuously attempt to perform VISA reads from the device until one is successful. Alternatively, you might attempt to read the result infrequently until it is available or wait a long time before checking to see if the result is available.

Several problems arise with these solutions. Continuous polling interferes with concurrent operations, such as controlling other instruments, writing data to disk, or updating the screen display. Also, these methods do not provide timely solutions. This scenario illustrates the need for other means of communication between VISA resources and a VISA application. Events provide this alternate means of communication.

Event Types

In addition to the events produced by asynchronous I/O operations, the VISA control generates the following events.

- `ServiceRequest`—Notifies the application when a device requests service from its controller. This event returns the status byte that was obtained by serial polling the device generating the request. This event is primarily produced by GPIB devices asserting the GPIB SRQ line and message-based VXI devices.
- `Trigger`—Notifies the application when a device asserts a trigger line on the VXI backplane. The returned VisaEvent object contains a `TriggerId` property specifying the trigger line with which the event is associated. When you set the `TriggerId` property of the VISA control, the `Trigger` event is sensitized to a single trigger line each time. You must set the line before you can start responding to trigger

events on that trigger line. For example, you can receive trigger events for triggers on TTL Trigger Line 5 with the following code.

```
CWVisa1.TriggerId = cwTrigTTL5
CWVisa1.EnableEvent CWVisaEventTrigger
```

To detect triggers on two different trigger lines at the same time, use a second VISA control.

- `VxiSignalProc`—Notifies the application when a device asserts an interrupt on the VXI backplane or produces a VXI signal. When a VXI interrupt or signal occurs, there is a `StatusId` value associated with the event. This value is returned by the `VxiSignalProc` event. Unlike triggers, a VXI interrupt on any of the interrupt levels produces a `VxiSignalProc` event. The lower 8 bits of this `StatusId` are set to the logical address of the device creating the signal or interrupt. VISA checks this `StatusId` value and generates an event only if the device creating the event is the device specified by the resource name property.

  The following code fragment checks the `StatusId` value associated with a `VxiSignalProc` event.

  ```
  Private Sub CWVisa1_VxiSignalProc(ByVal VisaEvent As
    CWVisaLib.CWVisaEvent)
      Text1.Text = VisaEvent.StatusId
  End Sub
  ```

  The `VxiSignalProc` event does not return the actual interrupt line that was asserted on the backplane. If you need the actual interrupt line, use the `VxiVmeInterrupt` event.

- `VxiVmeInterrupt`—Notifies the application when a device asserts an interrupt on the VXI backplane. This event is similar to the `VxiSignalProc` event except it is produced only for VXI or VME interrupts, not signals. It also returns the interrupt line on which the interrupt occurred and the `StatusId`.

Event Handling With The Event Queue

The VISA control provides an alternative mechanism for responding to events. Instead of implementing event handlers in your program code, you can configure the VISA control to place events in a queue from which you can later retrieve them. Use the `EnableEvent` method to specify the VISA events that the control should place in a queue. Set the VISA event type in the parameter for this method.

```
CWVisa1.EnableEvent cwVisaEventVXIVMEInterrupt
```

After this call, events are placed in the event queue as they occur. The maximum size of the queue is determined by the `Queue Length` property. The default size is 50 entries. If more than 50 entries are received before any are removed from the queue, the entries are discarded.

Note ***The VISA control automatically calls the*** `EnableEvent` ***on all event types except*** `Trigger` ***when a VISA session is opened. If you are concerned with events that occur at certain points in your program, use commands to empty the queue or disable the queuing of events. See the following sections,*** *Discarding Events From The Queue*, ***and*** *Disabling The Event Queue*, ***for more information.***

Checking Events in the Queue

In your program code, use the `WaitOnEvent` method to check the queue for events. The `WaitOnEvent` parameters specify the event type for which to wait, the timeout for waiting, and a return parameter for the event (if one is received). To determine if any events are currently in the event queue, specify a timeout of zero.

The following code fragment checks the event queue to determine if a `VXIVMEInterrupt` is available in the queue, and if one is available, it obtains the `StatusId` value of the interrupt.

```
Dim ReturnEvent As CWVisaEvent
Dim stat As Boolean
```

```
stat = CWVisa1.WaitOnEvent(cwVisaEventVXIVMEInterrupt,
       0, ReturnEvent)
If stat Then Text1.text = ReturnEvent.StatusId
```

If the `WaitOnEvent` method is successful, it removes the oldest event from the queue.

Discarding Events From The Queue

To empty the event queue of all of its current events, use the `DiscardEvents` method.

```
CWVisa1.DiscardEvents cwVisaEventVXIVMEInterrupt
```

Disabling The Event Queue

To stop events from being placed in the queue, use the `DisableEvent` method.

```
CWVisa1.DisableEvent cwVisaEventVXIVMEInterrupt
```

To continue queuing events, call the `EnableEvent` method again. If you call `DisableEvent` and then call `EnableEvent` later in the program, all events present in the queue when you called `DisableEvent` are still present in the queue.

Note ***When the event queue fills up, subsequent events are discarded.***

Tutorial: Using the VISA Control for Message-Based Communication

This tutorial shows you an example of using the VISA control in a simple program to control a GPIB Fluke 45 multimeter. While the Fluke 45 and this example use a specific command set, you can control most other message-based instruments in a similar manner. Substitute commands for your particular instrument in place of the Fluke 45-specific commands.

This tutorial uses Visual Basic syntax, but the discussion is in general terms so you can follow it in any compatible programming environment. Remember to adjust any code to your specific programming language. Refer to the *Building ComponentWorks Applications* chapters for information about implementing any step in different programming environments. You also can refer to the tutorial examples installed with ComponentWorks for a completed version of this example in several different programming environments.

Designing the Form

1. Open a new project and form. If you are working in Visual C++, select a dialogue-based application and name your project `MbasedExample`.
2. Load the ComponentWorks VISA control into your programming environment.

3. Place a VISA control on your form. You configure its properties in the next section.

4. Place a text edit box on the screen.
5. Set the `Name` property for the text box to `Text1`.

6. Place a button control on the form.
7. Change the `Name` and `Caption` properties to `FREQ`.

Your form should look similar to the one shown below.

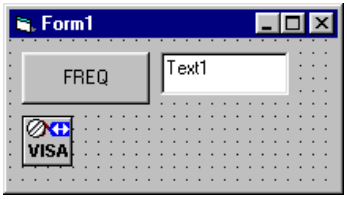

Figure 9-7. MbasedExample Form

Setting the VISA Control Properties

You normally configure the default property values of the VISA control before you develop your program code. When using the VISA control, most or all properties are set during design and do not change during program execution. If necessary, you can edit the properties of the VISA control at run time.

1. Open the custom property pages for the VISA control on the form by right clicking on the control and selecting **Properties...**.
2. In the **General** tab, select the resource name for your GPIB instrument from the **Resource Name** pull-down menu. If it is not present, make sure that the GPIB and VISA driver software is installed properly. Alternately, you can manually enter a resource name. For more information about manually entering resource names, see the *VISA Object* section earlier in this chapter.
3. Open the **RdWrt** property page for the VISA control. If your instrument uses a termination character or needs some other property change, enter the settings.
4. Open the **Test** property page for the VISA control.
5. To ensure that the control is properly communicating with the instrument, enter `"*IDN?"` in the Send String box and press the **Execute** button. If your instrument understands this command, a string similar to `"FLUKE, 45, 5005161, 1.6 D1.0"` appears in the output window. You can send any string to the instrument from this property page. For example, for the Fluke 45, enter `"val1?"` in the Send String box and press the **Execute** button. A value appears in the output window. The **Execute** button performs both write and read operations.

Developing the Code

Develop the code so that data is acquired and displayed in response to a user pressing the button. In the following steps, you define an event handler routine to be called when the **FREQ** button is pressed. In the event handler, you send a command to the Fluke 45 to read the frequency and display it in the text edit box.

1. Generate the event handler routine for the `Click` event of the **FREQ** button. The following code opens the VISA control and sends a command to the Fluke 45 to switch it into the frequency mode, reads the current value from the multimeter, and displays the value in the text box.

   Add the following code to the `FREQ_Click` subroutine. In Visual C++, remember to generate member variables for any controls referenced in the program.

   ```
   Private Sub FREQ_Click ()
      CWVisa1.Open
      CWVisa1.Write "FREQ"
      CWVisa1.Write "val1?"
      Text1.Text = CWVisa1.Read
   End Sub
   ```

2. Save the project as `MbasedExample`.

Testing Your Program

Run and test the program. Click on the **FREQ** button. The value displayed in the text edit box should match the value displayed on the Fluke. By selecting different modes, you can perform other types of measurements using the Fluke 45. Change the `"FREQ"` command sent to the instrument with a corresponding command, such as `"VDC"` for volts.

A very similar project can be used to communicate with a message-based serial or VXI device. Set the serial port configuration on the **Serial** property page to match the necessary settings for the serial instrument.

Tutorial: Using the VISA Control for Register-Based Communication

This tutorial shows you an example of using the VISA control to read the value of configuration registers for a VXI device.

This tutorial uses Visual Basic syntax, but the discussion is in general terms so you can follow it in any compatible programming environment. Remember to adjust any code to your specific programming language. Refer to the *Building ComponentWorks Applications* chapters for information about implementing any step in different programming environments. You also can refer to the tutorial examples installed with ComponentWorks for a completed version of this example in several different programming environments.

Designing the Form

1. Open a new project and form. If you are working in Visual C++, select a dialogue-based application and name your project `RbasedExample`.
2. Load the ComponentWorks VISA control and the ComponentWorks User Interface controls into your programming environment.

3. Place a VISA control on your form. You configure its properties in the next section.

4. Place a numeric edit box on the screen. Set the `Name` property for the numeric edit box to `Offset`.

   Right click on the Offset object and select **Properties...**.

   On the **Numeric** page, set **Maximum** range checking to `64`, **Values** to `Discrete`, **Base** to `0`, **Interval** to `2`, and the **Increment value** of the **Inc/Dec button** to `2`.

5. Place a second numeric edit box on the form. Set the `Name` property of the second numeric edit box to `Val`.

   Select the `Val` object.

   On the **Style** page, set **Control mode** to `Indicator` and uncheck the **Visible** box of the **Inc/Dec button** setting.
6. Add labels with the appropriate names for the two numeric edit controls.

Your form should look similar to the one shown below.

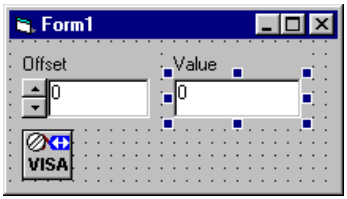

Figure 9-8. RbasedExample Form

Setting the VISA Control Properties

You normally configure the default property values of the VISA control before you develop your program code. Most or all properties are set during design and do not change during program execution. If necessary, you can edit the properties of the VISA control at run time.

1. Open the custom property pages for the VISA control on the form by right clicking on the control and selecting **Properties...**.
2. In the **General** tab, select the resource name for your VXI instrument from the **Resource Name** pull-down menu. If it is not present, make sure that the VXI and VISA driver software is installed properly. Alternately, you can manually enter a resource name. For more information about entering resource names, see the *VISA Object* section earlier in this chapter.

Developing the Code

Develop the code so that a configuration register is read and displayed in response to a user entering an offset value. For the Offset numeric edit control, define an event handler routine to be called when the offset value changes. In the event handler, read and display the configuration register at the specified offset in the numeric edit box on the user interface.

1. Generate the event handler routine for the `ValueChanged` event of the Offset control. In the event handler, open the VISA control, read a configuration register, and display the value in the numeric edit box in hexadecimal format.

   Add the following code to the `Offset_ValueChanged` subroutine. In Visual C++, remember to generate member variables for any controls referenced in the program.

   ```
   Private Sub Value_ValueChanged(Value As
     Variant,PreviousValue As Variant,ByVal OutOfRange
     As Boolean)
       Val.Value = Hex(CWVisa1.In16 (cwVxiA16Space,
          Offset.Value))
   End Sub
   ```

2. Add the following line to the `Form_Load` subroutine.

   ```
   Private Sub Form_Load()
          CWVisa1.Open
   End Sub
   ```

3. Save the project as `RbasedExample`.

Testing Your Program

Run and test the program. Try reading the value of configuration registers at various offsets. Keep in mind that all VXI devices are required to implement at least four basic configuration registers (offsets 0, 2, 4, and 6).

10

Using the Analysis Controls and Functions

This chapter describes how you can use the ComponentWorks Analysis controls to perform data analysis, manipulation, and simulation. It explains the individual controls and some of their functions and includes a tutorial that gives step-by-step instructions on using the Analysis controls in a simple program.

With the ComponentWorks analysis controls, you can perform operations such as matrix and array calculations, frequency analysis, statistical analysis and signal generation. The analysis functions are methods on different Analysis controls that are organized by functionality.

Refer to the *Building ComponentWorks Applications* chapters for information about using the ComponentWorks controls in different programming environments. You can find a detailed description of each individual analysis function in the online reference, which is available by selecting **Programs»National Instruments ComponentWorks» ComponentWorks Reference** from the Windows **Start** menu.

What Are the Analysis Controls?

ComponentWorks includes five ActiveX controls with more than 200 analysis functions. The functions are grouped into controls according to their functionality.

Control	Functionality
CWArray	Array manipulation functions
CWComplex	Complex scalars and array manipulation functions
CWMatrix	Vector and matrix algebra functions
CWStat	Statistical functions
CWDSP	Digital signal processing and signal generation functions

Analysis Library Versions

ComponentWorks is distributed with one of three different versions of the analysis library. Each version contains a different set of functions in the library. The functions available to you depend on the package of ComponentWorks you purchased.

- Base Analysis Library, *ComponentWorks Base Package*—Includes simple matrix algebra, array and complex number functions, and simple statistics functions.
- Digital Signal Processing (DSP) Analysis Library, *ComponentWorks Standard Development System*—Includes the functions of the Base Analysis Library plus DSP functions (time and frequency domain analysis, filters, windows), curve fitting functions, advanced array and complex number functions, and measurement functions.
- Advanced Analysis Library (AAL), *ComponentWorks Full Development System*—Includes the functions of the DSP analysis library plus advanced statistics and matrix algebra functions.

Although specific analysis functions or controls might not be part of your analysis library, every function and control is shown in your development environment, such as the Visual Basic Object Browser or the Visual C++ Component Gallery. When you attempt to use a function that is not included in your analysis library, an error message appears to notify you that the function is not supported. See *Adding Testing and Debugging to Your Application* section in Chapter 12, *Using the Analysis Controls and Functions*, for more information.

Table 10-1, *Analysis Control Function Tree*, lists all the analysis functions, grouped by control. The last column specifies the ComponentWorks version that includes the specific function.

- Base—Base Package (also in Standard and Full Development System)
- Standard—Standard Development System (also in Full Development System)
- Full—Full Development System

Table 10-1. Analysis Control Function Tree

Control	Function	Function Name	Development System
CWArray	**1D 2D Operations**		
	1D Maximum & Minimum	MaxMin1D	Base
	1D Array Subset	Subset1D	Base
	1D Array Reverse	Reverse1D	Standard
	1D Array Shift	Shift1D	Standard
	1D Array Sort	Sort1D	Base
	1D Array Interleave	Interleave1D	Base
	2D Array Transpose	Transpose2D	Base
	MultiDimensional Element Operations		
	Array Addition	AddArray	Base
	Array Subtraction	SubArray	Base
	Array Multiplication	MulArray	Base
	Array Division	DivArray	Base
	Absolute Value	AbsArray	Base
	Negative Value	NegArray	Base
	Linear Evaluation	LinEvArray	Base
	Polynomial Evaluation	PolEvArray	Standard
	Scaling	ScaleArray	Standard
	Quick Scaling	QScaleArray	Standard
	Array Clipping	ClipArray	Standard
	Array Clearing	ClearArray	Base

Table 10-1. Analysis Control Function Tree (Continued)

Control	Function	Function Name	Development System
CWArray (continued)	Array Setting	SetArray	Base
	Array Copying	CopyArray	Base
	Array Normalizing	NormalizeArray	Standard
	Variant Conversion	VarToDblArray	Base
	MultiDimensional Array Operations		
	Array Size	ArraySize	Base
	Sum of Elements	SumArray	Standard
	Product of Elements	ProArray	Standard
	Extract complete dimensions(s) from array	IndexArray	Base
	Array Subset	SubsetArray	Base
	Maximum and Minimum of Array	MaxMinArray	Base
	Search Array	SearchArray	Base
	Build/Concatenate Array	BuildArray	Base
	Reshape Array	ReshapeArray	Base
CWComplex	**Complex Numbers**		
	Complex Addition	CxAdd	Base
	Complex Subtraction	CxSub	Base
	Complex Multiplication	CxMul	Base
	Complex Division	CxDiv	Base
	Complex Reciprocal	CxRecip	Base

Table 10-1. Analysis Control Function Tree (Continued)

Control	Function	Function Name	Development System
CWComplex (continued)	Complex Square Root	CxSqrt	Standard
	Complex Logarithm	CxLog	Standard
	Complex Natural Log	Cxlon	Standard
	Complex Power	CxPow	Standard
	Complex Exponential	CxExp	Standard
	Rectangular to Polar	ToPolar	Base
	Polar to Rectangular	ToRect	Base
	MultiDimensional Complex Operations		
	Complex Addition	CxAddArray	Base
	Complex Subtraction	CxSubArray	Base
	Complex Multiplication	CxMulArray	Base
	Complex Division	CxDivArray	Base
	Complex Linear Evaluation	CxLinEvArray	Base
	Rectangular to Polar	ToPolarArray	Base
	Polar to Rectangular	ToRectArray	Base
CWMatrix	**Vector & Matrix Algebra**		
	Dot Product	DotProduct	Base
	Matrix Multiplication	MatrixMul	Base
	Matrix Inversion	InvMatrix	Base
	Transpose	Transpose	Base
	Determinant	Determinant	Base

Table 10-1. Analysis Control Function Tree (Continued)

Control	Function	Function Name	Development System
CWMatrix (continued)	Unit Vector	UnitVector	Base
	Trace	Trace	Full
	Solution of Linear Equations	LinEqs	Full
	LU Decomposition	LU	Full
	Forward Substitution	ForwSub	Full
	Backward Substitution	BackSub	Full
CWStat	**Statistics**		
	Mean	Mean	Base
	Standard Deviation	StdDev	Base
	Variance	Variance	Full
	Mean Squared Error	MeanSquaredError	Full
	RootMean Squared Value	RMS	Full
	Moments about the Mean	Moment	Full
	Median	Median	Full
	Mode	Mode	Full
	Histogram	Histogram	Base
	Probability Distributions		
	Normal Distribution Function	N_Dist	Full
	TDistribution Function	T_Dist	Full
	FDistricution Function	F_Dist	Full
	χ^2 Distribution Function	XX_Dist	Full

Table 10-1. Analysis Control Function Tree (Continued)

Control	Function	Function Name	Development System
CWStat (continued)	Normal Distribution Inverse Function	InvN_Dist	Full
	TDistribution Inverse Function	InvF_Dist	Full
	FDistribution Inverse Function	InvF_Dist	Full
	χ^2 Distribution Inverse Function	InvXX_Dist	Full
	Analysis of Variance		
	One-way Analysis of Variance	ANOVA1Way	Full
	Two-way Analysis of Variance	ANOVA2Way	Full
	Three-way Analysis of Variance	ANOVA3Way	Full
	Nonparametric Statistics		
	Contingency Table	Contingency_Table	Full
	Curve Fitting		
	Linear Fit	LinFit	Standard
	Exponential Fit	ExpFit	Standard
	Polynomial Fit	PolyFit	Standard
	General Least Squares Linear Fit	GenLSFit	Standard
	Interpolation		
	Polynomial Interpolation	PolyInterp	Full
	Rational Interpolation	RatInterp	Full
	Spline Interpolation	SpInterp	Full
	Spline Interpolant	Spline	Full

Table 10-1. Analysis Control Function Tree (Continued)

Control	Function	Function Name	Development System
CWDSP	**Signal Generation**		
	Impulse	Impulse	Standard
	Pulse	Pulse	Standard
	Ramp	Ramp	Standard
	Triangle	Triangle	Standard
	Sine Pattern	SinePattern	Standard
	Uniform Noise	Uniform	Standard
	White Noise	WhiteNoise	Standard
	Gaussian Noise	GaussianNoise	Standard
	Arbitary Wave	ArbitaryWave	Standard
	Chirp	Chirp	Standard
	Sawtooth Wave	SawtoothWave	Standard
	Sinc Waveform	Sinc	Standard
	Sine Waveform	SineWave	Standard
	Square Wave	SquareWave	Standard
	Triangle Wave	TriangleWave	Standard
	Frequency Domain Signal Processing		
	FFT	FFT	Standard
	Inverse FFT	InvFFT	Standard
	Real Valued FFT	ReFFT	Standard
	Real Value Inverse FFT	ReInvFFT	Standard

Table 10-1. Analysis Control Function Tree (Continued)

Control	Function	Function Name	Development System
CWDSP (continued)	Power Spectrum	Spectrum	Standard
	FHT	FHT	Standard
	Inverse FHT	InvFHT	Standard
	Cross Spectrum	CrossSpectrum	Standard
	Time Domain Signal Processing		
	Convolution	Convolve	Standard
	Correlation	Correlate	Standard
	Integration	Integrate	Standard
	Differentiate	Difference	Standard
	Pulse Parameters	PulseParam	Standard
	Decimate	Decimate	Standard
	Deconvolve	Deconvolve	Standard
	UnWrap Phase	UnWrapID	Standard
	IIR Digital Filters		
	Lowpass Butterworth	Bw_LPF	Standard
	Highpass Butterworth	Bw_HPF	Standard
	Bandpass Butterworth	Bw_BPF	Standard
	Bandstop Butterworth	Bw_BSF	Standard
	Butterworth Coefficients	BwCoef	Standard
	Lowpass Chebyshev	Ch_LPF	Standard
	Highpass Chebyshev	Ch_HPF	Standard
	Bandpass Chebyshev	Ch_BPF	Standard

Table 10-1. Analysis Control Function Tree (Continued)

Control	Function	Function Name	Development System
CWDSP (continued)	Bandstop Chebyshev	Ch_BSF	Standard
	Chebyshev Coefficients	ChCoef	Standard
	Lowpass Inverse Chebyshev	InvCh_LPF	Standard
	Highpass Inverse Chebyshev	InvCh_HPF	Standard
	Bandpass Inverse Chebyshev	InvCh_BPF	Standard
	Bandstop Inverse Chebyshev	InvCh_BSF	Standard
	Inverse Chebyshev Coefficients	InvChCoef	Standard
	Lowpass Elliptic	Elp_LPF	Standard
	Highpass Elliptic	Elp_HPF	Standard
	Bandpass Elliptic	Elp_BPF	Standard
	Bandstop Elliptic	Elp_BSF	Standard
	Elliptical Coefficients	ElpCoef	Standard
	IIR Filtering	IIRFiltering	Standard
	FIR Digital Filters		
	Lowpass Window	Wind_LPCoef	Standard
	Highpass Window	Wind_HPCoef	Standard
	Bandpass Window	Wind_BPCoef	Standard
	Bandstop Window	Wind_BSCoef	Standard
	Lowpass Kaiser Window	Ksr_LPCoef	Standard
	Highpass Kaiser Window	Ksr_HPCoef	Standard
	Bandpass Kaiser Window	Ksr_BPCoef	Standard
	Bandstop Kaiser Window	Ksr_BSCoef	Standard

Table 10-1. Analysis Control Function Tree (Continued)

Control	Function	Function Name	Development System
CWDSP (continued)	General EquiRipple FIR	Equi_Ripple	Standard
	Lowpass EquiRipple FIR	EquiRpl_LPCoef	Standard
	Highpass EquiRipple FIR	EquiRpl_HPCoef	Standard
	Bandpass EquiRipple FIR	EquiRpl_BPCoef	Standard
	Bandstop EquiRipple FIR	EquiRpl_BSCoef	Standard
	Windows		
	Triangle Window	TriWin	Standard
	Hanning Window	HanWin	Standard
	Hamming Window	HamWin	Standard
	Blackman Window	BkmanWin	Standard
	Kaiser Window	KsrWin	Standard
	BlackmanHarris Window	BlkHarrisWin	Standard
	Tapered Cosine Window	CosTaperedWin	Standard
	Exact Blackman Window	ExBkmanWin	Standard
	Exponential Window	ExpWin	Standard
	Flat Top Window	FlatTopWin	Standard
	Force Window	ForceWin	Standard
	General Cosine Window	GenCosWin	Standard

Table 10-1. Analysis Control Function Tree (Continued)

Control	Function	Function Name	Development System
CWDSP (continued)	**Measurement**		
	AC/DC Estimator	ACDCEstimator	Standard
	Amplitude/Phase Spectrum	AmpPhaseSpectrum	Standard
	Auto Power Spectrum	AutoPowerSpectrum	Standard
	Cross Power Spectrum	CrossPowerSpectrum	Standard
	Impulse Response	ImpulseResponse	Standard
	Network Functions	NetworkFunctions	Standard
	Peak Detector	PeakDetector	Standard
	Power Frequency Estimate	PowerFrequency Estimate	Standard
	Scaled Window	ScaledWindow	Standard
	Spectrum Unit Conversion	SpectrumUnit Conversion	Standard
	Threshold Peak Detector	ThresholdPeak Detector	Standard
	Transfer Function	TransferFunction	Standard

Controls

Each analysis function is a method of its corresponding control. Parameters are passed to analysis functions like any other functions. In many cases, the calculated value or data is returned as a result from the function, rather than in an output variable. This allows you to directly assign the result of the function to another part of the program, such as the user interface, or as the parameter of another function.

```
Text1.Text = CWStat1.Mean(Data)
```

Because each function is a method of a control, you must place the corresponding control in your application to use the function. You must include the name of the control in each call to an analysis function. For example, a call to the `AddArray` function, which is part of the `CWArray` control, might look like this:

```
SumArray = CWArray1.AddArray(Array1, Array2)
```

Do not assign a return variable for functions from which the information is not returned. For example, the `AutoPowerSpectrum` method returns results in the `Spectrum` and `deltaF` parameters.

```
CWDSP1.AutoPowerSpectrum Data, 0.001, Spectrum, deltaF
```

Many parameters passed to the analysis functions are of variant data type. When passing these parameters for output or return, you only need to declare them as a variant.

```
Dim Data as Variant
Data = CWDSP1.SinePattern(1024, 5, 0, 4.5)
```

Analysis Function Descriptions

Because there are many functions in the analysis libraries, individual functions are not described in this manual. Each function, with its purpose and parameters, is described in detail in the ComponentWorks online reference. The online reference also includes code examples for each function. You can access the online reference directly from most programming environments. See the chapter in this manual specific to your programming environment for more information.

Error Messages

If any analysis function encounters an error, it sends an exception back to the application, which displays a dialog box with the error number and description. The analysis functions do not return the error code from the function. Consult the Appendix D, *Error Codes*, for more information about individual error messages and how to resolve them. Error handling and debugging is described in more detail in Chapter 12, *Building Advanced Applications*.

Tutorial: Using Simple Statistics Functions

This tutorial shows you how to use some of the CWStat control statistical functions. The functions in this tutorial are part of the Base Analysis Library, which is supported by all versions of ComponentWorks.

By placing the Analysis control containing the functions you are using in your application, you can use the functions by adding them to your code manually or with the tools provided by your development environment. The Analysis controls do not have any properties that need to be edited. The only property you use with an Analysis control is the `Name` property, which you use when calling any function. For example, the default name of the CWStat controls is `CWStat1` and a call to the `Mean` function is `CWStat1.Mean`.

Although this tutorial uses the Graph control to display data arrays, you can still complete this tutorial if you do not have the user interface tools. Disregard references to the Graph control, and use your own method for displaying data arrays.

This tutorial uses Visual Basic syntax, but the discussion is in general terms so you can follow it in any compatible programming environment. Remember to adjust any code to your specific programming language. Refer to the *Building ComponentWorks Applications* chapters for information about implementing any step in different programming environments. You also can refer to the tutorial examples installed with ComponentWorks for a completed version of this example in several different programming environments.

Designing the Form

1. Open a new project and form. If you are working in Visual C++, select a dialog-based application and name your project `Stat`.
2. Load the ComponentWorks User Interface control, CWGraph, and the CWStat Analysis controls into your programming environment.

3. From the toolbox or toolbar, place a CWStat control on the form. Keep its default name, `CWStat1`.

4. From the toolbox or toolbar, place a CWArray control on the form. Name the control `CWArray1`.

5. Place a CWGraph control on the form. Keep its default name, `CWGraph1`. You can open its property page to change any of its properties.

6. Place a button on the form. Change its name and caption property to `Go`.

7. Create six text boxes on the form and name them `Max`, `MaxIndex`, `Min`, `MinIndex`, `Mean`, and `StdDev`.
8. Add a label to each text box with the following descriptions: `Maximum`, `Max Index`, `Minimum`, `Min Index`, `Mean`, and `Standard Deviation`.

Your form should look similar to the one shown below.

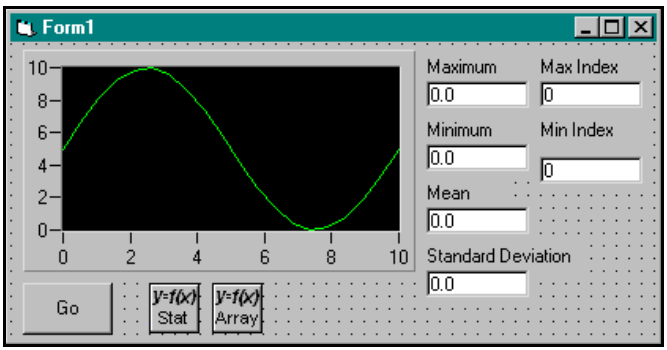

Figure 10-1. Stat Form

Developing the Program Code

When you press the **Go** button, the program generates an array of random numbers. It displays the data on the graph and also calculates and displays the following statistics of the data set: maximum, array index at maximum, minimum, array index at minimum, mean, and standard deviation.

1. Create a skeleton event handler for the `Click` event of the **Go** button.
   - In Visual Basic, double click on the button on the form to create the `Go_Click` subroutine.
   - In Visual C++, use the **MFC ClassWizard** to create the event handler routine. Right click on the button and select **ClassWizard**.
   - In Delphi, use the Object Inspector to create the event handler routine. Select the **Go** button, then press <F11> to open the Object Inspector. Select the **Events** tab. Double click the empty field next to the Click event.
2. Add code inside the event handler routine to generate an array, fill it with random data and display it on the graph. If you are working in Visual C++, you first need to add a member variable for the graph control using the MFC Class Wizard.

```
Dim data(0 To 99)

For i = 0 To 99
   data(i) = Rnd
Next i
CWGraph1.PlotY data
```

3. Add the function to calculate the statistics of the data set. Use the `StdDev` function of the CWStat control to calculate the standard deviation and mean of the dataset, and the `MaxMin1D` function of the CWArray control for the remaining values. You also need to declare a number of variables to store the different calculated values. Add the following code to the program, placing the variable declarations at the top of the event handler routine and the analysis functions after the call to the `PlotY` method from the Step 2.

```
Dim MeanVal as Variant, StdDevVal as Variant

Dim MaxVal as Variant, MaxIndexVal as Variant

Dim MinVal as Variant, MinIndexVal as Variant


CWStat1.StdDev data, MeanVal, StdDevVal

CWArray1.MaxMin1D data, MaxVal, MaxIndexVal,
   MinVal, MinIndexVal
```

4. Add the necessary code after the analysis functions to display the calculated values in the textboxes on the user interface.

```
Mean.Text = MeanVal
StdDev.Text = StdDevVal
Max.Text = MaxVal
Min.Text = MinVal
MaxIndex.Text = MaxIndexVal
MinIndex.Text = MinIndexVal
```

5. Save the project and associated files as `Stat`.

Testing Your Program

Run the program. Click on the **Go** button to generate a data set and calculate the statistical values. The result should be similar to the following illustration.

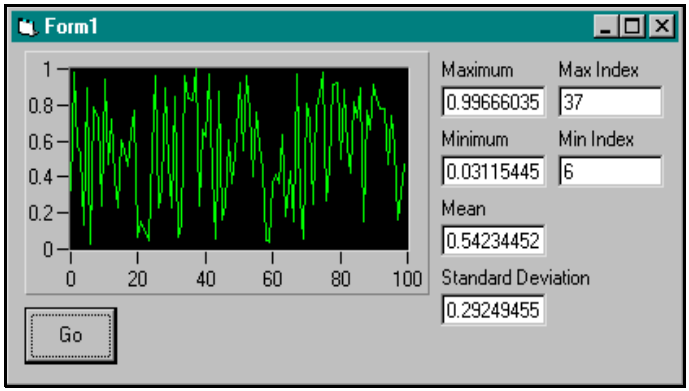

Figure 10-2. Testing Stat

When the statistical values are displayed in the text boxes, all digits of precision are displayed by default. You can edit the code that displays these values to limit the number of digits displayed. Consult your programming reference manual for information about limiting the degree of precision.

You also can use the Numeric Edit Box control to display the values and use its format string to limit the number of digits displayed.

11

Using the DataSocket Control and Tools

This chapter describes how you can use the ComponentWorks DataSocket control to read, write, or share data on a single machine or between multiple machines and includes tutorial exercises that provide step-by-step instructions for using the DataSocket tools.

Refer to the *Building ComponentWorks Applications* chapters for information about using the ComponentWorks controls in different programming environments. The software includes solutions for the tutorials in Visual Basic, Visual C++, and Delphi.

Note ***The DataSocket control is part of the ComponentWorks Standard and Full Development Systems.***

What is DataSocket?

DataSocket, both a technology and a group of tools, facilitates the exchange of data and information between an application and a number of different data sources and targets. These sources and targets include files and HTTP/FTP servers. Often, these sources and targets are located on a different computer. You can specify DataSocket sources and targets (connections) using URLs (uniform resource locators) that adhere to the familiar URL model.

DataSocket uses an enhanced data format for exchanging instrumentation style data, including data attributes and the actual data. Data attributes might include information such as an acquisition rate, test operator name, time stamp, quality of data, and so on.

Although you can use general purpose file I/O functions, TCP/IP functions, and FTP/HTTP requests to transfer data between different applications, applications and files, and different computers, you must write a significant amount of program code to do so. DataSocket greatly simplifies this task by providing a unified API for these low-level communication protocols. Transferring data across computers with DataSocket is as simple as using a browser to read Web pages on the Internet.

DataSocket Basics

To bring data into your application, connect the DataSocket control to the data source with the `Connect` method. The `Connect` method uses a URL to identify the source to which you want to connect.

Although it is not visible at run time, the DataSocket can be thought of as a connector on an application, as depicted by the following figure. When you connect to a data source, it is like plugging in a wire from the source. Because the DataSocket control parses the raw data and passes the value to your application, you can connect to different sources without having to support different data formats and protocols. If the DataSocket is connected to a dynamic source, new values are passed to the application when the value at the source changes.

When the DataSocket control has a new value loaded from the source, it stores the value in a local CWData object, which holds the data value and attributes associated with the data. For example, you can use DataSocket to convert a wave file automatically into an array of numbers that can be plotted or processed.

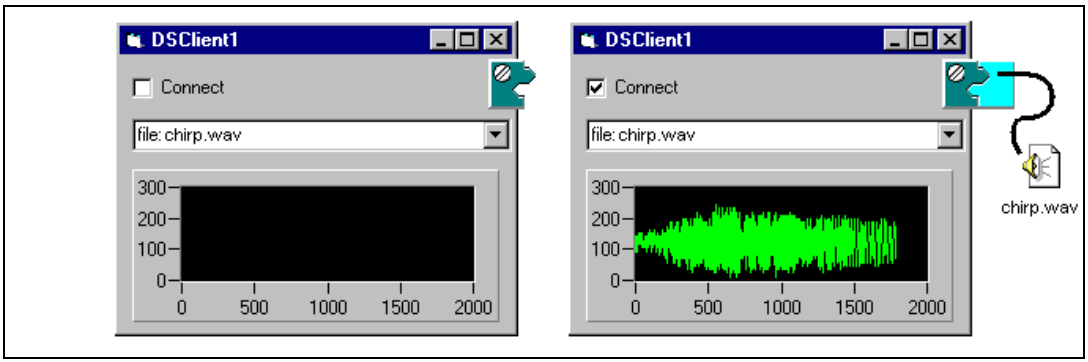

Figure 11-1. DataSocket Connection

Locating a Data Source

Specify a URL to point the DataSocket to the data source location. Like URLs you use in a Web browser, the data source locator can point to many different types of sources depending on the prefix. The prefix is called the *URL scheme*. DataSocket supports several existing schemes, including `http:` (hypertext transfer protocol), `ftp:` (file transfer protocol), and `file:` (local files). The DataSocket also supports a new scheme, `dstp:` (DataSocket transfer protocol), for connecting to DataSocket Servers. After the URL scheme, enter the data-source-path. The format of the path depends on the scheme you use.

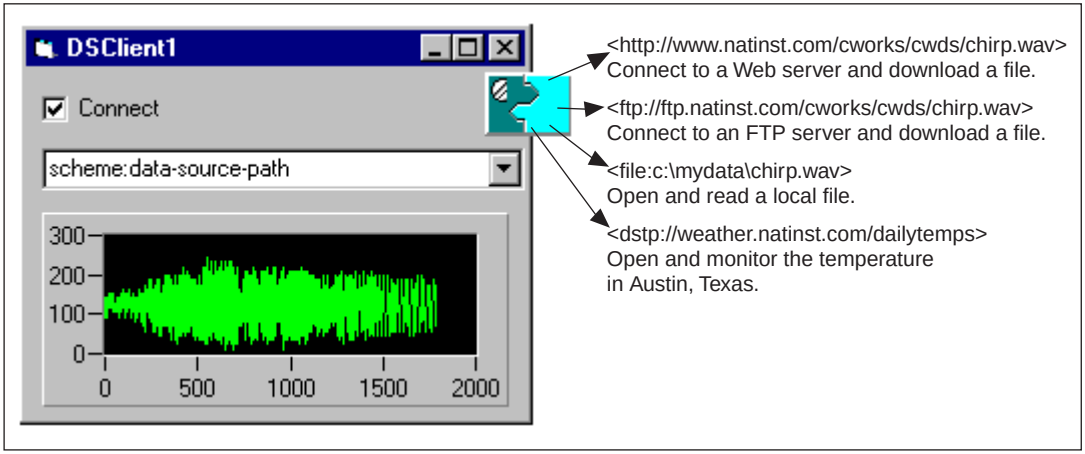

Figure 11-2. Specifying Data Source Locations

Reading Data from a Data Source

To read, load, or download data using the DataSocket control, call the `ConnectTo` method and pass in the URL and access mode. The access mode specifies whether the connection is reading or writing data. To read from a source, use the access mode `cwdsRead`.

```
'Connect to a new data source to read from.
CWDataSocket1.ConnectTo "http://host/path" cwdsRead
```

When you use the `ConnectTo` method, the URL and access mode values are saved in the `URL` and `AccessMode` properties of the DataSocket control. If you have already set these properties, you can use the `Connect` method instead.

```
' Connect using the current URL and
' Access mode property settings.
CWDataSocket1.Connect
```

As it connects and loads data, the DataSocket control can generate two events to notify your application of progress: `OnDataUpdated` and `OnStatusUpdated`.

OnDataUpdated Event

The DataSocket control generates an `OnDataUpdated` event when it receives a data value. The event passes a reference to the DataSocket CWData object.

```
Sub CWDataSocket1_OnDataUpdated(ByVal Data As CWData)
    CWGraph1.PlotY Data.Value
End Sub
```

You can use the DataSocket control `Data` property to access the currently stored value. The `Data` property returns a reference to the same CWData object that is passed to the event. From the CWData object, you can get the value or attributes associated with the current data. The data value is returned as a variant (the specific type of variant depends on the source to which the DataSocket is connected).

```
x = CWDataSocket1.Data.Value
```

Instead of waiting for an event, you might find it more convenient to check for new data. To determine if the DataSocket data has been updated since it was last read, query the value of the `DataUpdated` property. When a new value is loaded, `DataUpdated` becomes true. When the data value or any attribute is checked, `DataUpdated` reverts to false.

Use the `DataUpdated` property to check for updates in conjunction with another periodic operation, such as an operation initiated by a timer event.

```
Sub Timer1_OnTimer
    ' DataUpdated is true only if the data
    ' has been reloaded.
    If DataSocket1.DataUpdated then
        ' Reading the data property resets the
        ' DataUpdated property.
        MySub DataSocket1.Data
    End If
    ' Other timer event code.
End sub
```

Updating the Data

To reload data from the source, call the `Update` method. When the new data is loaded, the `DataUpdated` property is set to true and the `OnDataUpdated` event is fired.

```
' Update the data that DataSocket is holding.
CWDataSocket1.Update
```

If you read the `Data` property before the event occurs, you get the previous data.

Automatically Updating Data

Some data sources, such as the DataServer, support automatic notification of value changes. For example, the DataSocket Server can automatically download updates when another client changes the data item to which it is connected. When the DataSocket control receives an update, it generates an `OnDataUpdated` event and sets the `DataUpdated` property to `true`. To receive updates automatically, connect using the `cwdsReadAutoUpdate` access mode.

```
CWDataSocket1.ConnectTo sourceURL, cwdsReadAutoUpdate
```

OnStatusUpdated Event

The `OnStatusUpdated` event occurs every time you try to connect to the data source specified by the URL. The event returns parameters for the status, a system error code, and a string describing the most recent progress or error. You can use these parameters to identify the cause of a problem or to report the progress. The `OnStatusUpdated` event can occur a number of times, depending on the data source to which you are trying to connect.

```
Private Sub CWDataSocket1_OnStatusUpdated(ByVal Status
As Long, ByVal Error As Long, Message As String)
' Display the current status in the form
   Text1.Text = Message
End Sub
```

Disconnecting from a Data Source

When you finish retrieving data from the data source, call the `Disconnect` method to release system resources that are used by the data source to which you connected. If you call the `Connect` method while connected to

another source; delete the control; or unload the form it is on, the DataSocket control automatically calls `Disconnect`.

```
CWDataSocket1.Disconnect
```

Once disconnected, the DataSocket retains the last data loaded so you can continue to use it after the connection terminates. The `Update` method does not work after you disconnect.

Tutorial: Reading a Waveform

This tutorial shows you how to use the DataSocket to load a wave from several different data sources, some of which are installed with ComponentWorks and others are on the National Instruments's Web and FTP sites. To use the Internet sources, your computer must be connected to the Internet.

This tutorial uses Visual Basic syntax, but the discussion is in general terms so you can follow it in any compatible programming environment. Remember to adjust any code to your specific programming language. Refer to the *Building ComponentWorks Applications* chapters for information about implementing any step in different programming environments. You also can refer to the tutorial examples installed with ComponentWorks for a completed version of this example in several different programming environments.

Designing the Form

1. Open a new project and form. If you are working in Visual C++, select a dialog-based application and name your project `SimpleDSRead`.
2. Load the ComponentWorks DataSocket and ComponentWorks UI controls (specifically the graph) into your programming environment.

3. Place a CWDataSocket control on the form. Keep its default name, `CWDataSocket1`.

4. Place a CWGraph control on the form. Keep its default name, `CWGraph1`.

5. Place a CommandButton control on the form. Keep its default name, `Command1`.
6. Change the CommandButton caption to `Connect`.

7. Place a TextBox control on the form. Keep its default name `Text1`.
8. Place another TextBox control on the form. Keep its default name, `Text2`.

Your form should look similar to the one shown below.

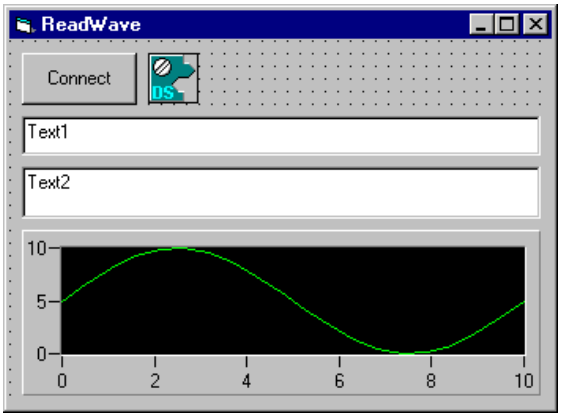

Figure 11-3. SimpleDS Form

Developing the Program Code

After completing the following steps, you can press the **Connect** button to connect the DataSocket to the data source. The DataSocket loads the data, and the graph displays the data.

You can specify the data source by typing the URL in the text box. Each time the DataSocket connects to a new source, it disconnects from the previous source.

1. Create a skeleton event handler for the `OnDataUpdated` event of `CWDataSocket1`.
   - In Visual Basic, double click on the DataSocket control that you placed on the form.
   - In Visual C++, use the MFC ClassWizard to create the event handler routine. Right click on the DataSocket control and select **ClassWizard**.
   - In Delphi, use the Object Inspector to create the event handler routine. Select the Slide control, press <F11> to open the object inspector, select the **Events** tab, and double click on the empty field next to the `OnDataUpdated` event.
2. Add the following code inside the event handler routine. If you are working in Visual C++, first add a member variable for each control to the application dialog class.
   - Visual Basic:
     ```
     CWGraph1.PlotY Data.Value
     ```
   - Visual C++:
     ```
     m_CWGraph1.PlotY(Data.GetValue(), COleVariant(0.0),
       COleVariant(1.0), COleVariant(1.0));
     ```
   - Delphi:
     ```
     CWGraph1.PlotY(Data.GetValue(), 0, 1, TRUE);
     ```
3. Add the following code to the `CWDataSocket1_OnStatusUpdated` event. Remember to adjust the syntax for your programming language.

   ```
   Text2.Text = message
   ```
4. Add the following code to the `Command1_Click` event routine. Remember to adjust the syntax for your programming language.

   ```
   CWDataSocket1.ConnectTo Text1.Text, cwdsReadAutoUpdate
   ```
5. Save the project and associated files as `SimpleDSRead`.

Testing Your Program

Run the program. Initially, the graph does not display any data because the DataSocket is not connected to a source. To connect to a data source, enter a URL in the text box and press **Connect**.

Note ***You must establish a network connection before you can connect to an Internet data source. If you normally dial into an Internet service provider over a phone line, do so before attempting to connect to a data source on the Internet.***

You do not need an Internet connection to read data from local files or connect to DataSocket Servers on the same computer.

The following list contains URLs you can use to test the program.

- To connect to and load data from a local file, enter a file URL. ComponentWorks includes sample files in the ComponentWorks installation directory.

  `file:c:\Program Files\National Instruments \ComponentWorks\tutorials\Visual Basic\chirp.wav`

- The following list specifies URLs to which you can connect with the DataSocket. If the URLs do not work, use your Web browser to locate new URLs at `www.natinst.com/cworks/datasocket`.
  - To connect to and load data from an http (Web) server, enter `http://www.natinst.com/cworks/datasocket/chirp.dsd`
  - To connect to and load data from an ftp server, enter `ftp://ftp.natinst.com/support/compworks/datasocket /chirp.dsd`
  - To connect to and load data from a DataSocket, enter `dstp://weather.natinst.com/weather/windspeed`

    This site has live weather information from our corporate headquarters in Austin, Texas.

Writing Data to a Data Target

When you connect to a URL with one of the read modes, the URL identifies a data source. When you connect to a URL with a write mode, the URL identifies a data target. That is, you read data from a data source and write data to a data target.

To connect to a data target, use the `Connect` method with the `cwdsWrite` access mode. Once connected, the DataSocket writes the data it is currently holding.

```
CWDataSocket1.Data.Value = x
CWDataSocket1.ConnectTo targetURL, cwdsWrite
```

When the connection is complete and the data is written to the target, the DataSocket fires the `OnDataUpdated` event and sets the `DataUpdated` property to true. `DataUpdated` reverts to false once the `Data` value or attributes are modified or checked.

Note ***Some data sources, such as DataSocket Servers and local files, can be either a data source or a data target, determined by the access mode used to connect to it. Other data sources, especially*** `http:` ***and*** `ftp:` ***sources, are read only and cannot be used as a data target.***

Updating a Data Target

The DataSocket rewrites data to the data target with the `Update` method. When the new data has been written, the `OnDataUpdated` event is fired and the `DataUpdated` property is set to true.

```
'Update the data target with a new value.
CWDataSocket1.Data.Value = x
CWDataSocket1.Update
```

Use the `Update` method to write new data to a DataSocket Server or a local file each time you run an experiment or acquire data. When a data target is updated, the existing value is replaced completely with the new value and attributes. The DataSocket control completely replaces the existing value with the new value and attributes.

Automatically Updating a Target

When you connect to the data target using the `cwdsWriteAutoUpdate` access mode, the DataSocket updates the target every time the data value or attributes are set.

```
CWDataSocket1.ConnectTo targetURL, cwdsWriteAutoUpdate
```

If you want to set the value and attributes in one operation, use a second CWData object to prepare the data, and then use the `CopyFrom` method to copy the value and attributes.

```
' Set the value and attributes in a
' second CWData object.
Dim cwd as New CWData
cwd.Value = x
cwd.SetAttribute "SampleRate", 10000
```

```
' Copy the data from the second object.
' into the CWData owned by the DataSocket.
' After the copy is done the target will be updated.
CWDataSocket1.Data.CopyFrom cwd
```

Working with CWData

The DataSocket control consists of two objects. The first is the CWDataSocket object. The top-level object manages the process of connecting to data sources or targets, transferring and parsing data, and firing events. The second object, CWData, holds the data that has been read or that will be written.

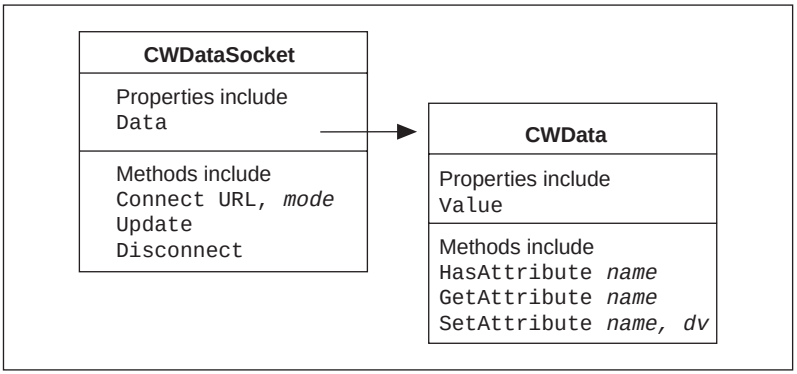

Figure 11-4. DataSocket Control

Because the CWData object is part of the CWDataSocket control, operations that read or modify the data value or attributes notify the DataSocket. Therefore, the DataSocket knows when data loaded from a source has been read or when a data target needs to be updated. When the DataSocket control is connected to a Read source, it prohibits modifications to the CWData objects it owns.

As shown in the proceeding figure, the `Data` property on the DataSocket returns a reference to a CWData object, which holds the value and attributes. CWData contains a property for the primary value of the data and methods to access attributes of the data. If the DataSocket is connected to a data source, the CWData object it owns is read only. For example, if `cwd` is a reference to a CWData object owned by a CWDataSocket that is connected to a data target with the `cwdsWriteAutoUpdate` mode, setting its value; calling `CopyFrom`; or setting its attributes causes the DataSocket to write the new value to the data target.

```
x = CWDataSocket1.Data.Value
```

Working with Attributes

You can use attributes to provide information such as the time the value was calculated or acquired, the units in which the value is expressed, the equipment used to generate the value, or any other properties that you want to define.

Use the `SetAttribute` method to change an existing attribute value or, if the attribute is not present in the data, add the attribute.

```
CWDataSocket1.Data.SetAttribute "Units", "volts"
```

To determine if existing data has a specific attribute, use the `HasAttribute` method.

```
x = CWDataSocket1.Data.HasAttribute ("Units")
' x is true if the attribute has a "Units" attribute,
' false otherwise.
```

To get the value of an attribute, use the `GetAttribute` method. `GetAttribute` accepts an optional parameter, `Default`. If the attribute you are trying to get is not present in the data, `Default` is returned.

```
x = CWDataSocket1.Data.GetAttribute ("Units", "volts")
' x is the value for the "Units" attribute or "volts"
' if the attribute is not present.


x = CWDataSocket1.Data.GetAttribute ("Units",  "")
' x is the value for the "Units" attribute or an empty
' string if the attribute is not present.


x = CWDataSocket1.Data.GetAttribute ("Units", Empty)
' x is the value for the "Units" attribute or an empty
' variant if the attribute is not present.
' “Empty” is a predefined value in VB that
' has the value of an empty variant.
```

To delete an attribute, use the `DeleteAttribute` method.

```
CWDataSocket1.DeleteAttribute "Units"
```

Standalone CWData Objects

You can create a CWData object and use it independently of the DataSocket control. To preserve the data value and attributes, use standalone CWData objects to hold copies of the data loaded by a DataSocket control.

```
' Store a copy of the CWData value in a variant.
' Attributes are not copied.
Dim v as Variant
v = CWDataSocket1.Data.Value

' Store a copy of the value and all attributes in a
' CWData object.
Dim cwd as new CWData
cwd.CopyFrom CWDataSocket1.Data
```

You can use CWData on its own as a variable that can hold both a regular value and attributes.

```
Dim cwd as new CWData
cwd.Value = 50.0
cwd.SetAttribute "Units",  "volts"
```

Use the `CopyFrom` method to copy the value and attributes from one CWData object to another.

```
cwd1 = cwd2        ' Only copies the value.
cwd1.CopyFrom cwd2 ' Copies the value and attributes.
```

Note ***Each CWData object knows if it is owned by a CWDataSocket control or if it is created as a standalone object. If it is owned, the control is notified of changes made to the data or attributes. For example, if*** `cwd` ***is a reference to a CWData object owned by a CWDataSocket that is connected to a data target with the*** `cwdsWriteAutoUpdate` ***mode, setting its value, calling*** `CopyFrom`***, or setting its attributes causes the DataSocket to write the new value to the data target.***

Setting Up a DataSocket Server

The DataSocket Server is a separate executable used to communicate and exchange data between two applications using DSTP (DataSocket transfer protocol). When you run it on your computer, you make data easily accessible to other DataSocket applications on the same computer or other computers connected though a TCP network, such as the Internet.

To launch the DataSocket Server, run **DataSocket Server** from the **National Instruments ComponentWorks** group in the Windows **Start** menu. Alternatively, you can run `DataSocketServer.exe` located in the `\ComponentWorks\DataSocketServer` directory. To make the DataSocket Server run every time you launch your computer, place a link to `DataSocketServer.exe` in your `Startup` folder.

Requirements for Running the DataSocket Server

The DataSocket Server requires support for TCP/IP networking on your machine. To check your configuration, launch the DataSocket Server. If you have a working Web browser on your computer, the TCP/IP driver should be installed.

Checking the Status of the DataSocket Server

When the server is running, you see a tray icon indicating that the server is up and running. To check on its status, double click on the icon or right click and choose **Show DataSocket Server** from the popup menu.

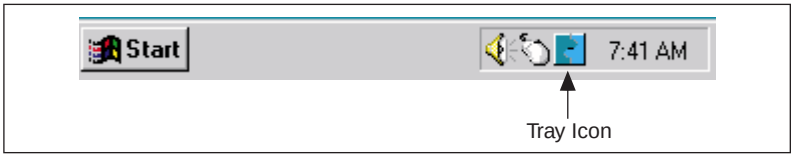

Figure 11-5. DataSocket Server Tray Icon

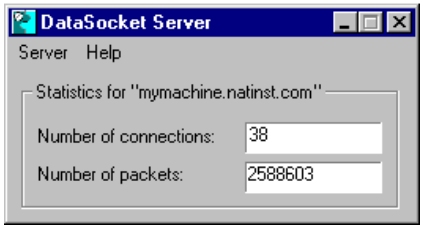

Figure 11-6. DataSocket Server Status Window

Creating Data Items on the Server

The default configuration of the server allows the local machine to create new data items and to change the data in these data items. Remote machines can connect to the server and read items. To create an item, run a client that uses a DataSocket control to write items to the server on your machine. When a DataSocket writes data to a data server using a new `DataItem` path, the server automatically creates a new entry and stores the value in that new entry. Your line of code should look similar to the following.

```
'A DataSocket client that writes to a DataSocket Server.
CWDataSocket1.ConnectTo "dstp://localhost/DataItem",
  cwdsWrite
```

Connecting to Data Items and Reading Them

You can connect to data items and read them just as you create data items or write to them. However, use the read access mode instead of the write access mode.

```
'A DataSocket client that reads from a DataSocket Server.
CWDataSocket1.ConnectTo "dstp://localhost/DataItem",
  cwdsReadAutoUpdate
```

If the reading client connects using `cwdsRead`, it gets updates only after calling the `Update` method on the DataSocket. If it uses `cwdsReadAutoUpdate`, it gets updates automatically when another client changes the value. If a reading client attempts to connect to a data item that does not exist, the connection completes, but no data is returned. Furthermore, the `OnDataUpdated` event does not fire until a writing client connects to the same data item and actually writes a value to it. You can start reading clients before launching the writing clients. When a writing client is launched, data is sent to the waiting reading clients.

Tutorial: Sharing Data between Applications

This tutorial shows you how to use the DataSocket Server to share data between different applications. The application is included in the ComponentWorks directory.

Note ***The host name*** `localhost` ***in this tutorial tells the DataSocket to connect to a server on the same machine. Use*** `localhost` ***when you do not want to hard code the local host name into applications.***

1. Launch the application `\ComponentWorks\tutorials\Visual Basic\ch11DSSReader.exe`.
2. Enter the URL `dstp://localhost/wave`.
3. Press the **Connect** button. The DataSocket connects to the server. Because the item is not yet created, no data is returned.
4. Launch the application `tutorials\Visual Basic\ch11DSSWriter.exe`.
5. Move the slide pointers to change the wave displayed in the graph. Because `ch11DSSWriter` has not connected to the server, `ch11DSSReader` is not affected.
6. Enter the URL `dstp://localhost/wave` in `ch11DSSWriter` and press **Connect**. `ch11DSSReader` gets the wave data item value each time it is set by the other client.

7. Launch another copy of `ch11DSSReader`, enter the same URL, and press **Connect**. This application immediately gets a copy of the data because the server already has a value for the item.
8. Move the slide pointer on the `ch11DSSWriter` application again. Notice that both applications show the same wave.

If you have another machine with ComponentWorks installed, you can run the `Ch11DSSReader` application on them. On other machines, you need to enter the actual host name of the server, not `localhost`. To determine the host name of the server, display the status window on the DataSocket Server.

Configuring the DataSocket Server

The default configuration of the DataSocket Server works for many intranet applications. By default, only programs running on the same computer as the server can create items or write to items, whereas applications on the same computer or other computers can read items. Items exist only as long as there is at least one DataSocket client connected to read or write the item's value. When no connections to the item remain, the DataSocket Server releases the item and its value.

Use the DataSocket Server Manager to change the default settings. You can specify which machines can create items, write items, and read items. You can specify items that should be automatically created and given an initial value when the server is started. The DataSocket Server never releases predefined items, so their values exist even when no DataSocket client is connected to them. Predefined items can have special read and write access groups so different machines can have different access to different items. Changes that you make to the server take affect the next time that you launch the DataSocket Server.

Items created dynamically can be read by hosts in the **Default Readers** group and modified by hosts in the **Default Writers** group.

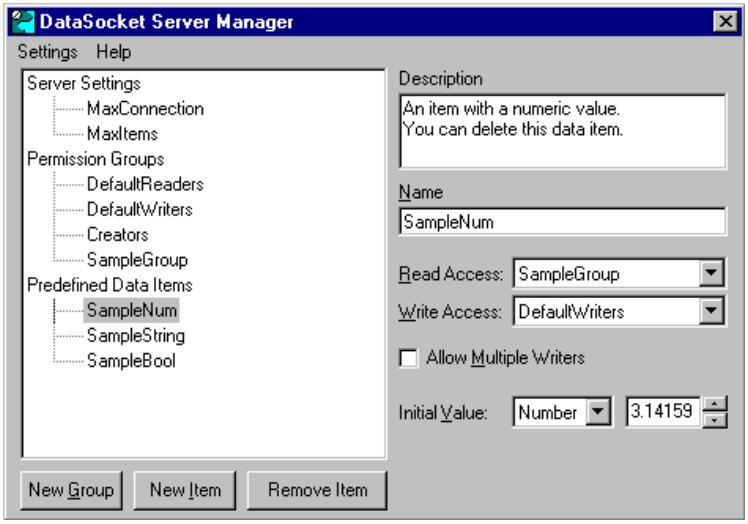

Figure 11-7. DataSocket Server Manager

12

Building Advanced Applications

This chapter discusses how you can build applications using more advanced features of ComponentWorks, including advanced data acquisition techniques, the DSP Analysis Library, and advanced user interface controls and offers techniques for error tracking, error checking, and debugging.

Using Advanced ComponentWorks Features

This section illustrates advanced data acquisition techniques, such as pretriggering and using start, stop, and pause conditions through real application examples. The examples demonstrate how to incorporate the DSP Analysis Library and use the spectrum functions. Finally, advanced user interface control features, such as graph cursors and multiple axes and pointers, are presented.

This chapter concentrates on the key features of sample applications located in the `\ComponentWorks\tutorials` directory. You can customize these examples to implement the advanced features in your own applications.

A Virtual Oscilloscope

The Virtual Oscilloscope application uses the ComponentWorks User Interface and Data Acquisition analog input controls and a DAQ board to build a simple one-channel oscilloscope. Load the sample program from `\ComponentWorks\tutorials` into your development environment to follow the discussion.

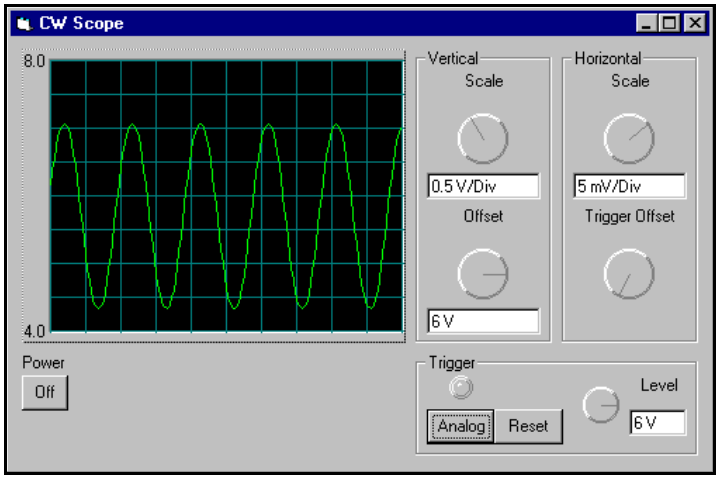

Figure 12-1. Virtual Oscilloscope

Depending on the state of the trigger mode button, the application acquires data in a single-shot, continuous, or analog trigger continuous operation. When the data is returned to the `AcquiredData` event, it is plotted on the graph. The vertical controls on the scope adjust the Y axis of the graph, while the horizontal and trigger settings affect the AI control. Any AI control property that the user does not control at run time, such as the device and channel number, is set directly in the AI property pages. By default, channel 1 on device 1 is used for the acquisition. Wire your signal accordingly or change these values in the property pages.

Data Acquisition Stop Condition Modes

The scope can run in one of three acquisition modes—single, continuous trigger, and analog continuous software trigger—which you select by using the **Single**/**Cont**/**Analog** button in the Trigger section of the form. The first two modes work well with any input signal, while the analog trigger mode works best with a periodic dynamic input signal.

The three trigger modes correspond to three types of stop conditions on the data acquisition analog input control. For example, the event handler for the **Analog** button programmatically sets a new value for the property `CWAI1.StopCondition.Type` according to the state of the button, as shown in the following code for the analog trigger mode.

```
CWAI1.StopCondition.Type = cwaiSWAnalog
```

The DAQ control or header file defines the constant `cwaiSWAnalog`. You can retrieve any `StopCondition.Type` constant value by using the Object Browser in Visual Basic, consulting the online documentation, or looking in the header file for your environment. After changing the property value, the event handler of the **Trigger** button also reconfigures and restarts the acquisition. It is important to set the analog trigger level to a value within the range of the input signal.

Data Acquisition Pretriggering

When the scope is in analog trigger mode, you can move the trigger point along the horizontal axis of the graph by using the **Trigger Offset** knob. In the code, the event handler for the knob changes the `PretriggerScans` property of the AI control.

```
CWAI1.StopCondition.PreTriggerScans = TOffset.Value
```

Without pretriggering, the application acquires all scans after the stop condition trigger (analog trigger). When the number of pretrigger scans is greater than zero, the application acquires the number of scans specified before the trigger occurs. After the trigger occurs, the array of data returned to the `CWAI1_AcquiredData` event handler contains data acquired both before and after the trigger. You can set the `PretriggerScans` property and all the stop trigger properties through the property pages of the DAQ AI control. Pretriggering works the same way with the analog software, the analog hardware, and digital hardware stop triggers.

User Interface Value Pairs

Value pairs are User Interface control features for assigning names to specific values on a scale or axis in the graph, slide and knob controls. All the value pairs for a given axis are stored in the ValuePairs collection object of that axis. Each value pair object consists of a name and a value corresponding to a scale or axis of the particular control. Although you usually set value pairs through the property page of the control, you can add, edit, or delete them programmatically.

After you assign value pairs, such as the settings for the **Vertical Scale** knob, you can limit the allowed values for a control to the predefined value pairs. To do this, set the control to `Value Pairs Only` on the **Style** page of the property sheet. In the Virtual Scope application, the vertical and horizontal scale knobs are **Value Pairs Only** controls. This way, the control is limited to preset settings and the program can retrieve the name, value, or index of the currently selected value pair. The application uses the value of the value pair to update the appropriate property on the graph or DAQ AI

control and uses the name of the value pair to update the appropriate text display.

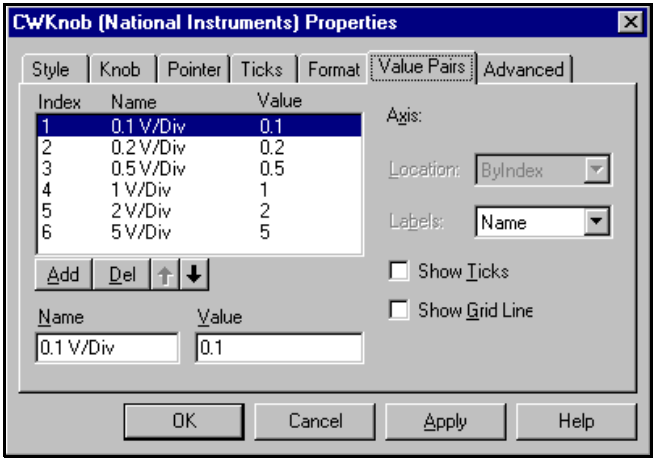

Figure 12-2. Knob Property Pages—Value Pairs Page

The following code shows how to retrieve the value, name, and index from the currently selected value pair of a knob control.

```
CWKnob1.Value
CWKnob1.Axis.ValuePairs(CWKnob1.ValuePairIndex).Name
CWKnob1.ValuePairIndex
```

You can specify multiple axes on a graph and associate value pairs with each individual axis.

Virtual Spectrum Meter

The Virtual Spectrum Meter application, which you can find in `\ComponentWorks\tutorials`, uses the DSP analysis functions to build a simple spectrum analyzer. The data can either be acquired with a DAQ board or simulated. If you do not have a DAQ board, select `Simulation` with the **Data Source** switch and characterize your signal in the Data Simulation section. The UI controls are used to display the information as

well as control the operations of the program. The following screen shows the application.

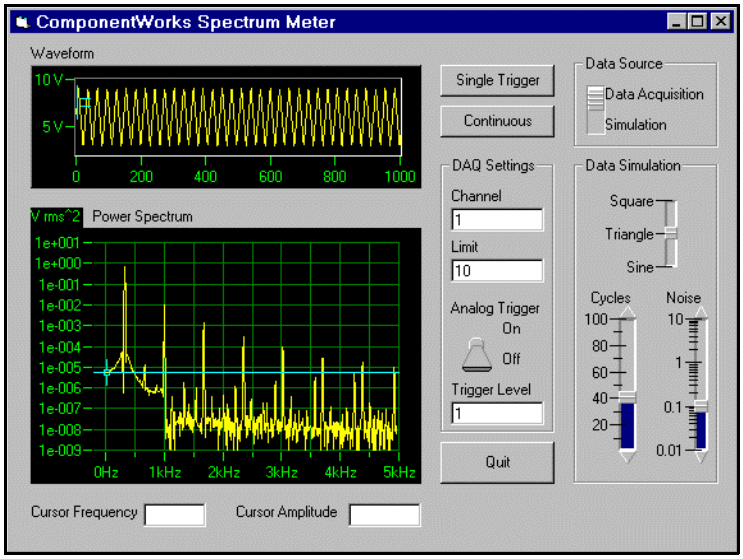

Figure 12-3. Virtual Spectrum Meter

Once the Spectrum Meter is started, you can select either a **Single Trigger** or a **Continuous Trigger** acquisition. The single trigger takes a snapshot of the incoming signal so you can study the spectrum of the acquired signal. Use the continuous trigger to monitor changes in a signal as they occur. You can set some DAQ parameters, such as the channel, using the DAQ settings section of the user interface. The input limit specifies the maximum expected absolute voltage on the input signal and determines the optimal gain to use on the acquisition process. You also can use an analog trigger with your data acquisition. Set other DAQ parameters, such as the device number, in the AI control property pages.

Use the cursors on the graphs to measure the amplitude of the incoming waveform and the frequency of any specific point in the spectrum. When you move one of the cursors, the corresponding display at the bottom of the user interface is automatically updated.

DSP Analysis Library

The digital signal processing (DSP) analysis functions are part of the ComponentWorks Standard and Full Development Systems. DSP functions include Fourier and Hartley transforms, spectrum analysis,

convolution and correlation of data sets, and digital windowing and filtering of data. If you have installed the ComponentWorks Base Package, you cannot run these functions from your development environment. However, you can examine the Spectrum Meter project and code and run the precompiled executable in the `\ComponentWorks\tutorials` folder. To use the DSP functions, you must load the Analysis controls into your environment, and then place the CWDSP control on your form.

The Virtual Spectrum Meter application contains all the analysis functions in the `AnalyzeAndGraph` subroutine, which is shown below. The application passes the data (acquired or simulated) to this routine for analysis and display.

```
Private Sub AnalyzeAndGraph(WaveformData() As Variant,
  SampleInterval As Double)

    Dim WindowedData As Variant
    Dim FreqInterval As Variant
    Dim SpectrumData As Variant
    Dim HalfSpectrumData As Variant

    WaveformGraph.PlotY WaveformData
    WindowedData = CWDSP1.HamWin(WaveformData)
    CWDSP1.AutoPowerSpectrum WindowedData,
    SampleInterval, SpectrumData, FreqInterval,
    SpectrumData, freqinterval
    HalfSpectrumData = CWArray1.Subset1D(SpectrumData,
      2, 510)
    SpectrumGraph.PlotY HalfSpectrumData, FreqInterval,
    FreqInterval * 2
End Sub
```

The analysis procedure consists of the Hamming window (`HamWin`) and the auto power spectrum function (`AutoPowerSpectrum`). The Hamming window reduces the effect of spectral noise leakage in the spectrum because of the finite length sample of a continuous signal. The auto power spectrum function is a single-sided, scaled spectrum that you can graph directly. The `AutoPowerSpectrum` function also calculates the frequency resolution of the spectrum (`FreqInterval`) using the `SampleInterval` value passed to `AnalyzeAndGraph`. The frequency resolution is used in the `PlotY` method of the graph for scaling the X axis of the spectrum graph.

The DSP Analysis Library also includes the `SpectrumUnitConversion` function for scaling the calculated spectrum between different formats, including linear, dB, and dBm, combined with Vrms, $Vrms^2$, Vpk, and Vpk^2, as well as amplitude and power spectral densities.

The DSP functions include FFT and inverse FFT algorithms for low-level spectrum calculations. Time domain functions, such as Convolution, Correlation, Differentiate, and Integrate, are included along with a number of different windowing, IIR (infinite impulse response), and FIR (finite impulse response) digital filtering functions.

Cursors

The Virtual Spectrum Meter application form has cursors on the graph for marking sections of the plots. Use the two cursors on the waveform graph to mark the minimum and maximum values of the acquisition waveform and measure the waveform amplitude. The cursor on the spectrum graph marks a particular point in the spectrum and displays the associated frequency.

To use a cursor on a graph, create and configure one or more cursors in the property pages of the graph. To create additional cursors, press the **Add** button in the **Cursors** tab and configure each cursor. The **Snap Mode** specifies whether the cursor jumps (snaps) to a point on the nearest plot or can be placed freely on the graph. When you select `Point on selected plot` for **Snap Mode**, you also can connect the cursor to a particular plot.

The following screen shows the property pages for the cursors on the waveform graph of the Virtual Spectrum Meter application.

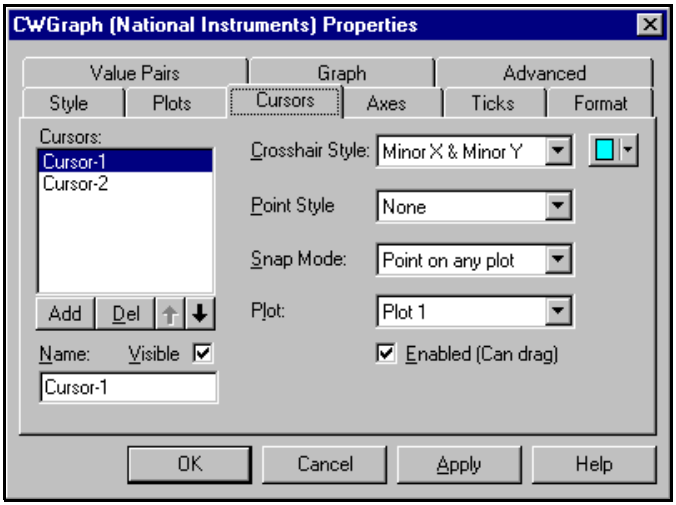

Figure 12-4. Graph Property Pages—Cursors Property Page

Individual cursors are represented in the object hierarchy by Cursor objects contained in the Cursors collection of the Graph control. You can manipulate individual cursors programmatically by following the standard conventions of working with collections and their objects. For example, reference the cursors with the name of the graph, cursor index, and cursor property.

```
Max = WaveformGraph.Cursors.Item(2).YPosition
```

Use event handler subroutines associated with the graph to process any user interactions with the cursors. There are four events on the graph relating to the cursors, of which the `CursorChange` and `CursorMouseUp` routines are the most commonly used. Generate the event handler skeleton for the `CursorChange` event, which is similar to the following.

```
Private Sub Graph_CursorChange(CursorIndex As Long, XPos
  As Variant, YPos As Variant, bTracking As Boolean)

End Sub
```

The event handler provides the index of the cursor (which you use to determine the cursor that generated the event), as well as the X and Y coordinates of the cursor. For the waveform graph cursors, the Virtual Spectrum Meter application reads the Y position of the two cursors to calculate the amplitude of the waveform data, as shown below.

```
Private Sub WaveformGraph_CursorChange(CursorIndex As
  Long, XPos As Variant, YPos As Variant, bTracking As
  Boolean)

   Dim Amplitude As Double

   Amplitude =
     Abs(WaveformGraph.Cursors.Item(1).YPosition -
     WaveformGraph.Cursors.Item(2).YPosition)

   CursorAmplitude = CStr(Round(Amplitude, 2)) + " V"

End Sub
```

Graph Track Mode

The `TrackMode` property of the graph determines how the graph reacts to mouse actions and which events the graph generates at these times. By default, the `TrackMode` property is set to `cwGTrackDragCursor`, which allows the mouse to move the cursors on the graph. Moving a cursor generates the `CursorChange` event.

The `TrackMode` property generates events when the mouse interacts with the plot area as a whole or with individual plots on the graph. With these

settings, you can use the mouse to select a specific plot or detect when the user moves the mouse over the plot area.

You also can use `TrackMode` to select panning and zooming. When `TrackMode` is set to **Pan**, any click-and-drag action on the graph shifts the contents of the graph following the mouse movement. When `TrackMode` is set to **ZoomRect**, any click-and-drag action on the graph draws a rectangular outline on the graph. When the mouse button is released, the graph zooms to the dimensions of the outline. For these two operations, you can select X axis, Y axis, and XY axis modes, limiting the motion to the specified axes.

A Virtual Data Logger

The Virtual Data Logger application records real time phenomena on multiple channels at slow rates over an extended period of time. It can either simulate the data being acquired or acquire data with a data acquisition card. The acquired data is logged to a serial ASCII file. Although it makes the file larger in size than binary format, ASCII format allows you to read the file in many other applications.

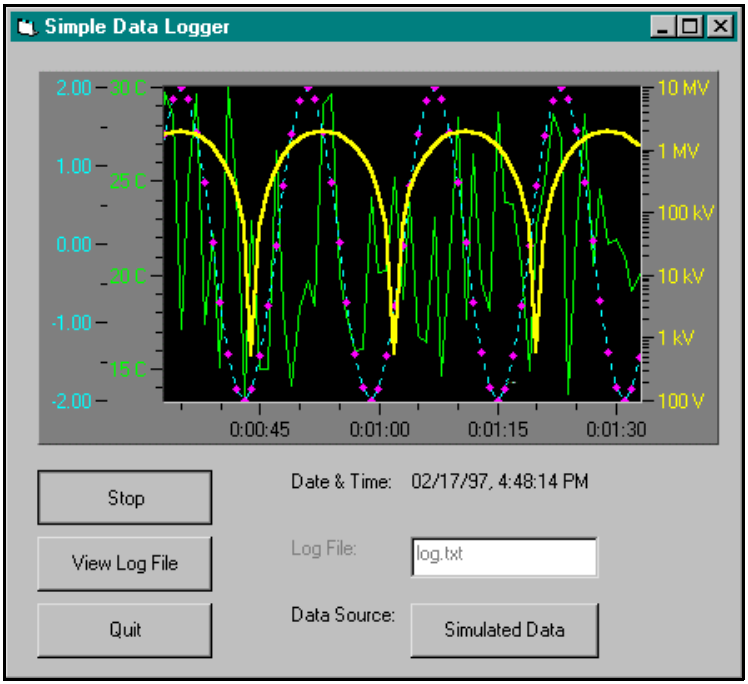

Figure 12-5. Virtual Data Logger

By default, the **Data Source** is set to simulated data, but you can acquire data from a DAQ board. If you use a DAQ board to acquire the data, remember to set the properties for the DAQ AI control accordingly.

The **Start/Stop** button is an on/off style for starting and stopping the acquisition/logging process. The **View Log File** button opens the recorded file using the Windows Notepad application. By default, the application does not clear the log file when you start the logging process. Rather, it appends the data to any existing data in the file.

Multiple Graph Axes

You can configure multiple Y axes for a graph using its property pages. On the **Axes** page, use the **Add** and **Del** buttons to add and delete axes. For each axis, select the range and style you want to use. After configuring the axes, switch to the **Ticks** page. Select each axis in the pulldown ring and configure the labels and ticks for each axis.

You can predefine the properties of multiple plots on the **Plots** page. By default, each plot is assigned to the same Y axis on the graph, but you can assign plots to different Y axes of a graph. When an application plots or charts data on the graph, the Y axis specified for a particular plot determines the scaling of that plot. If you generate more plots on a graph than there are predefined plots, the template plot style and the first Y axis are used for each undefined plot.

You can set plot or axes properties in your program code. Plots and axes are individual objects (of type Plot and Axis) that are stored in corresponding collection objects (Plots and Axes, respectively). To change a property of an individual axis or plot, you must select the collection and then the individual object in the collection, followed by the property to read or write.

For example, you can reference individual Plot objects by using the `Item` method on the collection. The methods of the Plots collection—such as `Add`, `Item`, and `Remove`—make changes to the collection as a whole, while the properties of the Plot object—such as `Name`, `AutoScale`, and `LineColor`—are individual settings for one plot.

Objects in a collection are referenced by their one-based index using the `Item` method, as in the following examples.

```
' Refers to the first axis on the graph (X axis)
CWGraph1.Axes.Item(1)

' The second plot on the graph (default name Plot-2)
CWGraph1.Plots.Item(2)

' Retrieves/sets the name of the second (first Y ) axis
CWGraph1.Axes.Item(2).Name
```

To assign simple properties—properties that contain Boolean, double, integer, variant, or string values—to a plot or axis, use the property name as you would any other variable in your program. The following example changes the name of a plot on a graph.

```
CWGraph1.Plots(2).Name = "Temperature"
```

You can assign an object as a property of another object. Because each plot on a graph is assigned a specific Y axis that is used for data scaling, you can assign a new axis (an object) as a property to a plot. In most programming environments, assign the object as you would any other variable. In Visual Basic, you must use the `Set` keyword, which differentiates the assignment of data from the assignment of objects. The following Visual Basic code assigns the third axis as an object to the `YAxis` property of the first plot.

```
Set CWGraph1.Plots.Item(1).YAxis =
  CWGraph1.Axes.Item(3)
```

Graph Axes Formats

The axes on the graph, as well as the slide and knob, support special formatting modes you can use to edit the labels on the tick marks. In addition to simple formatting, such as using exponential or engineering notation, you can add alphanumeric characters such as units (for example, `Hz` or `V`) and convert to currency, percentage, or time formats. The time format includes options for time and date.

Two more advanced formatting modes are scaling and symbolic engineering. Use the scaling format mode to automatically perform simple mathematical functions (addition, subtraction, multiplication, and division) on your data before displaying it on your graph. For example, if you are using an IC temperature sensor whose output in volts corresponds to the temperature divided by 100, you can specify your axis to automatically scale the data by `*100`. The data displayed on this axis is automatically multiplied by 100 without any changes in the program code.

You can use a symbolic engineering format when displaying units with your tick labels. The symbolic engineering format adds prefixes such as *k* for kilo and *m* for milli before the units. This format is useful for logarithmic scaling.

File Input/Output

The file I/O functions in the Virtual Data Logger application demonstrate how to perform simple file I/O in a program. Because file I/O is not part of the ComponentWorks functionality, the functions vary between different programming environments. Consult the data logger example in your programming environment for an example of how to perform file I/O.

All the file I/O functions are in the `LogData` subroutine, which you can use as a template for other input/output routines. The application uses a sequential ASCII file to store the data and appends new data to any existing data already in the file.

Adding Testing and Debugging to Your Application

Although they vary depending on the programming environment, debugging tools normally include features such as breakpoints, step-run modes, and watch windows.

Error Checking

ComponentWorks controls can report error information to you and to the application in a number of different ways:

- Return an error code from a function or method call
- Generate an error or warning event
- Throw an exception handled by your programming environment

The type of error reporting depends on the type of application and the preference of the programmer.

By default, all ComponentWorks controls generate exceptions when errors occur, rather than returning error codes from the methods. However, the DAQ and Instrumentation controls have a property, `ExceptionOnError`, that you can set to `False` if you want methods to return error codes instead of generating exceptions. Error events are generated by these controls if an error occurs during specific contexts of an operation. The contexts for which error events are generated are set in the `ErrorEventMask` property of the controls.

Exceptions

Exceptions are error messages returned directly to your programming environment. Usually, exceptions are processed by displaying a default error message. The error message allows you to end your application or to enter debug mode and perform certain debugging functions. Part of the exception returned is an error number and error description, displayed as part of the error message. For example the AI control might return the following exception to Visual Basic.

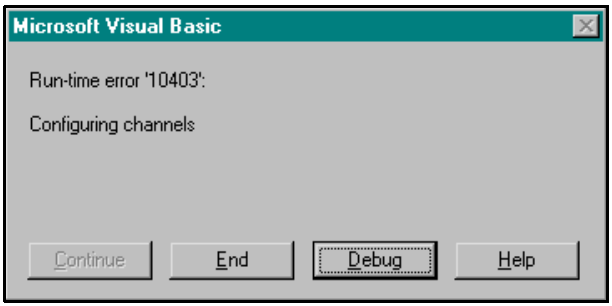

Figure 12-6. Visual Basic Error Messages

Depending on your programming environment, you might be able to insert code that can catch exceptions being sent to your application and handle them in another manner. In Visual Basic, you can do this by using the `On Error` statement.

- `On Error Resume Next` disables automatically generated error messages. The program continues running at the next line. To handle an error in this mode, you should check and process the information in the `Err` object in your code.

```
Private Sub Acquire_Click()
On Error Resume Next
    CWAI1.Configure
    If Err.Number <> 0 Then MsgBox "Configure: " +
      CStr(Err.Number)
    CWAI1.Start
    If Err.Number <> 0 Then MsgBox "Start: " +
      CStr(Err.Number)
End Sub
```

- `On Error GoTo` disables automatically generated error messages and causes program execution to continue at a specified location in the subroutine. You can define one error handler in your subroutine.

```
Private Sub Acquire_Click()
On Error GoTo ErrorHandler
    CWAI1.Configure
    CWAI1.Start
    Exit Sub
ErrorHandler:
    MsgBox "DAQ Error: " + CStr(Err.Number)
    Resume Next
End Sub
```

Return Codes

If the `ExceptionOnError` property is set to `False`, the DAQ and Instrumentation control methods return a status code to indicate whether an operation completed successfully. If the return value is something other than zero, it indicates a warning or error. A positive return value indicates a warning, signifying that a problem occurred in the operation, but that you should be able to continue with your application. A negative value indicates an error—a critical problem that has occurred in the operation—and that all other functions or methods dependent on the failed operation also will fail.

You can use the specific value of the return code for more detailed information about the error or warning. The ComponentWorks DAQ controls can convert the error code into a more descriptive text message, as described in the *GetErrorText Function* section of this chapter.

To retrieve the return code from a method call, assign the value of the function or method to a long integer variable and check the value of the variable after calling the function or method. For example, the following code checks the return code of the CWAI control `Start` method.

```
lerr = CWAI1.Start
```

```
If lerr <> 0 Then MsgBox "Error at DAQ Start: " +
CStr(lerr)
```

In Visual Basic, you can use the **MsgBox** popup window to display error information. Normally, you can write one error handler for your application instead of duplicating it for every call to a function or method. For example, the following code creates a `LogError` subroutine to use with the `Start` method and later functions or methods.

```
Private Sub LogError(code As Long)
   If code <> 0 Then
      MsgBox "DAQ Error: " + CStr(code)
   End If
End Sub
```

To use the `LogError` subroutine, call `LogError` before every function or method call. The return code is passed to `LogError` and processed.

```
LogError CWAI1.Start
```

Error and Warning Events

The DAQ and Instrumentation controls also include their own error and warning events—`DAQError` and `DAQWarning`. Although you normally use return codes for error checking of method calls, you cannot use return codes for error checking in asynchronous operations, such as a continuous analog input or asynchronous instrument control.

In this case, the controls generate their own error events if an error or warning occurs during an on-going process. You can develop an event handler to process these error and warning events. The following code shows the skeleton event functions for the CWAI control.

```
CWAI1_DAQError(ByVal StatusCode As Long, ByVal ContextID
  As Long, ByVal ContextDescription As String)
```

```
CWAI1_DAQWarning(ByVal StatusCode As Long, ByVal
  ContextID As Long, ByVal ContextDescription As String)
```

The `StatusCode` variable that is passed to the event handler contains the value of the error or warning condition. The `ContextID` contains a value describing the operation where the error or warning occurred, and the `ContextDescription` contains a string describing the operation where the error or warning occurred.

The following code, which produces the following error message box, shows an example of how you can use the `AI_DAQError` event in a Visual Basic application.

```
Private Sub CWAI1_DAQError(ByVal StatusCode As Long,
  ByVal ContextID As Long, ByVal ContextDescription As
  String)
   MsgBox ContextDescription + ": CStr(StatusCode)
End Sub
```

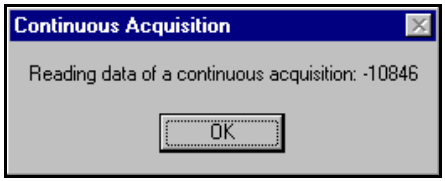

Figure 12-7. Error Message Box

By default, only asynchronous operations call error and warning events. You can set the `ErrorEventMask` property to specify the operations for which the error and warning events are generated.

GetErrorText Function

If you use return error codes to perform error checking, you might want to convert the error code values into more descriptive error texts. The ComponentWorks DAQ controls include a utility control that includes a method to convert error codes into descriptive error strings. To use this method, create a `DAQTools` control in your program and use the `GetErrorText` method as shown in the following example.

```
DAQError = CWAI1.Start
If DAQError <> 0 Then MsgBox
CWDAQTools.GetErrorText(DAQError)
```

The following screen shows a message box generated by using the `GetErrorText` function in the previous example.

Figure 12-8. Error Handling Message Box

Debugging

This section outlines a number of general debugging methods that you might use in your application development. If you experience some unexpected behavior in your program, use these methods to locate and correct the problem in your application.

Debug Print

One of the most common debugging methods is to print out or display important variables throughout the program execution. You can monitor critical values and determine when your program varies from the expected progress. Some programming environments have dedicated debugging windows that are used to display such information without disturbing the rest of the user interface. For example, you can use the `Debug.Print` command in Visual Basic to print information directly to the debug window.

```
Debug.Print CWAI1.Channels.Item(1).ChannelString
```

Breakpoint

Most development environments include breakpoint options so you can suspend program execution at a specific point in your code. Breakpoints are placed on a specific line of executable code in the program to pause program execution.

Stopping at a breakpoint confirms that your application ran to the line of code containing the breakpoint. If you are unsure whether a specific section of code is being called, place a breakpoint in the routine to find out. Once you have stopped at a specific section of your code, you can use other tools, such as a watch window or debug window, to analyze or even edit variables.

In some environments, breakpoints might also include conditions so program execution halts if certain conditions are met. These conditions usually check program variables for specific values. Once you have completed the work at the breakpoint, you can continue running your program, either in the normal run mode or in some type of single-step mode.

Watch Window

Use a watch window to display the value of a variable during program execution. You can use it to edit the value of a variable while the program is paused. In some cases, you can display expressions, which are values calculated dynamically from one or more program variables.

Single Step, Step Into, and Step Over

Use *single stepping* to execute a program one line at a time. This way, you can check variables and the output from your program during execution. Single stepping is commonly used after a breakpoint to slowly step though a questionable section of code.

If you use *step into*, the program executes any code available for subroutines or function calls and steps through it one line at a time. Use this mode if you want to check the code for each function called. The *step over* mode assumes that you do not want to go into the code for functions being called and runs them as one step.

In some cases, you might want to test a limited number of iterations of a loop but then run the rest of the iterations without stopping again. For this type of debugging, several environments include the *step to cursor* or *run to cursor* options. Under this option, you can place your cursor at a specific point in the code, such as the first line after a loop and run the program to that point.

A

Using Previous Versions of Visual Basic, Visual C++, and Delphi with ComponentWorks

This appendix outlines differences between the current and previous version of the programming environments with respect to using the ComponentWorks controls. This revision of the *Getting Results with ComponentWorks* manual was written with the most current environments available: Visual Basic 5, Visual C++ 5, and Delphi 3. In this appendix, the most current versions are compared with Visual Basic 4, Visual C++ 4.*x*, and Delphi 2.

Visual Basic 4

Differences between Visual Basic versions 4 and 5 include the design of the object browser, code completion in the code editor, and slightly different menu names. In addition to these environment differences, creating a default project to include ComponentWorks controls in version 5 is significantly different from creating one in version 4.

Menus and Commands

The Visual Basic 4 environment uses slightly different menus and command names for different options. Use the following commands for common operations with ComponentWorks.

- To open the controls toolbox, select **Toolbox** from the **View** menu.
- To add controls to the project toolbox, right click on the toolbox and select **Custom Controls...**.
- To add a DLL instrument driver to a project, select **References...** from the **Tools** menu.
- To open the object browser, select **Object Browser...** from the **View** menu or press <F2>.

- To open the Visual Basic properties page, select **Properties** from the **View** menu or press <F4>.
- To open custom property page for a control, right click on the control and select **Properties**.

Object Browser

Although its functionality is nearly the same, the Object Browser design is different between the two versions of Visual Basic. Compare Figure A-1, *Visual Basic 4 Object Browser*, and Figure A-2, *Visual Basic 5 Object Browser*, which both have the ComponentWorks Graph control selected.

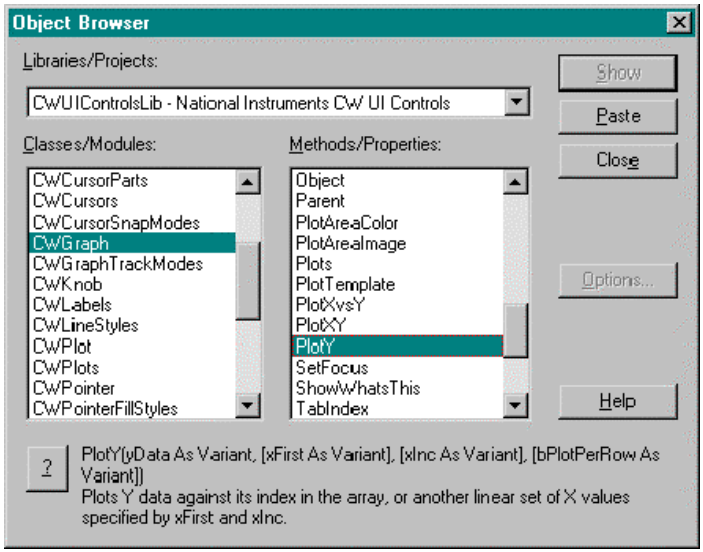

Figure A-1. Visual Basic 4 Object Browser

Notice that the Visual Basic 4 Object Browser does not list control events. The **Paste** button copies a selected item directly into the code editor; however, the button is enabled only when you open the Object Browser from the code editor.

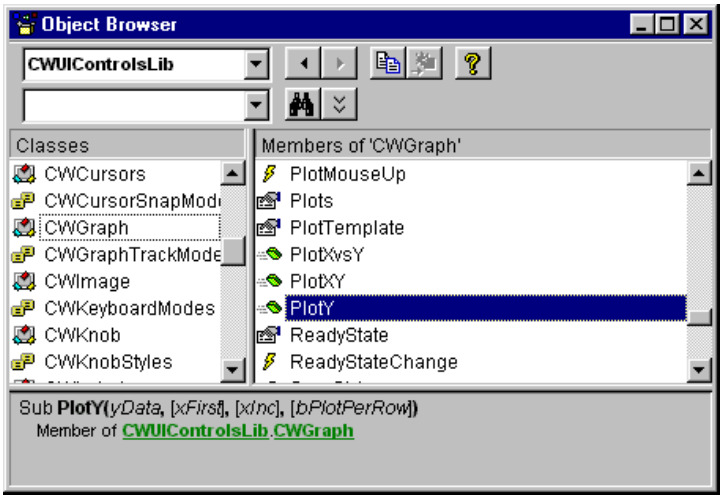

Figure A-2. Visual Basic 5 Object Browser

Code Completion

The Visual Basic 4 code editor does not have code completion. When entering your program code you must completely enter the syntax or use the Object Browser to paste selected properties and methods into your code.

Creating a Default ComponentWorks Project

Visual Basic 4 supports only one default project. The name of that project is `AUTO32LD.VBP`, and you can find it in the Visual Basic directory.

Use the following procedure to add the ComponentWorks controls to the default project or change the default controls.

1. Load the project `AUTO32LD.VBP` from the Visual Basic directory.
2. Add the ComponentWorks controls to the toolbox and make any other necessary changes.
3. Save the project in the same location.

When you create a new project, the ComponentWorks controls appear in the toolbox.

Visual C++ 4.*x*

Visual C++ 4.*x* and Visual C++ 5.0 require different methods to install the ActiveX control in the controls toolbar for use in the dialog editor. For other operations, such as creating a new workspace and building your user interface and code, follow the directions in Chapter 4, *Building ComponentWorks Applications with Visual C++*. Although some of the dialog windows in Visual C++ 4.*x* are slightly different from the manual, the descriptions still apply.

Creating Your Application

To create a new application in Visual C++ 4.*x*, use the MFC AppWizard as described in Chapter 4, *Building ComponentWorks Applications with Visual C++*. Initially, you see one extra dialog box, in which you should create a new **Project Workspace**. The next dialog allows you to select the **MFC AppWizard**.

In the second step of the MFC APPWizard, you must enable **OLE controls** support to use the ComponentWorks controls. This option is not automatically selected in Visual C++ 4.*x*. Complete the MFC AppWizard.

Adding ComponentWorks Controls to the Visual C++ Controls Toolbar

This section explains how you can add the ComponentWorks controls to the toolbar in Visual C++ 4.*x*.

Before you can use the ComponentWorks controls in your application, you must load the controls into the Controls toolbar in Visual C++, which is done from the Component Gallery in the Visual C++ environment. When you load the controls with the Component Gallery, you automatically generate a set of C++ wrapper classes in your project, which you need to work with the ComponentWorks controls.

The Controls toolbar is visible in the Visual C++ environment only if the Visual C++ dialog editor is currently active. To open the dialog editor, open the Project Workspace window (select **Project Workspace** from the **View** menu), select **ResourceView**, and double click on one of the **Dialog** entries.

1. To add a new control, select the **Component...** option from the **Insert** menu. This opens the Component Gallery in Visual C++, as shown in the following illustration.

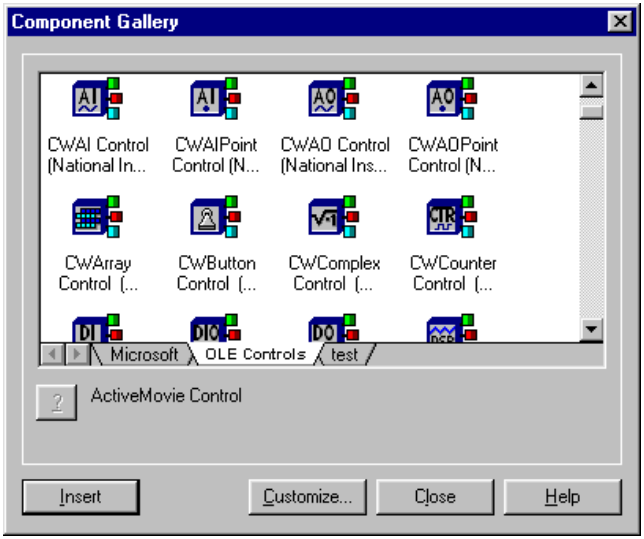

Figure A-3. Visual C++ Component Gallery

2. Select the **OLE Controls** tab in the gallery and look for the ComponentWorks controls. All ComponentWorks control names start with `CW`.
3. If the ComponentWorks controls are not shown in the OLE controls tab of the Component Gallery, push the **Customize...** button and the **Import** button in the following dialog. Select the OCX file on your hard drive that contains the controls you want to load. The ComponentWorks OCX files are located in the `\Windows\System(32)` directory and have names of the form `CW*.OCX`. Repeat the import process for any additional OCX files you want to add to the Component Gallery.
4. From the Component Gallery, select a control you want to add to the Controls toolbar and push the **Insert** button. The dialog window that appears lists the classes generated for the ActiveX control and the file names used.
5. Click **OK** to continue.

With this procedure, you have added the new classes to your project and the new control to the Controls toolbar. Repeat this process for additional controls you want to add.

When you have completed adding controls, click on **Close** in the Component Gallery. The new controls should be visible in the Visual C++ environment Controls toolbar.

Building Your User Interface and Code

Follow the directions in Chapter 4, *Building ComponentWorks Applications with Visual C++*, to build your user interface and use different VC++ tools, including the MFC Class Wizard, to build your code.

Delphi 2

Delphi 2 and Delphi 3 require different methods for loading ActiveX controls into the Component palette of the environment. For all other operations, such as creating the user interface and code, refer to Chapter 5, *Building ComponentWorks Applications with Delphi*.

☞ **Note** ***Remember that you must load the ComponentWorks controls into the Delphi environment before you can load and run any programs that use the ComponentWorks controls, including the examples installed with ComponentWorks.***

Loading the ComponentWorks Controls into the Component Palette

This section explains how you can add the ComponentWorks controls to the Component palette in Delphi 2.

Before you can use the ComponentWorks controls in your Delphi applications, you must add them to the Component palette in the Delphi environment. The controls need to be added only once to the palette because they will be available until they are explicitly removed from the Component palette. Adding the controls to the palette also creates a Pascal import unit (header file) that declares all the properties, methods, and events of a control. When you use a control on a form, a reference to the import unit is automatically added to the program.

☞ **Note** ***Before adding a new control to the Component palette, make sure to save all your work in Delphi, including files and projects. After loading the controls, Delphi closes any open projects and files to complete the loading process.***

1. To add ActiveX controls to the Component palette, select **Install** from the **Component** menu in the Delphi environment.
2. In the Install Components window, press the **OCX** button.
3. In the Import OLE Control window, select the desired registered control that appears on the Registered Controls field. The ComponentWorks controls all start with `National Instruments`.

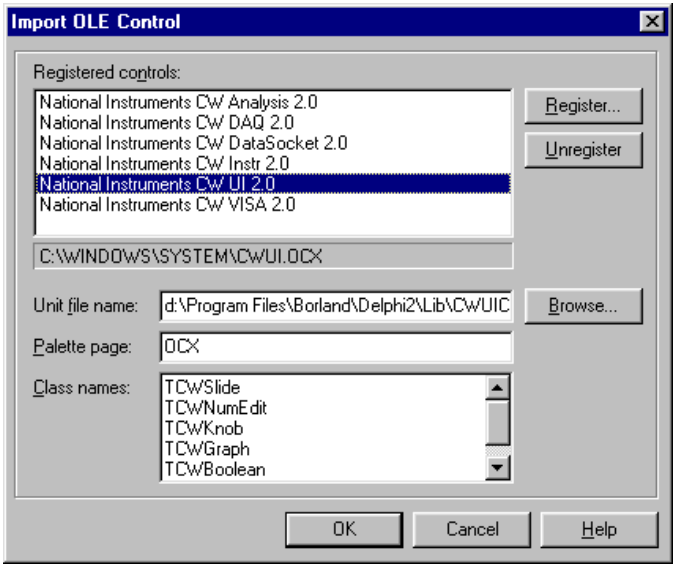

Figure A-4. Delphi Import OLE Control Dialog Box

4. After you have selected the proper control, click on **OK** to close the window. When you click the **OK** button, Delphi generates a Pascal import unit file for the selected OCX file, which is stored in the `\Lib` directory of Delphi. If you had previously installed the same `.OCX` file, it prompts you to overwrite the existing import unit file.

If your control does not appear in the Import OLE Control window it is not registered with the operating system. In this case, click on the **Register...** button to open the Register OLE Control window, find the OCX file that contains the control, and select it. This process registers the control with the operating system. Most OCX files are located in the `\System(32)` directory under Windows.

To load additional controls, return to the Import OLE Controls window and select more controls.

When you have finished selecting controls, click on **OK** in the Install Components window to load the new controls and add them to the **OCX** tab of the Component palette. Because this step closes any open projects in the Delphi environment, you need to reopen any projects you had open.

You can rearrange controls on the Component palette in Delphi by right clicking on the palette and selecting **Properties**.

B

Background Information about Data Acquisition

This appendix provides background information on data acquisition (DAQ) software and hardware specific to the ComponentWorks DAQ controls and describes the underlying architecture used by these controls.

Chapter 7, *Using the Data Acquisition Controls*, and the online reference provide information on the ComponentWorks DAQ controls. You can access the online reference from the Windows **Start** menu by selecting **Programs»National Instruments ComponentWorks» ComponentWorks Reference**.

Installation

Before you can use data acquisition hardware and software, you must completely install both the DAQ software and hardware and run the software configuration. To complete the installation, follow the instructions provided with your hardware and software. Usually, you first install the hardware. Verify that your computer is turned off and unplugged before opening it, and follow all other safety precautions. After installing the NI-DAQ software driver, restart your system. With most plug-in cards (AT and PCI cards), the operating system automatically detects that you have a new card installed.

If you plan to use ISA Plug-and-Play DAQ devices on Windows NT 4.0, you must install the Windows NT 4.0 ISA Plug-and-Play driver before configuring your device with the NI-DAQ Configuration Utility.

The Windows NT 4.0 ISA Plug-and-Play driver is not installed by default. Use the following steps to install the driver.

1. Insert your Windows NT 4.0 CD.
2. Access the `\Drvlib\Pnpisa\X86` directory.
3. Right click on the `Pnpisa.inf` file, select the **Install** option, and follow the on-screen instructions.

4. After you have installed the `Pnpisa.inf` file, shut down your computer.
5. If you have not already done so, install NI-DAQ and then shut down your computer.
6. Install your ISA Plug-and-Play DAQ device.
7. Turn on your computer. When Windows NT 4.0 detects your ISA Plug-and-Play DAQ device, it specifies the necessary driver files. Because this results in a configuration change, you need to restart your computer.
8. After you have restarted your computer, run the NI-DAQ Configuration Utility to configure your device.

Configuration

With the NI-DAQ Configuration Utility, you specify default settings for certain hardware options, such as input mode, input range, accessories, SCXI devices, and so on. Follow the directions provided with the NI-DAQ driver to configure your hardware.

Most data acquisition boards are automatically detected by the Windows operating system. If you have a board that Windows does not automatically detect, use the **Add New Hardware** option in the Windows Control Panel to manually inform Windows of the hardware.

You can use the DAQ Configuration Utility to test the operation of the device with its assigned resources (that is, the Input/Output range, Interrupt Request, and DMA channels). To run these tests, select a device in the NI-DAQ Devices list and click **Configure**. On the **System** property page, click the **Test Resources**. You also can perform simple input and output operations with your hardware by clicking **Run Test Panels...**.

SCXI

Configure any SCXI hardware on the **SCXI Devices** tab of the NI-DAQ Configuration Utility. Select **add new SCXI chassis and modules** and configure them according to your settings. Specify in the module configuration which module is connected to your DAQ. Save your changes.

Device Number

When configuring your hardware using the DAQ Configuration Utility, select a device number for each DAQ device. When you use the ComponentWorks AI, AIPoint, AO, and AOPoint controls, you specify the

device number as well as the channel number to use for the input and output operation. You can specify a Named Channel that you previously defined with the DAQ Channel Wizard rather than specifying the device number and channel number. Refer to the *Channel Wizard* section for more information about Named Channels.

Channel Wizard

You can use the DAQ Channel Wizard to create Named Channels. A Named Channel is a channel configuration that specifies a DAQ device; a hardware-specific channel string; channel attributes such as input limits, input mode, and actuator type; and a scaling formula for making a measurement or generating a signal in terms of your actual physical quantity. When you create a Named Channel, the DAQ Channel Wizard determines whether your DAQ device is capable of making the measurement.

You can start the Channel Wizard from the NI-DAQ Configuration Utility by selecting **File»Run Channel Wizard** or from the NI-DAQ program group in the Windows **Start** menu. Refer to the DAQ Channel Wizard help for instructions on creating Named Channels.

Programming

After completing the NI-DAQ configuration, you can use the ComponentWorks DAQ controls. To become familiar with the DAQ controls, read Chapter 7, *Using the Data Acquisition Controls*, and complete the tutorials. Use the ComponentWorks online reference to obtain complete information about the DAQ controls.

The following sections describe several concepts that you might encounter when working with the ComponentWorks DAQ controls and DAQ hardware.

Device Number and Channels

When you select and configure a DAQ control, first specify the device number and channel(s) you want to use with the control, or you might use named channels. See the *Channel Wizard* section for information about naming channels.

Use the `Channels` property of each control—Ports on the digital controls and Counter on the Counter and Pulse controls—to specify the particular channels you want to use with a specific control. A *channel* or *port* is a physical I/O location on a DAQ device. Most of the DAQ controls have a Channels or Ports collection to allow you to individually select and configure multiple channels or ports. These collections contain individual Channel and Port objects. For each Channel object, you can select one or more channels and configure additional properties to specify the operation of the hardware. For each Port object, you can select the direction (input or output) of the port.

Buffers

Buffers are memory locations used to store data, either acquired on a buffered input operation or generated on a buffered output operation. The individual control (AI, AO, DI, and DO) allocates the buffers according to the number of points it acquired. Normally, the buffer size is set equal to the number of points acquired or generated. In continuous or double buffered input operations, the buffer size is automatically set to a multiple of the number of points selected in the control. In continuous output operations, you explicitly select the buffer size.

When the buffer cannot store all the data as quickly as you can process it, you encounter an overflow condition. In such scenarios, you can increase the buffer size. The AI control allows you to disable the default buffer size and select your own value.

In single buffer operations, the buffer is written to or read from once. When the buffer is used up, the operation is completed. In continuous or double buffered operations, the same buffer is used multiple times. After using the buffer once, the operation returns to the beginning of the buffer and continues writing data to the buffer or reading data form the buffer. As the data acquisition operation writes data to the buffer or reads data from it, the application must retrieve data from the buffer or supply new data to the buffer. The application must complete this process quickly enough so unread input data is not overwritten or data is not generated twice on an output operation. On an output operation, you can allow regeneration of data.

Use the `AcquiredData` and `Progress` events with the buffers to indicate when a given number of points has been acquired or generated, which allows the application and developer to manage the data in the buffer more efficiently.

Clocks

Clocks are counter/timers on data acquisition devices that control the timing of operations such as a data acquisition or waveform generation. Some operations require multiple clocks.

The data acquisition driver programs the counter/timers that are used for clocks. Generally, the counter/timers cannot be controlled by the developer for other purposes. Configure the clocks for a particular DAQ process using the Clock objects in the ComponentWorks DAQ controls.

Typical Clock objects are the ScanClock and ChannelClock on the CWAI control. You can set properties such as a timebase source and frequency to determine the exact operation of the clocks. With this information, you can exactly configure the timing of a process to be controlled by the board with a preset frequency or to be synchronized with another onboard or external process. See the sections on *PFI* and *RTSI* for more information about routing timing signals.

Channel and Scan Clocks

You can acquire signals on multiple channels simultaneously. A multi-channel acquisition sometimes is referred to as a *scan*. Most data acquisition boards (for example, MIO E Series) have only one analog-to-digital converter with a multiplexor that connects only one of the analog input channels to the analog-to-digital converter at any given point in time. On these boards, the hardware cannot sample the channels in a multi-channel acquisition simultaneously. Rather, the hardware samples the channels in quick succession to approximate simultaneous sampling. The channel clock controls the interchannel daley between samplings of the channels in a multi-channel acquisition. By default, the CWAI control specifies that NI-DAQ automatically chooses the best possible interchannel delay, but you also can explicitly specify the interchannel delay using the `ChannelClock` property on the CWAI control.

The scan clock controls the timing between individual scans or between individual measurements in a one-channel acquisition. The scan rate is the actual acquisition rate per channel. You set the frequency of the scan clock in Hertz using the ScanClock object of the CWAI control. To synchronize the acquisition with another process, you can configure the ScanClock to take its input from another source.

SCXI

SCXI is a set of signal conditioning hardware used with plug-in data acquisition cards. Refer to the *Hardware SCXI* section for more information.

SCXI Channel String

When using SCXI modules with your ComponentWorks DAQ control, set a special SCXI channel string to indicate which modules and channels are used with a specific process. The device number set in the DAQ control corresponds to the DAQ device connected to and controlling the SCXI chassis, which could be an SCXI-1200 module in the chassis. The SCXI channel string specifies which chassis, modules, and channels are used by your DAQ process. For analog input operations, you can specify an onboard DAQ channel to acquire all analog signals from the SCXI chassis.

The SCXI channel string is of the format `"obA!scX!mdY!Z"` (for example, `"ob0!sc1!md1!0"`). `A`, `X`, `Y`, and `Z` represent specific values that do vary.

- `A` is the onboard analog DAQ channel used to acquire analog input signals, and `"obA!"` is used only in analog input operations. The onboard channel `A` is usually one less than the chassis number x.
- `X` represents the chassis number starting with 1 and allows you to specify additional chassis in multi-chassis configurations. If you use more than one SCXI chassis, all chassis are controlled from the same DAQ board by routing a DAQ cable from the first chassis to the second and so on. Optionally, you can use additional DAQ boards to control an individual chassis.
- `Y` indicates which SCXI module (actually module slot) in a chassis to use. Module slots are numbered starting with 1 for the left most slot.
- `Z` indicates which channels on an SCXI module to use.

For analog input operations, you can specify multiple channels on a module with the colon (`:`), but you can select only one continuous range of channels per module. For example, `"3:11"` specifies channels 3 through 11. For digital modules, all digital lines in a module are grouped in port 0 and you specify only `"0"` for your channel number.

For analog input operations, you can specify channels from multiple modules and chassis by using multiple Channel objects on your AI DAQ control.

Examples

`ob0!sc1!md3!0:4`	Analog Channels 0 through 4 on module 3 of chassis 1.
`ob1!sc2!md1!5`	Analog Channel 5 on module 1 of chassis 2.
`sc1!md3!0`	Digital port 0 on module 3 of chassis1.
`ob0!sc1!md2!4:11` `ob0!sc1!md3!8:15`	Analog Channel 4 through 11 on module 2 and Channels 8 through 15 on module 3 of chassis1.

Hardware

SCXI

SCXI is a set of signal conditioning hardware used with plug-in data acquisition cards. Signal conditioning prepares certain analog and digital signals for a plug-in DAQ card and provides functionality such as amplification, isolation, current supplies to sensors, multiplexing for higher channel counts, current to voltage conversion, and more.

An SCXI setup consists of an SCXI chassis and one or more SCXI modules selected for specific purposes and inserted in the SCXI chassis. Only one SCXI module per chassis is connected to the data acquisition board in your computer using a ribbon or shielded cable. This cable controls the chassis and all inserted modules from the DAQ board and passes signals from the chassis back to the computer. The modules in the chassis communicate with each other and route signals using the SCXI bus in the back of the chassis. All signal connections are made to the front of the individual chassis, which processes the signals and then passes them to the DAQ board using the SCXI bus and connected cable.

With analog input signals, the SCXI modules usually perform the signal conditioning and pass the conditioned signals to the DAQ card for analog-to-digital conversion. All analog input signals are multiplexed at the SCXI chassis on to one channel when transferred to the DAQ board. Analog output and digital I/O signals are directly processed on specific SCXI modules. Alternatively, you can use an SCXI-1200 module in the SCXI chassis to act as a DAQ device and perform the analog-to-digital conversion in the chassis. This module communicates with the computer using a digital connection such as a parallel cable.

Consult your SCXI hardware manual for more information about system configuration and settings. Remember that all SCXI hardware must be configured in the NI-DAQ configuration tool.

RTSI

The Real Time System Integration (RTSI) bus is an optional cable connection between multiple DAQ devices that allows you to share and exchange timing and trigger signals between devices. Use it to synchronize data acquisition processes running on multiple devices. For ISA (AT) and PCI bus cards, the RTSI connector is located along the top edge of the card and multiple cards can be connected to each other using an optional RTSI ribbon cable. PXI bus cards have the RTSI bus located on the PXI bus, and no additional cables are required.

The RTSI bus itself consists of seven signal lines (numbered 0 through 6) that can be freely defined using the DAQ software. Each DAQ device supporting RTSI contains a multiplexer that allows the card to route any available onboard signal, such as a timing or trigger signal, onto any of the seven available RTSI lines. Any board receiving a signal from the RTSI bus can route any of the seven signal lines back into one of its onboard signals. Although only one board at a time can supply a signal onto one of the RSTI lines, multiple boards can receive this signal into their onboard circuitry.

Programming

When using RTSI to share signals between devices, such as synchronizing two analog input operations, always call two operations to specify both the sender and receiver of the RTSI signal. To supply a signal to a RTSI line, use the `RouteSignal` (all E-series cards) or `RouteRTSI` (other cards) method of the CWDAQTools control. On these methods, specify both the onboard signal and RTSI line. To receive a signal from a RTSI line and use it in a DAQ process, specify the RTSI line in your regular DAQ control. The different Clock and Condition objects that control the timing and triggering of the different processes include options to select a RTSI line as the signal source. For E-series cards to use a RTSI line as the scan clock of an analog input acquisition, set the `ClockSourceType` to one of the RTSI options (low-to-high or high-to-low) and set the RTSI line number in the `ClockSourceSignal`.

PFI

Programmable Function Inputs (PFI) are variable input lines on the I/O connectors of specific DAQ devices (E-series cards) that route input signals to a variety of onboard signals. They also can be used to route output signals to any of the PFI lines on the I/O connector. PFI lines are used to output onboard timing and trigger signals for external purposes, including synchronization between multiple devices when the RTSI bus is not available. Most devices have 10 PFI lines, numbered 0 through 9. When used as an input, each PFI line is mapped to a specific onboard signal and can be used only for that purpose. On the output direction, you can freely assign onboard signals to a specific PFI line.

Use the `RouteSignal` method of the CWDAQTools control to route a specific onboard signal to a PFI line. To use a signal connected to a PFI line as an input signal, specify the PFI line in the corresponding Clock or Condition object of the DAQ control.

Consult the online reference for the `RouteSignal` method and Clock and Condition objects for more information about using the PFI and RTSI lines.

C

Common Questions

This appendix contains a list of answers to frequently asked questions. It contains general ComponentWorks questions as well as specific graphical user interface, data acquisition, instrument control, and analysis library questions.

Installation and Getting Started

How do I run the installer? What do I do if the AutoRun screen does not appear?

The ComponentWorks CD contains a different installer for each development system. The Full Development System includes the Instrument Driver Factory and Instrument Drivers installers. You can start all installers from the ComponentWorks CD AutoRun screen.

The AutoRun screen appears automatically when you load the ComponentWorks CD in your computer. You also can open the AutoRun screen by running the `SETUP` program in the root directory of the CD. To manually start an individual installer, run the `SETUP` program in the corresponding subdirectory of the CD.

Note ***You must run the SETUP program if you copied the installer to floppy disks for installation on a system without a CD-ROM drive.***

What does it mean when I place a ComponentWorks control on my form and get an error saying that I am not licensed to use this control?

This error indicates that ComponentWorks was not installed properly. Make sure to install ComponentWorks on your computer using your installation disks or CD. Copying the OCX files from another machine or sharing them over a network does not work.

Make sure to close all other applications before running the ComponentWorks installer and restart your machine after completing the installation. If you had previously installed a demo or evaluation version of ComponentWorks, uninstall this version first and restart your computer before installing the licensed version of ComponentWorks.

How do I distribute an application using ComponentWorks?

To distribute an application using ComponentWorks or any ActiveX controls, you need to distribute all OCX files, DLL files, and supporting OCXs and DLLs referenced in the application. You also need to distribute any support DLLs required by your specific programming environment.

Any OCXs and OLE Automation DLLs (OLE Automation Servers) distributed with an application need to be registered in the operating system on the target computer. Usually, you can do this with an installer, which you build with the Setup Wizard/Tool provided by your programming environment. If your setup tool does not provide this functionality or if your environment does not include a setup tool, you need to manually install all necessary files and register the OCX files using the `REGSVR32.EXE` utility provided by Microsoft.

To install and manually register an OCX file, copy the file to the `\System` (for Windows 95) or `\System32` (for Windows NT) subdirectory of the Windows directory on the target computer. Run the following:

```
regsvr32 c:\windows\system(32)\<ocxname>.ocx
```

To unregister a control, use the following:

```
regsvr32 /u c:\windows\system(32)\<ocxname>.ocx
```

If you distribute the ComponentWorks OCXs, you also need to make sure that all the necessary support DLLs are installed on the target machine. All the necessary support DLLs for the ComponentWorks controls are located in the `\ComponentWorks\Setup\redist` directory on the ComponentWorks CD.

Remember to include any files required by your programming environment, such as run-time DLLs. Check the documentation of your development environment for a list of required DLLs.

Visual Basic

I do not see any new controls in my Visual Basic toolbox. How do I load the ComponentWorks controls into Visual Basic?

To load the ComponentWorks controls in Visual Basic, right click on the toolbox and select **Components...** (**Custom Controls...** in Visual Basic 4) from the popup menu. Select the controls you want to use from the list of registered controls. If necessary, click on the **Browse** button to select a new unregistered control file. The ComponentWorks controls are located in the `\Windows\System` directory and start with `CW`. Select each of the controls and then click **OK** to return to Visual Basic. The new controls are placed in the toolbox.

How can I have the ComponentWorks controls and libraries automatically loaded when I start Visual Basic?

In Visual Basic 5, you can have the ComponentWorks controls and libraries loaded by adding them to a template project. To do this, create a new project, load the controls, and save the project with a descriptive name in the `(VB)\Template\Projects` directory. When creating new projects, you have the option of including the ComponentWorks controls.

In Visual Basic 4, load the `AUTO32LD` project located in the Visual Basic directory, add the ComponentWorks controls and save the project.

What is the difference in Visual Basic between using Base 0 or Base 1 to declare arrays?

Visual Basic can use either zero-based or one-based arrays. The default is Base 0. To change to the Base 1 option, use the following statement at the top of your code:

```
Option Base 1
```

You also can specify the exact range when declaring an array.

```
Dim voltBuffer(0 To 9) As Double
Dim voltBuffer(1 To 10) As Double
Dim voltBuffer(10 To 19) As Double
```

What manuals and additional information are available for ComponentWorks?

Refer to Chapter 2, *Getting Started with ComponentWorks*, for information sources, including the online reference and Web site.

User Interface Controls

How do I set the default value for a control such as the Knob or Slider?

To set the default value for a Knob, Slider, or Switch, open the default property page (<F4> in Visual Basic, <F11> in Delphi) and set the `Value` property.

In Visual C++, open the custom property pages and set the value. If the control is a ValuePairs Only control, set the `ValuePairIndex` property to the one-based index of the desired value pair.

How do I plot my data on the graph?

The easiest way to plot data is to use the `PlotY` method, which displays a simple plot of your values.

```
' dataArray can be an array or a variant containing an
array
```

```
CWGraph1.PlotY dataArray, 0, 1, True
```

Additional methods you can use to plot or chart data include `PlotXY`, `PlotXvsY`, `ChartY`, `ChartXY`, and `ChartXvsY`. The `Plot` methods plot a whole data set at once, deleting any previous information on the same plot. You can define multiple plots on a graph in the property pages and use the `Plot` methods to update individual plots.

The `Chart` methods append data to existing plots and create scrolling charts. Refer to the online reference for more information about these methods.

How do I display a value on a Slide, Knob, or NumEdit control and read values back from them? How do I read or set a button?

To pass a value to or read a value from one of these controls in Visual Basic and Borland Delphi, use their `Value` property. The `Value` property acts as a variable in your program, except that the value of this variable is the value of the control on the form.

```
' set the value of a slide to 5
CWSlide1.Value = 5
```

```
' read back the value from a knob
Dim ReadValue As Double
ReadValue = CWKnob1.Value
```

Buttons work in the same way, except that their values are Booleans.

```
' set a button
CWButton1.Value = True
' read a button
If CWButton1.Value = True then
' insert code here
End If
```

In Visual C++, control properties are not read or set directly (like variables). Instead, the wrapper class created for each control provide functions to read and write the value of that property. These functions are named with either `Get` or `Set` followed by the name of the property.

For example, to set the `Value` property of a slide, use the `SetValue` function. In the C code, the function call is preceded by the member variable name of the control to which it applies.

```
m_Slide.SetValue(5);
```

To read the value of a control, use the `GetValue` function. You can use the `GetValue` function to pass a value to another part of your program. For example, to pass the value of a slide to a meter use the following line of code.

```
m_Meter.SetValue(m_Slide.GetValue());
```

You can view the names of all the property functions in the ClassView of the Project Workspace in Visual C++. In the Project Workspace, select the ClassView and then the control/object to view its property functions (as well as its methods).

How can I change the style of my UI controls programmatically?

The Button, Knob, and Slide controls each have a number of default styles, which you can choose in the property pages of the control. In some applications, you might want to switch the style of a control while the program is running.

Use the `SetBuiltinStyle` method to change the style at run time to one of the predefined styles. The different styles are defined as constants in the UI controls.

```
CWSlide1.SetBuiltinStyle cwSlideStyleTank
```

```
CWKnob1.SetBuiltinStyle cwKnobStyleDial
```

```
CWButton1.SetBuiltinStyle cwButtonStyleRoundLED
```

You can use the `ExportStyle` and `ImportStyle` methods on all UI controls to save and load custom-defined styles. To save a predefined style, configure a control and click the Export Style (floppy disk icon) button in the property pages of the control (or right click on the control and select **Export Style**). Assign a file name for the new style. Using `ImportStyle`, you can interactively or programmatically load the settings from this file.

How do I access or change a particular axis, plot, cursor, or value pair on one of the UI controls?

Each of these objects, with the exception of the axis on the knob and slide, is contained within a collection object on the control. A collection object is a special object on a control that is used to store multiple objects of the same type. For example a graph can have many different axes. Rather than linking each Axis object directly to the graph, one Axes collection is linked to the graph and it contains all the axes.

Other objects are contained in collections—the Plot, Cursor, ValuePair, and Channel objects on the Data Acquisition controls. The name of the collection object is the name of the contained object in plural form. For example, the collection of Axis objects is Axes, Plot is Plots, Cursor is Cursors, and so on.

The ValuePair object is contained in the ValuePairs collection, which itself is part of the Axis object contained in the Knob or Slide control or in the Axes collection of the graph.

To access one of the objects in a collection, use the `Item` method of the collection object. The `Item` method extracts a particular object in the collection using a parameter, which is the one-based index of the object in the collection. For example, to access the first plot on a graph, use the following code.

```
CWGraph1.Plots.Item(1)
```

This code segment refers to the first plot on a graph as an object. You can then access the properties of the plot object by appending the name of the property. For example to read the X position of the second cursor on a graph, use the following code.

```
x = CWGraph1.Cursors.Item(2).XPosition
```

The properties of individual objects are described in the online reference. Search for the corresponding object name, such as CWAxis, CWPlot, CWCursor, and so on, to find the description. Each of the collection objects also has a number of properties and methods, described in the online reference under the collection name, such as CWAxes, CWPlots, and so on.

Use the `Count` property to determine the number of objects in a collection.

```
NumAxes = CWGraph1.Axes.Count
```

Use the `Add`, `Remove`, and `RemoveAll` methods to programmatically change the number of objects in a collection (for example, you can add and delete axes, cursors, value pairs, and so on in your program). The `Remove` method requires the index of the object you want to remove.

```
CWGraph1.Axes.Add
```

```
CWSlide1.Axis.ValuePairs.Remove 3
```

```
CWGraph1.Cursors.RemoveAll
```

Remember that ValuePair objects are contained in the ValuePairs collection, which itself is part of an Axis object. The following code shows you how to access value pairs.

```
CWKnob1.Axis.ValuePairs.Item(1).Name = "Maximum"
```

```
CWGraph1.Axes.Item(3).ValuePairs.Item(2).Value = 7.5
```

How do I add labels to the ComponentWorks graph axes?

Set the Caption property of the corresponding axis interactively on the **Ticks** tab of the Graph property pages.

How do I show gridlines on my graph without displaying the scales (similar to an oscilloscope screen)?

In the Ticks section of the Graph property pages, enable the grid lines for the selected axis and disable all the ticks and labels for the same axis. Make sure to keep the Visible property of the axis enabled.

Data Acquisition Controls

What methods and events do I need to use for the DAQ analog input control?

The DAQ AI control has four main methods—`Configure`, `Start`, `Stop`, `Reset`. The methods are listed and described in the ComponentWorks online reference.

After you set the properties of the AI control in the property pages, call the `Configure` method to pass the property values to the driver and hardware. Then call `Start` to start the actual acquisition. Use the `Stop` method to stop a continuous acquisition, and the `Reset` method to stop any acquisition and reset the driver and hardware. You need to call Configure anytime you change a property value programmatically, after you call the `Reset` method, or after the acquisition stops because of an error.

The AI control `AcquiredData` event, which is called when a set number of points is acquired, returns the data to the program. Two additional events, `DAQError` and `DAQWarning`, are generated in response to any errors or warnings that occur in the DAQ process. The `ErrorEventMask` property determines for which operations (contexts) of the AI control these events are called.

How do I assign new channels dynamically or retrieve channel information from the DAQ AI control in my program?

Use the Channels collection object of the AI control to retrieve information from individual Channel objects. Use the `CWAI.Channels.Item(n)` syntax to access individual Channel objects.

```
Dim ChannelInfo As String
```

```
ChannelInfo = CWAI1.Channels.Item(1).ChannelString
```

Use `CWAI.Channels.RemoveAll` and `CWAI.Channels.Add` with the appropriate information in the parameter list to delete all the Channel objects in the channel list (Channels collection) and to add new Channel objects to the collection.

```
CWAI.Channels.RemoveAll
```

```
CWAI.Channels.Add 1
```

Refer to the properties on `CWAI.Channels` in the online reference manual for more information.

Why does my EISA-A2000, AT-A2150, or AT-DSP2200 data acquisition card not work with ComponentWorks?

These three boards are not compatible with the ComponentWorks data acquisition controls.

How do I pass an array to `CWAI.AcquireData`? I keep getting type mismatches on my declared arrays. What is a Variant data type?

The two data buffers (`ScaledData` and `BinaryCodes`) of the `AcquireData` method are defined as variant data types. Do not declare them as arrays. Select two variable names, declare them as Variant, and pass them to `AcquireData`. The `AcquireData` method re-declares these variables to the correct array data type and array size.

Variant data types are used when the resultant data type is not known ahead of time. This allows a function or method to re-declare the variant variable to the appropriate type.

```
Dim Volts As Variant

Dim BinaryBuffer As Variant

Dim Timeout As Single

Timeout = 5

CWAI1.AcquireData (Volts, BinaryBuffer, Timeout)
```

When performing a single channel acquisition, data buffers are declared as one-dimensional arrays; a multi-channel acquisition are declared with two-dimensional arrays.

What is a named channel? What is a hardware-specific channel string?

A named channel is a symbolic name that represents a specific channel on a data acquisition device. For analog input channels, a named channel also includes information about the input range of the signal that your are measuring, the input mode of the channel, the coupling of the channel, and the scaling formula that the data acquisition driver (NI-DAQ) uses to convert the acquired voltages into engineering units. For analog output channels, a named channel also includes information specifying the actuator to use and a scaling formula that NI-DAQ uses to convert data specified in engineering units into the units of the actuator (for example, volts or milliAmps). You create named channels using the Channel Wizard feature of the NI-DAQ Configuration Utility.

Hardware-specific channel strings identify a channel on a DAQ device and contain no information about the device, scaling, or other attributes.

Examples of hardware-specific channel string refers are `"1"`, `"2:0"`, and `"ob0!sc1!md2!0:3"`.

What is the Channel Wizard?

The Channel Wizard is a tool that you use to define named channels. You launch the Channel Wizard from the NI-DAQ Configuration Utility, which is launched from a shortcut in the NI-DAQ program group (usually **Start»Programs»NI-DAQ for Windows»NI-DAQ Configuration Utility**).

Can I mix named channels with hardware-specific channel strings?

No. You can specify either named channels or hardware-specific channel strings within one Channels collection or channel string of your DAQ control, but not both.

Can I mix named channels that refer to different I/O types in the same channel string, such as analog and digital input?

No. Each ComponentWorks DAQ controls can work with only named channels that refer to the I/O type corresponding to the control.

I start my acquisition, but I get no data back and no errors. What is happening?

First, make sure that you set the `ExceptionOnError` property of the DAQ controls to `True`. If this property is set to `False`, you must check for non-zero return values from the methods that you call on the DAQ controls.

Second, make sure that you implement the `DAQError` and `DAQWarning` event handlers.

Third, if you are using an external clock or external triggers on the acquisition, make sure that you have wired valid signals to the data acquisition hardware.

Finally, if you are using all of the above mechanisms and you still do not receive any `AcquiredData` events and no errors, it is possible that the acquisition has stopped because of an overflow error or even an unexpected computer error. During an ongoing acquisition, the NI-DAQ driver only posts messages to the DAQ controls when the specified number of points has been acquired. If the acquisition has stopped because of an error,

NI-DAQ does not post a message and, as such, the DAQ control cannot fire any events notifying you of the error.

You can implement a watchdog timer using a Timer control that you reset on each `AcquiredData` event. If the acquisition unexpectedly stops sending events, the watchdog timer will fire an event, indicating that an error occurred.

Computer errors that can stop the acquisition include an electromagnetic pulse or power glitch that resets the data acquisition hardware without resetting the computer.

My mouse stops responding and the computer locks up after starting my acquisition?

Your acquisition rate is too fast. The NI-DAQ driver is posting messages at a rate too fast for your system. You can increase the number of scans between events, slow the acquisition rate, or turn off events and use a synchronous method for obtaining the data such as the `AcquireData` or `Read` method.

If you have a very fast acquisition and you want to read the most recent data acquired, turn off the `AcquiredData` and `Progress` event. Call the `Start` method and then call the `Read` method, passing in a `ReadSpec` variable with the `ReadMode` property set to `cwaiEndOfData`.

How do I program the counter/timer on my Lab-PC+, DAQCard-1200, or DAQCard-700 (or other 1200, 700, 500 series DAQ device)?

To program the counter/timer on these DAQ devices, use the ICtr functions in the DAQTools control. Consult the online reference for the descriptions of the `StartICtr`, `ReadICtr`, and `ResetICtr` functions. Do not use the CWCounter and CWPulse controls to program the counters on these devices.

GPIB, Serial, and VISA Controls

Why doesn't my device show up in the Detected Devices list?

First, check to make sure that the cables are secure. It is possible that a properly connected device might not show up as a listener if there are errors on the bus. Try changing the address of the device or resetting the device. With GPIB, check the results of the detection by using GPIB/NI-Spy, which comes with the National Instruments GPIB board.

Why can't I communicate with my device or instrument?

For GPIB, check the NI-488 global variables, `ibsta` and `iberr`, in the **Test** property page to see what the status is after the function. If an error is being reported, such as `ENOL` (Error No Listener), your device might not be at the correct GPIB address. Also, verify that the cables are secure. The NI-488 documentation discusses other troubleshooting methods for other errors.

If no error conditions exist but you still cannot communicate as you expect, you might not have sent the right command in the sequence that the device requires or you might not have sent the correct terminating characters. Terminating characters are typically semicolon, carriage return, line feed, or combinations of characters. The GPIB driver software never appends these to your data automatically. You must include these characters, if needed by your device, in the character string passed to the control method function. Check your device programming manual for the exact protocols required.

For serial, check the error status in the **Test** property page to see what the status is after the call that does not seem to be operating correctly. If an error is being reported such as `Bad Comm Port`, your device might not be at the correct serial port. Also, verify that the cables are secure.

If no error condition exists but you still cannot communicate as you expect, you might not have sent the right command in the sequence that the device requires or you may not have sent the correct terminating (EOS) characters. Terminating characters are typically semicolon, carriage return, line feed, or combinations of characters. The Serial control never appends these to your data automatically. You must include these characters, if needed by your device, in the character string passed to the device. You might need to adjust other settings on the Serial control also, such as flow control, parity,

baud rate, and so on. These settings must match those of the device. Check your device programming manual for the exact protocols required.

For VISA, follow the same method as described above, corresponding to your hard interface. Make sure you have the newest version of the NI-VISA driver, version 1.2 or newer, installed.

Why does the Test property page work with my device but my program does not?

The **Test** Property page in the Instrument controls lets users interactively type commands to be sent to the device and is an excellent place to start your development. It lets you test your commands before writing code. However, since the **Test** Property Page is interactive, you are naturally introducing very long delays between commands sent to your device. When you put the same commands in a program, the delays are gone. To solve the problem, put delays in your code where the program is failing. If this does not work, make sure you are checking for errors in the `OnError` event. This information can reveal details that will help troubleshoot your problem.

What does the Number Parser do and how do I use it?

The Number Parser is a parsing pattern that takes all of the numbers in a string and returns them as an array of doubles, ignoring all other information in the string.

Where can I learn more about how to use the parsing tools included in the Serial, GPIB, and VISA controls?

There are several parsing examples provided in the `\ComponentWorks\samples\<programming language>\Instrument Drivers` directory, in addition to the online reference.

Analysis Controls

How do I call an analysis function?

All ComponentWorks analysis functions are methods of the different Analysis controls. To use a specific function, place the corresponding control in your program. Then call the function as a method of the control in your program, passing any necessary parameters and assigning the return value to a variable if appropriate. For example, calling the `Mean` function on the CWStat control has the following syntax in Visual Basic.

```
MeanVal = CWStat1.Mean(data)
```

Consult the online reference manual for complete documentation of each analysis function.

My analysis functions seem to have no effect. What does the return code –30008 mean?

The analysis functions are licensed in three different levels of the analysis library corresponding to the three different levels of ComponentWorks packages. If you attempt to call a function that is not in your level of the analysis library, you get an error –30008.

See Chapter 10, *Using the Analysis Controls and Functions*, for a list of the analysis functions and their corresponding license level.

D

Error Codes

This appendix lists the error codes returned by the ComponentWorks DAQ controls, ComponentWorks VISA control, and Analysis Library functions. It also lists some general ComponentWorks error codes.

Table D-1 lists the negative error codes returned by the DAQ controls. Each DAQ control returns an error code that indicates whether it executed successfully. When a control returns a code that is negative, the control did not execute.

Table D-1. Data Acquisition Control Error Codes

Error Code	Description
–10001	An error was detected in the input string; the arrangement or ordering of the characters in the string is not consistent with the expected ordering.
–10002	An error was detected in the input string; the syntax of the string is correct, but certain values specified in the string are inconsistent with other values specified in the string.
–10003	The value of a numeric parameter is invalid.
–10004	The value of a numeric parameter is inconsistent with another one, and therefore the combination is invalid.
–10005	The device is invalid.
–10006	The line is invalid.
–10007	A channel, port, or counter is out of range for the device type or device configuration; or the combination of channels is not allowed; or the scan order must be reversed (0 last).
–10008	The group is invalid.
–10009	The counter is invalid.
–10010	The count is too small or too large for the specified counter, or the given I/O transfer count is not appropriate for the current buffer or channel configuration.

Table D-1. Data Acquisition Control Error Codes (Continued)

Error Code	Description
–10011	The analog input scan rate is too fast for the number of channels and the channel clock rate; or the given clock rate is not supported by the associated counter channel or I/O channel.
–10012	The analog input or analog output voltage range is invalid for the specified channel, or you are writing an invalid voltage to the analog output.
–10013	The driver returned an unrecognized or unlisted error code.
–10014	The group size is too large for the board.
–10015	The time limit is invalid.
–10016	The read count is invalid.
–10017	The read mode is invalid.
–10018	The offset is unreachable.
–10019	The frequency is invalid.
–10020	The timebase is invalid.
–10021	The limits are beyond the range of the board.
–10022	Your data array contains an incomplete update, or you are trying to write past the end of the internal buffer, or your output operation is continuous and the length of your array is not a multiple of one half of the internal buffer size.
–10023	The write mode is out of range or is disallowed.
–10024	Adding the write offset to the write mark places the write mark outside the internal buffer.
–10025	The requested input limits exceed the board's capability or configuration. Alternative limits were selected.
–10026	The requested number of buffers or the buffer size is not allowed. For example, the buffer limit for Lab and 1200 devices is 64K samples, or the board does not support multiple buffers.
–10027	For DAQEvents 0 and 1 general value A must be greater than 0 and less than the internal buffer size. If DMA is used for DAQEvent 1, general value A must divide the internal buffer size evenly, with no remainder. If the TIO-10 is used for DAQEvent 4, general value A must be 1 or 2.

Table D-1. Data Acquisition Control Error Codes (Continued)

Error Code	Description
–10028	The cutoff frequency specified is not valid for this device.
–10029	The function you are calling is no longer supported in this version of the driver.
–10030	The specified baud rate for communicating with the serial port is not valid on this platform.
–10031	The specified SCXI chassis does not correspond to a configured SCXI chassis.
–10032	The SCXI module slot that was specified is invalid or corresponds to an empty slot.
–10033	The window handle passed to the function is invalid.
–10034	No configured message matches the one you tried to delete.
–10035	The specified attribute is not relevant.
–10036	The specified year is invalid.
–10037	The specified month is invalid.
–10038	The specified day is invalid.
–10039	The specified input string is too long. For instance, DAQScope 5102 devices can only store a string up to 32 bytes in length on the calibration EEPROM. In that case, please shorten the string.
–10080	The gain is invalid.
–10081	The pretrigger sample count is invalid.
–10082	The posttrigger sample count is invalid.
–10083	The trigger mode is invalid.
–10084	The trigger count is invalid.
–10085	The trigger range or trigger hysteresis window is invalid.
–10086	The external reference is invalid.
–10087	The trigger type is invalid.
–10088	The trigger level is invalid.

Table D-1. Data Acquisition Control Error Codes (Continued)

Error Code	Description
–10089	The total count is inconsistent with the buffer size and pretrigger scan count or with the board type.
–10090	The individual range, polarity, and gain settings are valid but the combination is not allowed.
–10091	You have attempted to use an invalid setting for the iterations parameter. The iterations value must be 0 or greater. Your device might be limited to only two values, 0 and 1.
–10092	Some devices require a time gap between the last sample in a scan and the start of the next scan. The scan interval you have specified does not provide a large enough gap for the board. See your documentation for an explanation.
–10093	FIFO mode waveform generation cannot be used because at least one condition is not satisfied.
–10094	The calDAC constant passed to the function is invalid.
–10095	The calibration stimulus passed to the function is invalid.
–10100	The requested digital port width is not a multiple of the hardware port width or is not attainable by the DAQ hardware.
–10120	Invalid application used.
–10121	Invalid counterNumber used.
–10122	Invalid paramValue used.
–10123	Invalid paramID used.
–10124	Invalid entityID used.
–10125	Invalid action used.
–10200	Unable to read data from EEPROM.
–10201	Unable to write data to EEPROM.
–10202	You cannot write into this location or area of your EEPROM because it is write-protected. You may be trying to store calibration constants into a write-protected area; if this is the case, you should select user area of the EEPROM instead.
–10240	The driver interface could not locate or open the driver.

Table D-1. Data Acquisition Control Error Codes (Continued)

Error Code	Description
–10241	One of the driver files or the configuration utility is out of date, or a particular feature of the Channel Wizard is not supported in this version of the driver.
–10242	The specified function is not located in the driver.
–10243	The driver could not locate or open the configuration file, or the format of the configuration file is not compatible with the currently installed driver.
–10244	The driver encountered a hardware-initialization error while attempting to configure the specified device.
–10245	The driver encountered an operating system error while attempting to perform an operation, or the operating system does not support an operation performed by the driver.
–10246	The driver is unable to communicate with the specified external device.
–10247	The CMOS configuration-memory for the device is empty or invalid, or the configuration specified does not agree with the current configuration of the device, or the EISA system configuration is invalid.
–10248	The base addresses for two or more devices are the same; consequently, the driver is unable to access the specified device.
–10249	The interrupt configuration is incorrect given the capabilities of the computer or device.
–10250	The interrupt levels for two or more devices are the same.
–10251	The DMA configuration is incorrect given the capabilities of the computer/DMA controller or device.
–10252	The DMA channels for two or more devices are the same.
–10253	Unable to find one or more jumperless boards you have configured using the NI-DAQ Configuration Utility.
–10254	Cannot configure the DAQCard because 1) the correct version of the card and socket services software is not installed; 2) the card in the PCMCIA socket is not a DAQCard; or 3) the base address and/or interrupt level requested are not available according to the card and socket services resource manager. Try different settings or use AutoAssign in the NI-DAQ Configuration Utility.

Table D-1. Data Acquisition Control Error Codes (Continued)

Error Code	Description
–10255	There was an error in initializing the driver for Remote SCXI.
–10256	There was an error in opening the specified COM port.
–10257	Bad base address specified in the configuration utility.
–10258	Bad DMA channel 1 specified in the configuration utility or by the operating system.
–10259	Bad DMA channel 2 specified in the configuration utility or by the operating system.
–10260	Bad DMA channel 3 specified in the configuration utility or by the operating system.
–10261	The user mode code failed when calling the kernel mode code.
–10340	No RTSI signal/line is connected, or the specified signal and the specified line are not connected.
–10341	The RTSI signal/line cannot be connected as specified.
–10342	The specified RTSI signal is already being driven by a RTSI line, or the specified RTSI line is already being driven by a RTSI signal.
–10343	The specified SCXI configuration parameters are invalid, or the function cannot be executed with the current SCXI configuration.
–10344	The Remote SCXI unit is not synchronized with the host. Reset the chassis again to resynchronize it with the host.
–10345	The required amount of memory cannot be allocated on the Remote SCXI unit for the specified operation.
–10346	The packet received by the Remote SCXI unit is invalid. Check your serial port cable connections.
–10347	There was an error in sending a packet to the remote chassis. Check your serial port cable connections.
–10348	The Remote SCXI unit is in reprogramming mode and is waiting for reprogramming commands from the host (NI-DAQ Configuration Utility).
–10349	The module ID read from the SCXI module conflicts with the configured module type.

Table D-1. Data Acquisition Control Error Codes (Continued)

Error Code	Description
–10360	The DSP driver was unable to load the kernel for its operating system.
–10370	The scan list is invalid; for example, you are mixing AMUX-64T channels and onboard channels, scanning SCXI channels out of order, or have specified a different starting channel for the same SCXI module. Also, the driver attempts to achieve complicated gain distributions over SCXI channels on the same module by manipulating the scan list and returns this error if it fails.
–10400	The specified resource is owned by the user and cannot be accessed or modified by the driver.
–10401	The specified device is not a National Instruments product, the driver does not support the device (for example, the driver was released before the device was supported), or the device has not been configured using the NI-DAQ Configuration Utility.
–10402	No device is located in the specified slot or at the specified address.
–10403	The specified device does not support the requested action (the driver recognizes the device, but the action is inappropriate for the device).
–10404	No line is available.
–10405	No channel is available.
–10406	No group is available.
–10407	The specified line is in use.
–10408	The specified channel is in use.
–10409	The specified group is in use.
–10410	A related line, channel, or group is in use; if the driver configures the specified line, channel, or group, the configuration, data, or handshaking lines for the related line, channel, or group will be disturbed.
–10411	The specified counter is in use.
–10412	No group is assigned, or the specified line or channel cannot be assigned to a group.
–10413	A group is already assigned, or the specified line or channel is already assigned to a group.

Table D-1. Data Acquisition Control Error Codes (Continued)

Error Code	Description
–10414	The selected signal requires a pin that is reserved and configured only by NI-DAQ. You cannot configure this pin yourself.
–10415	This function does not support your DAQ device when an external multiplexer (such as an AMUX-64T or SCXI) is connected to it.
–10440	The specified resource is owned by the driver and cannot be accessed or modified by the user.
–10441	No memory is configured to support the current data transfer mode, or the configured memory does not support the current data transfer mode. (If block transfers are in use, the memory must be capable of performing block transfers.)
–10442	The specified memory is disabled or is unavailable given the current addressing mode.
–10443	The transfer buffer is not aligned properly for the current data transfer mode. For example, the buffer is at an odd address, is not aligned to a 32-bit boundary, is not aligned to a 512-bit boundary, and so on. Alternatively, the driver is unable to align the buffer because the buffer is too small.
–10444	No more system memory is available on the heap, or no more memory is available on the device, or insufficient disk space is available.
–10445	The transfer buffer cannot be locked into physical memory. On PC AT machines, portions of the DMA data acquisition buffer may be in an invalid DMA region, for example, above 16 megabytes.
–10446	The transfer buffer contains a page break; system resources may require reprogramming when the page break is encountered.
–10447	The operating environment is unable to grant a page lock.
–10448	The driver is unable to continue parsing a string input due to stack limitations.
–10449	A cache-related error occurred, or caching is not supported in the current mode.
–10450	A hardware error occurred in physical memory, or no memory is located at the specified address.

Table D-1. Data Acquisition Control Error Codes (Continued)

Error Code	Description
–10451	The driver is unable to make the transfer buffer contiguous in virtual memory and therefore cannot lock it into physical memory; thus, the buffer cannot be used for DMA transfers.
–10452	No interrupt level is available for use.
–10453	The specified interrupt level is already in use by another device.
–10454	No DMA controller is available in the system.
–10455	No DMA channel is available for use.
–10456	The specified DMA channel is already in use by another device.
–10457	DMA cannot be configured for the specified group because it is too small, too large, or misaligned. Consult the device user manual to determine group ramifications with respect to DMA.
–10458	The storage disk you specified is full.
–10459	The NI-DAQ DLL could not be called due to an interface error.
–10460	You have mixed VIs from the DAQ library and the _DAQ compatibility library (LabVIEW 2.2 style VIs). You may switch between the two libraries only by running the DAQ VI Device Reset before calling _DAQ compatibility VIs or by running the compatibility VI Board Reset before calling DAQ VIs.
–10461	The specified resource is unavailable because it has already been reserved by another entity.
–10462	The specified resource has not been reserved, so the action is not allowed.
–10480	The scan list is too large to fit into the mux-gain memory of the board.
–10481	You must provide a single buffer of interleaved data, and the channels must be in ascending order. You cannot use DMA to transfer data from two buffers; however, you may be able to use interrupts.
–10540	At least one of the SCXI modules specified is not supported for the operation.
–10541	CTRB1 will drive COUTB1, however CTRB1 will also drive TRIG1. This may cause unpredictable results when scanning the chassis.
–10560	The DSP handle input is not valid.

Table D-1. Data Acquisition Control Error Codes (Continued)

Error Code	Description
–10561	Either DAQ or WFM can use a PC memory buffer, but not both at the same time.
–10600	No setup operation has been performed for the specified resources. Or, some resources require a specific ordering of calls for proper setup.
–10601	The specified resources have already been configured by a setup operation.
–10602	No output data has been written into the transfer buffer.
–10603	The output data associated with a group must be for a single channel or must be for consecutive channels.
–10604	Once data generation has started, only the transfer buffers originally written to may be updated. If DMA is active and a single transfer buffer contains interleaved channel data, new data must be provided for all output channels currently using the DMA channel.
–10605	No data was written to the transfer buffer because the final data block has already been loaded.
–10606	The specified resource is not armed.
–10607	The specified resource is already armed.
–10608	No transfer is in progress for the specified resource.
–10609	A transfer is already in progress for the specified resource, or the operation is not allowed because the device is in the process of performing transfers, possibly with different resources.
–10610	A single output channel in a group may not be paused if the output data for the group is interleaved.
–10611	Some of the lines in the specified channel are not configured for the transfer direction specified. For a write transfer, some lines are configured for input. For a read transfer, some lines are configured for output.
–10612	The specified line does not support the specified transfer direction.
–10613	The specified channel does not support the specified transfer direction.
–10614	The specified group does not support the specified transfer direction.
–10615	The clock configuration for the clock master is invalid.

Table D-1. Data Acquisition Control Error Codes (Continued)

Error Code	Description
–10616	The clock configuration for the clock slave is invalid.
–10617	No source signal has been assigned to the clock resource.
–10618	The specified source signal cannot be assigned to the clock resource.
–10619	A source signal has already been assigned to the clock resource.
–10620	No trigger signal has been assigned to the trigger resource.
–10621	The specified trigger signal cannot be assigned to the trigger resource.
–10622	The pretrigger mode is not supported or is not available in the current configuration, or no pretrigger source has been assigned.
–10623	No posttrigger source has been assigned.
–10624	The delayed trigger mode is not supported or is not available in the current configuration, or no delay source has been assigned.
–10625	The trigger configuration for the trigger master is invalid.
–10626	The trigger configuration for the trigger slave is invalid.
–10627	No signal has been assigned to the trigger resource.
–10628	A signal has already been assigned to the trigger resource.
–10629	The specified operating mode is invalid, or the resources have not been configured for the specified operating mode.
–10630	The parameters specified to read data were invalid in the context of the acquisition. For example, an attempt was made to read 0 bytes from the transfer buffer, or an attempt was made to read past the end of the transfer buffer.
–10631	Continuous input or output transfers are not allowed in the current operating mode, or continuous operation is not allowed for this type of device.
–10632	Certain inputs were ignored because they are not relevant in the current operating mode.
–10633	The specified analog output regeneration mode is not allowed for this board.
–10634	No continuous (double buffered) transfer is in progress for the specified resource.

Table D-1. Data Acquisition Control Error Codes (Continued)

Error Code	Description
–10635	Either the SCXI operating mode specified in a configuration call is invalid, or a module is in the wrong operating mode to execute the function call.
–10636	You cannot start a continuous (double-buffered) operation with a synchronous function call.
–10637	Attempted to configure a buffer after the buffer had already been configured. You can configure a buffer only once.
–10680	All channels of this board must have the same gain.
–10681	All channels of this board must have the same range.
–10682	All channels of this board must be the same polarity.
–10683	All channels of this board must have the same coupling.
–10684	All channels of this board must have the same input mode.
–10685	The clock rate exceeds the board's recommended maximum rate.
–10686	A configuration change has invalidated the scan list.
–10687	A configuration change has invalidated the acquisition buffer, or an acquisition buffer has not been configured.
–10688	The number of total scans and pretrigger scans implies that a triggered start is intended, but triggering is not enabled.
–10689	Digital trigger B is illegal for the number of total scans and pretrigger scans specified.
–10690	This board does not allow digital triggers A and B to be enabled at the same time.
–10691	This board does not allow an external sample clock with an external scan clock, start trigger, or stop trigger.
–10692	The acquisition cannot be started because the channel clock is disabled.
–10693	You cannot use an external scan clock when doing a single scan of a single channel.
–10694	The sample frequency exceeds the safe maximum rate for the hardware, gains, and filters used.

Table D-1. Data Acquisition Control Error Codes (Continued)

Error Code	Description
–10695	You have set up an operation that requires the use of interrupts. DMA is not allowed. For example, some DAQ events, such as messaging and LabVIEW occurrences, require interrupts.
–10696	Multi-rate scanning cannot be used with the AMUX-64, SCXI, or pretriggered acquisitions.
–10697	Unable to convert your timebase/interval pair to match the actual hardware capabilities of this board.
–10698	You cannot use this combination of scan and sample clock timebases for this board.
–10699	You cannot use this combination of scan and sample clock source polarities for this operation and board.
–10700	You cannot use this combination of scan and convert clock signal sources for this operation and board.
–10701	The call had no effect because the specified channel had not been set for later internal update.
–10702	Pretriggering and posttriggering cannot be used simultaneously on the Lab and 1200 series devices.
–10710	The specified port has not been configured for handshaking.
–10720	The specified counter is not configured for event-counting operation.
–10740	A signal has already been assigned to the SCXI track-and-hold trigger line, or a control call was inappropriate because the specified module is not configured for one-channel operation.
–10780	When you have an SC2040 attached to your device, all analog input channels must be configured for differential input mode.
–10781	The polarity of the output channel cannot be bipolar when outputting currents.
–10782	The specified operation cannot be performed with the SC-2040 configured in hold mode.
–10783	Calibration constants in the load area have a different polarity from the current configuration. Therefore, you should load constants from factory.
–10800	The operation could not complete within the time limit.

Table D-1. Data Acquisition Control Error Codes (Continued)

Error Code	Description
–10801	An error occurred during the calibration process. Possible reasons for this error include incorrect connection of the stimulus signal, incorrect value of the stimulus signal, or malfunction of your DAQ device.
–10802	The requested amount of data has not yet been acquired.
–10803	The ongoing transfer has been stopped. This is to prevent regeneration for output operations, or to reallocate resources for input operations.
–10804	The transfer stopped prior to reaching the end of the transfer buffer.
–10805	The clock rate is faster than the hardware can support. An attempt to input or output a new data point was made before the hardware could finish processing the previous data point. This condition may also occur when glitches are present on an external clock signal.
–10806	No trigger value was found in the input transfer buffer.
–10807	The trigger occurred before sufficient pretrigger data was acquired.
–10808	An error occurred in the parallel port communication with the DAQ device.
–10809	Attempted to start a pulse width measurement with the pulse in the phase to be measured (e.g., high phase for high-level gating).
–10810	An unexpected error occurred inside the driver when performing this given operation.
–10840	The contents or the location of the driver file was changed between accesses to the driver.
–10841	The firmware does not support the specified operation, or the firmware operation could not complete due to a data integrity problem.
–10842	The hardware is not responding to the specified operation, or the response from the hardware is not consistent with the functionality of the hardware.
–10843	Because of system limitations, the driver could not write data to the device fast enough to keep up with the device throughput. This error may be returned erroneously when an overRunErr has occurred.
–10844	New data was not written to the output transfer buffer before the driver attempted to transfer the data to the device.

Table D-1. Data Acquisition Control Error Codes (Continued)

Error Code	Description
–10845	Because of system limitations, the driver could not read data from the device fast enough to keep up with the device throughput; the onboard device memory reported an overflow error.
–10846	The driver wrote new data into the input transfer buffer before the previously acquired data was read.
–10847	New buffer information was not available at the time of the DMA chaining interrupt; DMA transfers will terminate at the end of the currently active transfer buffer.
–10848	The driver could not obtain a valid reading from the transfer-count register in the DMA controller.
–10849	The configuration file or DSP kernel file could not be opened.
–10850	Unable to close a file.
–10851	Unable to seek within a file.
–10852	Unable to read from a file.
–10853	Unable to write to a file.
–10854	An error occurred accessing a file.
–10855	NI-DAQ does not support the current operation on this particular version of the operating system.
–10856	An unexpected error occurred from the operating system while performing the given operation.
–10857	An unexpected error occurred inside the kernel of the device while performing this operation.
–10858	The system has reconfigured the device and has invalidated the existing configuration. The device requires reinitialization to be used again.
–10880	A change to the update rate is not possible at this time because 1) when waveform generation is in progress, you cannot change the interval timebase or 2) when you make several changes in a row, you must give each change enough time to take effect before requesting further changes.
–10881	You cannot do another transfer after a successful partial transfer.

Table D-1. Data Acquisition Control Error Codes (Continued)

Error Code	Description
–10882	The data collected on the Remote SCXI unit was overwritten before it could be transferred to the buffer in the host. Try using a slower data acquisition rate if possible.
–10883	New data could not be transferred to the waveform buffer of the Remote SCXI unit to keep up with the waveform update rate. Try using a slower waveform update rate if possible.
–10884	Could not rearrange data after a pretrigger acquisition completed.
–10920	One or more data points may have been lost during buffered GPCTR operations due to speed limitations of your system.
–10940	No response was received from the Remote SCXI unit within the specified time limit.
–10941	Reprogramming the Remote SCXI unit was unsuccessful. Please try again.
–10942	An invalid reset signature was sent from the host to the Remote SCXI unit.
–10943	The interrupt service routine on the remote SCXI unit is taking longer than necessary. You do not need to reset your remote SCXI unit, however, clear and restart your data acquisition.

Table D-2 lists the error codes returned from the VISA control.

Table D-2. VISA Control Error Codes

Error Code	Description
–30500	Unknown system error (miscellaneous error).
–30514	The given session or object reference is invalid.
–30515	Specified type of lock cannot be obtained, or specified operation cannot be performed, because the resource is locked.
–30516	Invalid expression specified for search.
–30517	Insufficient location information or the device or resource is not present in the system.
–30518	Invalid resource reference specified. Parsing error.
–30519	Invalid access mode.

Table D-2. VISA Control Error Codes (Continued)

Error Code	Description
–30521	Timeout expired before operation completed.
–30522	Unable to deallocate the previously allocated data structures corresponding to this session or object reference.
–30527	Specified degree is invalid.
–30528	Specified job identifier is invalid.
–30529	The specified attribute is not defined or supported by the referenced resource.
–30530	The specified state of the attribute is not valid, or is not supported as defined by the resource.
–30531	The specified attribute is read only.
–30532	The specified type of lock is not supported by this resource.
–30533	The access key to the specified resource is invalid.
–30538	Specified event type is not supported by the resource.
–30539	Invalid mechanism specified.
–30540	A handler was not installed.
–30541	The given handler reference is either invalid or was not installed.
–30542	Specified event context is invalid.
–30548	User abort occurred during transfer.
–30552	Violation of raw write protocol occurred during transfer.
–30553	Violation of raw read protocol occurred during transfer.
–30554	Device reported an output protocol error during transfer.
–30555	Device reported an input protocol error during transfer.
–30556	Bus error occurred during transfer.
–30558	Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state).
–30559	Unable to queue the asynchronous operation.
–30560	Insufficient system resources to perform necessary memory allocation.

Table D-2. VISA Control Error Codes (Continued)

Error Code	Description
–30561	Invalid buffer mask specified.
–30562	Could not perform operation because of I/O error.
–30563	A format specifier in the format string is invalid.
–30565	A format specifier in the format string is not supported.
–30566	The specified trigger line is currently in use.
–30574	Service request has not been received for the session.
–30578	Invalid address space specified.
–30581	Invalid offset specified.
–30584	Specified offset is not accessible from this hardware.
–30587	The specified session is not currently mapped.
–30595	No listeners condition is detected (both NRFD and NDAC are de-asserted).
–30596	The interface associated with this session is not currently the controller in charge.
–30603	The given session or object reference does not support this operation.
–30606	A parity error occurred during transfer.
–30607	A framing error occurred during transfer.
–30608	An overrun error occurred during transfer. A character was not read from the hardware before the next character arrived.
–30618	Specified width is not supported by this hardware.
–30620	The value of some parameter (which parameter is not known) is invalid.
–30621	The protocol specified is invalid.
–30623	Invalid size of window specified.
–30628	The specified session currently contains a mapped window.
–30629	The given operation is not implemented.
–30631	Invalid length specified.
–30656	The current session did not have a lock on the resource.

Table D-2. VISA Control Error Codes (Continued)

Error Code	Description
–30582	Invalid access width specified.
–30585	Cannot support source and destination widths that are different.
–30612	The specified offset is not properly aligned for the access width of the operation.
–30613	A specified user buffer is not valid or cannot be accessed for the required size.
–30614	The resource is valid, but VISA cannot currently access it.
–30547	You must be enabled for events of the specified type in order to receive them.
–30658	A code library required by VISA could not be located or loaded.
–30589	A previous response is still pending, causing a multiple query error.
–30597	The interface associated with this session is not the system controller.
–30645	Invalid mode specified.

If an error occurs during a call to any of the functions in the ComponentWorks Analysis controls, the exception contains the error code. This code is a value that specifies the type of error that occurred. You can find the currently defined error codes and their associated meanings in Table D-3.

Table D-3. Analysis Error Codes

Error Code	Description
–20001	There is not enough memory to perform the specified routine.
–20002	The input sequences must be the same size.
–20003	The number of samples must be >0.
–20004	The number of samples must be >= 0.
–20006	The number of samples must be >= 2.
–20007	The number of samples must be >= 3.
–20008	The input arrays do not contain the correct number of data values for this function.
–20009	The size of the input array must be a power of two: *size* = 2^*m*, 0<*m*<23.

Table D-3. Analysis Error Codes (Continued)

Error Code	Description
–20010	The maximum allowable transform size has been exceeded.
–20011	The duty cycle must be equal to or fall between 0 and 100: 0 <= *duty cycle* <= 100.
–20012	The number of cycles must be > 0 and <= the number of samples.
–20013	The width must meet: 0 < *width* < *samples*.
–20014	The following conditions must be met: 0 <= (*delay* + *width*) < *samples*.
–20015	dt must be >= 0.
–20016	dt must be > 0.
–20017	The following condition must be met: 0 <= *index* < *samples*.
–20018	The following condition must be met: 0 <= (*index* + *length*) < *samples*.
–20019	The upper value must be >= the lower value.
–20020	The cut-off frequency, *fc*, must meet: 0 <= *fc* <= *fs*/2.
–20021	The order must be > 0.
–20022	The decimating factor must meet: 0 < *decimating factor* <= *samples*.
–20023	The following conditions must be met: 0 < *f_low* <= *f_high* <= *fs*/2.
–20024	The ripple amplitude must be > 0.
–20025	The attenuation must be > 0.
–20026	The width must be > 0.
–20027	The final value must be > 0.
–20028	The attenuation value must be greater than the ripple amplitude.
–20029	The step-size, u, must meet: 0 <= *u* <= 0.1.
–20030	The leakage coefficient, leak, and step-size parameter, u, must meet: 0 <= *leak* <= *u*.
–20031	The filter cannot be designed with the specified input values.
–20032	The rank of the filter must meet: 1 <= (2**rank* + 1) <= *size*.
–20033	The number of coefficients must be odd for this filter.

Table D-3. Analysis Error Codes (Continued)

Error Code	Description
–20034	The number of coefficients must be even for this filter.
–20035	The standard deviation must be greater than zero for normalization.
–20036	The elements of the Y Values array must be nonzero and either all positive or all negative.
–20037	The number of data points in the Y Values array must be greater than the order.
–20038	The number of intervals must be > 0.
–20039	The number of columns in the first matrix is not equal to the number of rows in the second matrix or vector.
–20040	The input matrix must be a square matrix.
–20041	The system of equations cannot be solved because the input matrix is singular.
–20042	The number of levels is out of range.
–20043	The level of factors is outside the allowable range of some data.
–20044	Zero observations were made at some level of a factor.
–20045	The total number of data points must be equal to the product of *levels/each factor * observations/cell*.
–20046	There is an overflow in the calculated F-value for the ANOVA Fit function.
–20047	The data is unbalanced. All cells must contain the same number of observations.
–20048	The Random Effect model was requested when the Fixed Effect model is required.
–20049	The x-values must be distinct.
–20050	The interpolating function has a pole at the requested value.
–20051	All values in the first column of X matrix must be one.
–20052	The degrees of freedom must be one or more.
–20053	The probability must be between zero and one.
–20054	The probability must be greater than or equal to zero and less than one.

Table D-3. Analysis Error Codes (Continued)

Error Code	Description
–20055	The number of categories or samples must be greater than one.
–20056	The contingency table has a negative number.
–20057	The parameter to the beta function should be $0 < p < 1$.
–20058	Invalid number of dimensions or dependent variables.
–20059	Negative number error.
–20060	Divide by zero error.
–20061	The selection is invalid.
–20062	The maximum number of iterations was exceeded.
–20063	The coefficients of the polynomial are invalid.
–20064	The internal memory state of this function was not initialized correctly.
–20065	The elements in the vector can not be all zero.
–20066	The information in IIR filter structure is not correct.
–20080	Time increment must be greater than the (window length)/16.
–20081	dN must be greater than zero.
–20082	Time increment must not be greater than dM.
–20083	Window length must be > 4 and a power of 2.
–20084	Time increment must not be greater than (window length)/4.
–20085	The size of the input array and its Hilbert transform must be equal.
–20086	Window length must be > 2 and a power of 2.
–20101	Parameter must meet the condition: *Top*>*Base*.
–20102	The shifts must meet: $\|shifts\| < samples$.
–20103	The order must be positive.
–20999	Serious algorithm failure. Call National Instruments support.

Table D-4 lists the negative error codes that can be returned by any ComponentWorks operation.

Table D-4. General ComponentWorks Error Codes

Error Code	Description
–30000	An unexpected error has occurred.
–30001	Too many controls are configured for this DAQ device. Reset one of the other controls configured for this device before configuring any more controls for this device.
–30002	You have passed an invalid value for one of the parameters to the function, method or property.
–30003	You have passed an invalid type into a parameter of a VARIANT type.
–30004	A divide by zero error has occurred.
–30005	The result of a calculation is an imaginary number.
–30006	An overflow error has occurred.
–30007	Out of memory.
–30008	You have called a function or method requiring a ComponentWorks product that you do not have a license for. For example, you may be using a method that is not supported in the base, (or standard) Analysis package. To upgrade your product, contact National Instruments.
–30009	Cannot reconfigure when PulseType has changed.
–30010	DAQ operation cannot be started because it is not configured or is currently active.
–30011	The type of counter/timer on the device cannot perform the requested operation.
–30013	Newer version of NI-DAQ required.
–30014	Invalid ReturnDataType property. Scaled data will be returned.
–30015	CWAI.ProgressInterval must divide evenly into CWAI.NScans.
–30016	CWAI.NScans must divide evenly into CWAI.NScansPerBuffer.
–30017	ProgressInterval must divide evenly into NPatterns.
–30018	Invalid trigger source.
–30019	Current PulseType cannot be reconfigured.

Table D-4. General ComponentWorks Error Codes (Continued)

Error Code	Description
–30020	DAQ operation not active.
–30021	DAQ operation not configured.
–30100	The data type is not correct.
–30101	Array cannot be locked.
–30102	Array cannot be unlocked.
–30103	The argument is not an array.
–30104	The input parameters must all contain the same data type.
–30301	Operation Timed out.
–30302	No more data.
–30303	Could not parse character.
–30304	IO Operation in Progress.
–30305	Cannot Add Token to non User-Defined Pattern.
–30307	Cannot load GPIB DLL.
–30308	Cannot load VISA DLL.
–30312	Invalid COM Port.
–30315	Access denied: Specified resource is already in use.
–30316	Specified Device is Not Configured.
–30317	Specified Device is Not Opened.
–30322	File Not Found.
–30324	Cannot open file: Too Many Open Files.
–30325	Cannot Open File: Access Denied.
–30330	Invalid GPIB Board Number.
–30331	The GPIB Board is not the Controller In Charge (CIC).
–30332	No Listeners on the GPIB are addressed.
–30333	GPIB Board is not properly addressing itself.
–30334	Invalid Parameter.

Table D-4. General ComponentWorks Error Codes (Continued)

Error Code	Description
–30335	The GPIB Board is not the System Controller.
–30336	I/O operation timed out.
–30337	Invalid GPIB IO Address.
–30338	GPIB DMA Error.
–30340	I/O In Progress.
–30341	GPIB Board or device does not have the ability to perform the requested operation.
–30342	File System Error.
–30344	Error on the GPIB Bus.
–30345	Serial Poll Overflow: Too many status bytes received.
–30346	The SRQ line is stuck asserting.
–30350	Problem with GPIB Listening Table.
–30400	The value or attribute cannot be set because the CWData object is read only.
–30401	The data read is corrupt or in an unrecognized format.
–30402	You are initializing too many cluster elements.
–30403	You are trying to create an array with too many dimensions.
–30404	You are using an element index that is out of range.
–30405	You cannot index the current value because it is not a vector or cluster.
–30420	The URL prefix (http:, ftp:, file:, and so on) is not recognized.
–30421	The URL syntax could not be recognized.

E

Distribution and Redistributable Files

This chapter contains information about ComponentWorks 2.0 redistributable files and distributing applications that use ComponentWorks controls.

Files

The files in the `\Setup\redist` directory of the ComponentWorks CD are necessary for distributing applications and programs that use ComponentWorks controls. You need to distribute only those files needed by the controls you are using in your application.

Distribution

When installing an application using ComponentWorks controls on another computer, you also must install the necessary control files and supporting libraries on the target machine. In addition to installing all necessary OCX files on a target computer, you must register each of these files with the operating system. This allows your application to find the correct OCX file and create the controls.

If your application performs any I/O operations requiring separate driver software, such as data acquisition or GPIB, you must install and configure the driver software and corresponding hardware on the target computer. For more information, consult the hardware documentation for the specific driver used.

When distributing applications with the ComponentWorks controls, do not violate the license agreement (section 5) provided with the software. If you have any questions about the licensing conditions, contact National Instruments.

Automatic Installers

Many programming environments include some form of a setup or distribution kit tool. This tool automatically creates an installer for your application so that you can easily install it on another computer. To function successfully, this tool must recognize which control files and supporting libraries are required by your application and include these in the installer it creates. The resulting installer also must register the controls on the target machine.

Some of these tools, such as the Visual Basic 5 Setup Wizard, use dependency files to determine which libraries are required by an OCX file. Each of the ComponentWorks OCX files includes a corresponding dependency file located in the `\Windows\System` directory (`\Windows\System32` for WindowsNT) after you install the ComponentWorks software.

Some setup tools might not automatically recognize which files are required by an application but provide an option to add additional files to the installer. In this case, verify that all necessary OCX files (corresponding to the controls used in your application) as well as all the DLL and TLB files from the `\redist` directory are included. You also should verify that the resulting installer does not copy older versions of a file over a newer version on the target machine.

If your programming environment does not provide a tool or wizard for building an installer, you may use third-party tools, such as InstallShield. Some programming environments provide simplified or trial versions of third-party installer creation tools on their installation CDs.

Manual Installation

If your programming environment does not include a setup or distribution kit tool, you must build your own installer and perform the installation task manually. To install your application on another computer, follow these steps:

1. Copy the application executable to the target machine.
2. Copy all required ComponentWorks OCX files (corresponding to the controls used in your application) to the System directory (`\Windows\System` for Windows 95 or `\Windows\System32` for WindowsNT) on the target machine.

3. Copy all DLL and TLB files in the `\redist` directory to the System directory on the target machine.
4. Copy any other DLLs and support files required by your application to the System directory on the target machine.

Some of these files might already be installed on the target machine. If the file on the target machine has an earlier version number than the file in the `\redist` directory, copy the newer file to the target machine.

After copying the files to the target machine, you must register all OCX files with the operating system. To register an OCX file, you need a utility such as `REGSVR32.EXE`. You must copy this utility to the target machine to register the OCX files, but you can delete it after completing the installation. Use this utility to register each OCX file with the operating system, as in the following example.

```
regsvr32 c:\windows\system\cwui.ocx
```

DataSocket Server

If your application uses the DataSocket Server, you also might need to distribute the server with your application. The server executable can be easily copied to the target machine and does not need to be registered. It requires the same support libraries as the ComponentWorks controls. With the DataSocket server, you also should distribute the server configuration file (`CWDSSINI.DSS`) to start the server on the target in the same state as configured on your development machine. You can distribute multiple configuration files to switch between different setups of the server.

In rare cases, you might want to distribute the DataSocket Server Manager utility with your application. In general, this should not be necessary. Contact National Instruments technical support for information about distributing the DataSocket Server Manager.

Instrument Drivers

If you are using any instrument drivers in your applications, you must distribute the instrument driver DLL file with your application. Copy the instrument driver DLL and necessary support DLL (`INSTRSUP.DLL`) to the target machine. Make sure to install and configure any required driver software, such as NI-488.2, on the target machine.

ComponentWorks Evaluation

Once the ComponentWorks OCX files are installed and registered on a target computer, your application can create the controls as necessary. You or your customer also can use the same OCX files in any compatible development environment as an evaluation version of the controls. If desired, you may distribute the ComponentWorks reference files (from the `\redist` directory) with your application, which provide complete documentation of the ComponentWorks controls when used in evaluation mode.

If you would like to use the ComponentWorks controls as a development tool on this target machine, you must purchase another ComponentWorks development system. Contact National Instruments to purchase additional copies of the ComponentWorks software.

Run-Time Licenses

For each copy of your ComponentWorks-based application that you distribute, you must have a valid run-time license. A limited number of run-time licenses are provided with the ComponentWorks development systems. National Instruments driver software also provides you with ComponentWorks run-time licenses. You can purchase additional ComponentWorks run-time licenses from National Instruments. Consult the license agreement (section 5) provided with the software for more detailed information. If you have any questions about the licensing conditions, contact National Instruments.

Troubleshooting

Try the following suggestions if you encounter problems after installing your application on another computer.

The application is not able to find an OCX file or is not able to create a control.

- The control file or one of its supporting libraries is not copied on the computer. Verify that the correct OCX files and all their supporting libraries are copied on the machine. If one control was built using another, you might need multiple OCX files for one control.
- The control is not properly registered on the computer. Make sure you run the registration utility and that it registers the control.

Controls in the application run in evaluation (demo) mode.

- The application does not contain the correct run-time license. When developing your application, verify that the controls are running in a fully licensed mode. Although most programming environments include a run-time license for the controls in the executable, some do not.

  If you are developing an application in Visual C++ using SDI (single document interface) or MDI (multiple document interface), you must include the run-time license in the program code for each control you create. Consult the ComponentWorks documentation, National Instruments Knowledgebase (`www.natinst.com/support`) or technical support if you are not familiar with this operation.

F

Customer Communication

For your convenience, this appendix contains forms to help you gather the information necessary to help us solve your technical problems and a form you can use to comment on the product documentation. When you contact us, we need the information on the Technical Support Form and the configuration form, if your manual contains one, about your system configuration to answer your questions as quickly as possible.

National Instruments has technical assistance through electronic, fax, and telephone systems to quickly provide the information you need. Our electronic services include a bulletin board service, an FTP site, a fax-on-demand system, and e-mail support. If you have a hardware or software problem, first try the electronic support systems. If the information available on these systems does not answer your questions, we offer fax and telephone support through our technical support centers, which are staffed by applications engineers.

Electronic Services

Bulletin Board Support

National Instruments has BBS and FTP sites dedicated for 24-hour support with a collection of files and documents to answer most common customer questions. From these sites, you can also download the latest instrument drivers, updates, and example programs. For recorded instructions on how to use the bulletin board and FTP services and for BBS automated information, call 512 795 6990. You can access these services at:

United States: 512 794 5422
Up to 14,400 baud, 8 data bits, 1 stop bit, no parity

United Kingdom: 01635 551422
Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

France: 01 48 65 15 59
Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

FTP Support

To access our FTP site, log on to our Internet host, `ftp.natinst.com`, as `anonymous` and use your Internet address, such as `joesmith@anywhere.com`, as your password. The support files and documents are located in the `/support` directories.

Fax-on-Demand Support

Fax-on-Demand is a 24-hour information retrieval system containing a library of documents on a wide range of technical information. You can access Fax-on-Demand from a touch-tone telephone at 512 418 1111.

E-Mail Support (Currently USA Only)

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

`support@natinst.com`

Telephone and Fax Support

National Instruments has branch offices all over the world. Use the list below to find the technical support number for your country. If there is no National Instruments office in your country, contact the source from which you purchased your software to obtain support.

Country	Telephone	Fax
Australia	03 9879 5166	03 9879 6277
Austria	0662 45 79 90 0	0662 45 79 90 19
Belgium	02 757 00 20	02 757 03 11
Brazil	011 288 3336	011 288 8528
Canada (Ontario)	905 785 0085	905 785 0086
Canada (Quebec)	514 694 8521	514 694 4399
Denmark	45 76 26 00	45 76 26 02
Finland	09 725 725 11	09 725 725 55
France	01 48 14 24 24	01 48 14 24 14
Germany	089 741 31 30	089 714 60 35
Hong Kong	2645 3186	2686 8505
Israel	03 6120092	03 6120095
Italy	02 413091	02 41309215
Japan	03 5472 2970	03 5472 2977
Korea	02 596 7456	02 596 7455
Mexico	5 520 2635	5 520 3282
Netherlands	0348 433466	0348 430673
Norway	32 84 84 00	32 84 86 00
Singapore	2265886	2265887
Spain	91 640 0085	91 640 0533
Sweden	08 730 49 70	08 730 43 70
Switzerland	056 200 51 51	056 200 51 55
Taiwan	02 377 1200	02 737 4644
United Kingdom	01635 523545	01635 523154
United States	512 795 8248	512 794 5678

Technical Support Form

Photocopy this form and update it each time you make changes to your software or hardware, and use the completed copy of this form as a reference for your current configuration. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

If you are using any National Instruments hardware or software products related to this problem, include the configuration forms from their user manuals. Include additional pages if necessary.

Name _____

Company _____

Address _____

_____

Fax (___) _____Phone (___) _____

Computer brand_____ Model _____Processor_____

Operating system (include version number) _____

Clock speed _____MHz RAM _____MB Display adapter _____

Mouse ___yes ___no Other adapters installed_____

Hard disk capacity _____MB Brand_____

Instruments used _____

_____

National Instruments hardware product model _____ Revision _____

Configuration _____

National Instruments software product _____ Version _____

Configuration _____

The problem is: _____

_____

_____

_____

_____

List any error messages: _____

_____

_____

The following steps reproduce the problem: _____

_____

_____

_____

_____

ComponentWorks Hardware and Software Configuration Form

Record the settings and revisions of your hardware and software on the line to the right of each item. Complete a new copy of this form each time you revise your software or hardware configuration, and use this form as a reference for your current configuration. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

National Instruments Products

DAQ hardware _____

Interrupt level of hardware _____

DMA channels of hardware _____

Base I/O address of hardware _____

Programming choice _____

ComponentWorks and NI-DAQ version _____

Other boards in system _____

Base I/O address of other boards _____

DMA channels of other boards _____

Interrupt level of other boards _____

Other Products

Computer make and model _____

Microprocessor _____

Clock frequency or speed _____

Type of video board installed _____

Operating system version _____

Operating system mode _____

Programming language _____

Programming language version _____

Other boards in system _____

Base I/O address of other boards _____

DMA channels of other boards _____

Interrupt level of other boards _____

Documentation Comment Form

National Instruments encourages you to comment on the documentation supplied with our products. This information helps us provide quality products to meet your needs.

Title: *Getting Results with ComponentWorks™*

Edition Date: April 1998

Part Number: 321170C-01

Please comment on the completeness, clarity, and organization of the manual.

_____

_____

_____

_____

_____

_____

_____

If you find errors in the manual, please record the page numbers and describe the errors.

_____

_____

_____

_____

_____

_____

_____

Thank you for your help.

Name _____

Title _____

Company _____

Address _____

_____

E-Mail Address _____

Phone (___) _____ Fax (___) _____

Mail to: Technical Publications
National Instruments Corporation
6504 Bridge Point Parkway
Austin, Texas 78730-5039

Fax to: Technical Publications
National Instruments Corporation
512 794 5678

Glossary

Prefix	Meanings	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}
G-	giga-	10^{9}
t-	tera-	10^{12}

Numbers/Symbols

12-bit — Resolution of a data acquisition device. A 12-bit device converts an analog voltage into a 12-bit binary integer. The binary value is scaled to a voltage representation in software.

16-bit — Resolution of a data acquisition device. A 16-bit device converts an analog voltage into a 16-bit binary integer. The binary value is scaled to a voltage representation in software.

1D — One-dimensional.

2D — Two-dimensional.

9513 Counter — *See* AM9513 Counter.

A

A — Amperes.

A/D — Analog-to-Digital.

AC — Alternating Current.

AC signal	Signal with significant frequency components.
ActiveX	Set of Microsoft technologies for reusable software components. Formerly called *OLE*.
ActiveX control	Standard software tool that adds additional functionality to any compatible ActiveX container. The DAQ, UI, and analysis tools in ComponentWorks are all ActiveX controls. An ActiveX control has properties, methods, objects, and events.
ADC	Analog-to-Digital Converter. An electronic device, often an integrated circuit, that converts an analog voltage to a digital number.
ADC resolution	Resolution of the ADC, which is measured in bits. An ADC with 16 bits has a higher resolution than a 12-bit ADC.
AI	Analog Input.
AIGND	Analog Input GrouND pin on a DAQ device.
AIPoint	Analog Input Single Point control.
AM 9513 Counter	Counter/Timer chip used on legacy MIO and other data acquisition devices, replaced by the DAQ-STC chip on newer devices, including the E-series DAQ boards. The AM9513 Counter does not support some of the advanced features of the DAQ-STC.
amplification	Type of signal conditioning that improves accuracy in the resulting digitized signal and reduces noise.
AMUX devices	*See* analog multiplexer.
analog multiplexer	Device that increase the number of measurement channels while still using a single instrumentation amplifier. Also called *AMUX devices*.
analog trigger	Trigger that occurs at a user-selected point on an incoming analog signal. Triggering can be set to occur at a specified level either on an increasing or a decreasing signal (positive or negative slope). Analog triggering can be implemented either in software or in hardware. When implemented in software, all data is collected, transferred into system memory, and analyzed for the trigger condition. When analog triggering is implemented in hardware, no data is transferred to system memory until the trigger condition has occurred.
ANOVA	ANalysis Of VAriance.

AO	Analog Output.
AOPoint	Analog Output single Point control.
array	Ordered, indexed set of data elements of the same type.
asynchronous	Property of a function or operation that begins an operation and returns control to the program prior to the completion or termination of the operation.
ATC	Analog Trigger Circuit; an analog trigger contained on some E-series DAQ devices, required for any analog triggering (including ETS operations).

B

BCD	Binary-Coded Decimal.
bipolar	Signal range that includes both positive and negative values (for example, –5 to 5 V).
buffer	Temporary storage for acquired or generated data.

C

callback (function)	User-defined function called in response to an event from an object. Also called an *event handler.*
cascading	Process of extending the counting range of a counter chip by connecting to the next higher counter.
channel	Pin or wire lead to which you apply or from which you read the analog or digital signal. Analog signals can be single-ended or differential. For digital signals, group channels to form ports. Ports usually consist of either four or eight digital channels.
channel clock	Clock controlling the time interval between individual channel sampling within a scan. Boards with simultaneous sampling do not have this clock.
channel list	Collection of channel objects that specify the channels used by a control.
Channel Wizard	Utility that guides you through naming and configuring your DAQ analog and digital channels.

Chart History	CWGraph property for charting that determines how many points the graph stores before deleting old data.
Chart Style	CWGraph property that specifies how a chart method updates the display as new data is plotted.
clock	Hardware component that controls timing for reading from or writing to groups; an input pin on a counter/timer.
cm	Centimeters.
code width	Smallest detectable change in an input voltage of a DAQ device.
Collection	Control property and object that contains a number of objects of the same type, such as pointers, axes, and plots. The type name of the collection is the plural of the type name of the object in the collection. For example, a collection of CWAxis objects is called CWAxes. To reference an object in the collection, you must specify the object as part of the collection, usually by index. For example, `CWGraph.Axes.Item(2)` is the second axis in the CWAxes collection of a graph.
column-major order	Systematic way to organize the data in a 2D array by columns.
common-mode voltage	Any voltage present at the instrumentation amplifier inputs with respect to amplifier ground.
condition object	Object used to specify the Start, Pause, or Stop condition on a data acquisition process.
Control Refresh	CWGraph property that determines when changes to the graph are displayed. The property name is `ImmediateUpdates`.
conversion device	Device that transforms a signal from one form to another. For example, analog-to-digital converters (ADCs) for analog input, digital-to-analog converters (DACs) for analog output, digital input or output ports, and counter/timers are conversion devices.
counter/timer group	Collection of counter/timer channels. You can use this type of group for simultaneous operation of multiple counter/timers.
coupling	Manner in which a signal is connected from one location to another.

D

D/A	Digital-to-Analog.
DAC	Digital-to-Analog Converter. An electronic device, often an integrated circuit, that converts a digital number into a corresponding analog voltage or current.
DAQ	Data acquisition.
DAQ-STC	DAQ System Timing Controller. An ASIC developed by National Instruments for enhanced timing control on data acquisition devices. DAQ-STC is used on E-Series and other National Instruments DAQ devices and is required for certain counter/timer operations, including buffered counter measurements and frequency shift keying.
data acquisition	Process of acquiring data, typically from A/D or digital input plug-in boards.
DataSocket	Technology that simplifies data exchange between an application and other applications, files, FTP servers, and Web servers. It provides one common API to a number of different communication protocols.
DataSocket control	ActiveX control that implements the DataSocket client functionality. Enables applications to exchange data with other data sources and targets.
DataSocket Server	Executable that enables data exchange between multiple applications, each of which uses a DataSocket client. The server accepts and stores information from data sources and relays it to other data targets.
DataSocket Server Manager	Executable to configure the default settings of the DataSocket Server, including access permission, number of connections, and predefined data items stored in the server.
DC	Direct Current.
DC signal	Signal solely comprised of a non-dynamic component, steady or very slowly changing voltage.
Delphi	Borland Delphi programming environment.
device	Plug-in data acquisition board that can contain multiple channels and conversion devices.

device number	Slot number or board ID number assigned to the board when configured.
DFT	Discrete Fourier Transform.
DI	Digital Input.
DIFF	Differential. A differential input is an analog input consisting of two terminals, both of which are isolated from computer ground and whose difference you measure.
differential measurement	Method to configure your device to read signals so you do not need to connect either input to a fixed reference, such as the earth or a building ground.
digital trigger	TTL level signal having two discrete levels—a high and a low level used to trigger (start or stop) another process.
DIO	Digital Input/Output.
dithering	Addition of Gaussian noise to an analog input signal.
DLL	Dynamic Link Library.
DMA	Direct Memory Access. A method by which you can transfer data to computer memory from a device or memory on the bus (or from computer memory to a device) while the processor does something else. DMA is the fastest method of transferring data to or from computer memory.
DO	Digital Output.
down counter	Performing frequency division on an internal signal.
driver	Software that controls a specific hardware device, such as a data acquisition board.
DSP	Digital Signal Processing.
DSTP	DataSocket Transfer Protocol. Protocol based on TCP/IP to exchange data directly between two applications using DataSocket clients. Data is passed through a DataSocket Server between the applications.

E

E-Series Device	Series of enhanced data acquisition devices that include technologies such as DAQ-STC ASIC, Plug and Play compatibility, and the NI-PGIA. Some functionality in the ComponentWorks DAQ controls can be used only by E-Series devices.
EEPROM	Electrically Erased Programmable Read-only Memory. Read-only memory that you can erase with an electrical signal and reprogram.
EISA	Extended Industry Standard Architecture.
Equivalent Time Sampling (ETS)	Analog input data acquisition method in which points are acquired with increasing delay from a fixed point on a repetitive waveform.
event	Object-generated response to some action or change in state, such as a mouse click or x number of points being acquired. The event calls an event handler (callback function), which processes the event. Events are defined as part of an OLE control object.
event handler	*See* callback (function) *and* event.
exception	Error message generated by a control and sent directly to the application or programming environment containing the control.
external trigger	Voltage pulse from an external source that triggers an event such as A/D conversion.

F

FFT	Fast Fourier Transform. A method used to compute the Fourier transform of an image.
FHT	Fast Hartley Transform.
FIFO	First-In, First-Out memory buffer. In a FIFO structure, the first data stored is the first data sent to the acceptor.
File I/O	Saving and loading data to and from files in an application.
filtering	Type of signal conditioning that allows you to filter unwanted signals from the signal you are trying to measure.

FIR	Finite Impulse Response.
fires	Occurs. An event fires in response to predefined conditions, such as the completion of a specified interval of time with a timer control, the acquisition of a specified number of data points with a CWAI control, or a mouse-click on a CWButton.
floating signal sources	Signal sources with voltage signals that are not connected to an absolute reference or system ground. Also called *nonreferenced signal sources*. Common example of floating signal sources include batteries, transformers, and thermocouples.
form	Window or area on the screen on which you place controls and indicators to create the user interface for your program.
Format	Flexible specification that defines how a number is displayed on an axis or on some other display. The specification is a format string for formatting all values on a specific display. You specify the format string in the property sheet of a control.
FSR	Frequency Shift Keying. An advanced pulse generation mode in which the output frequency is switched by another digital signal.
FTP	File Transfer Protocol. Protocol based on TCP/IP to exchange files between computers.
FTP Server	Application running on a computer that enables the storing and retrieving of files by different clients via FTP. Most FTP servers allow anonymous connections so that any networked user can exchange files.
function tree	Hierarchical structure in which the functions in a library or an instrument driver are grouped. The function tree simplifies access to a library or instrument driver by presenting functions organized according to the operation they perform, as opposed to a single linear listing of all available functions.

G

gain	Factor by which a signal is amplified, sometimes expressed in decibels.
GATE input pin	Counter input pin that controls when counting in your application occurs.

GPIB	General Purpose Interface bus. The standard bus used for controlling electronic instruments with a computer. Also called IEEE 488 bus because it is defined by ANSI/IEEE Standards 488-1978, 488.1-1987, and 488.2-1987.
grounded measurement system	*See* referenced single-ended measurement system.
grounded signal sources	Signal sources with voltage signals that are referenced to a system ground, such as the earth or a building ground. Also called *referenced signal sources*.
GUI	Graphical User Interface.

H

handshaked digital I/O	Type of digital acquisition/generation where a device or module accepts or transfers data after a digital pulse has been received. Also called *latched digital I/O*.
hardware triggering	Form of triggering where you set the start time of an acquisition and gather data at a known position in time relative to a trigger signal.
hex	Hexadecimal.
HTML	HyperText Markup Language. Language (syntax) used to build Web pages. HTML files are downloaded from an HTTP server and viewed in a Web browser.
HTTP	HyperText Transfer Protocol. Protocol based on TCP/IP, which is used to download Web pages from an HTTP server to a Web browser.
HTTP Server	Application running on a computer that serves Web pages and other information to client computers using HTTP. Clients display Web pages in Web browsers but can retrieve information using other tools, like a DataSocket client.
Hz	Hertz. The number of scans read or updates written per second.

I

I/O	Input/Output. The transfer of data to or from a computer system involving communications channels, operator interface devices, and/or data acquisition and control interfaces.
I/O Connector	Connector on a data acquisition device used to connect the device to external signals or devices.
ICtr(82C53)	Simple counter/timer chip used on 1200-, 700-, 500-, Lab-, and LPM- series data acquisition devices. ICtr(82C53) supports limited counter pulse capabilities and can be controlled using the ICtr functions in the ComponentWorks DAQTools control.
IDFT	Inverse Discrete Fourier Transform.
IEEE	Institute of Electrical and Electronic Engineers.
IEEE 488	Shortened notation for ANSI/IEEE Standards 488-1978, 488.1-1987, and 488.2-1987. See also *GPIB*.
IFFT	Inverse Fast Fourier Transform.
IFHT	Inverse Fast Hartley Transform.
IIR	Infinite Impulse Response.
immediate digital I/O	Type of digital acquisition/generation where LabVIEW updates the digital lines or port states immediately or returns the digital value of an input line. Also called *nonlatched digital I/O*.
in.	Inches.
input limits	The upper and lower voltage inputs for a channel. You must use a pair of numbers to express the input limits.
instrument driver	Library of functions to control and use one specific physical instrument. Also a set of functions that adds specific functionality to an application.
interrupt	Signal indicating that the central processing unit should suspend its current task to service a designated activity.
interval clock	Clock used in a DAQ device in an analog input operation to control the delay between samples acquired from consecutive channels in a scan.

interval delay	Delay between samples acquired from consecutive channels in a scan during an analog input operation.
interval scanning	Scanning method where there is a longer interval between scans than there is between individual channels comprising a scan.
ISA	Industry Standard Architecture.
isolation	Type of signal conditioning in which you isolate the transducer signals from the computer for safety purposes. This protects you and your computer from large voltage spikes and makes sure the measurements from the DAQ device are not affected by differences in ground potentials.

K

ksamples	1,000 samples.
Kwords	1,024 words of memory.

L

latched digital I/O	Type of digital acquisition/generation where a device or module accepts or transfers data after a digital pulse has been received. Also called *handshaked digital I/O*.
LED	Light-Emitting Diode.
limit settings	Maximum and minimum voltages of the analog signals you are measuring or generating.
linearization	Type of signal conditioning that linearizes the voltage levels from transducers, so the voltages can be scaled to measure physical phenomena.
LSB	Least Significant Bit.

M

MB	Megabytes of memory.
memory buffer	*See* buffer.

method — Function that performs a specific action on or with an object. The operation of the method often depends on the values of the object properties.

mse — Mean squared error.

multibuffered I/O — Input operation for which you allocate more than one memory buffer so you can read and process data from one buffer while the acquisition fills another.

multiplexed mode — SCXI operating mode in which analog input channels are multiplexed into one module output so that your cabled DAQ device has access to the module's multiplexed output as well as the outputs on all other multiplexed modules in the chassis through the SCXI bus. Also called *serial mode*.

multiplexer — Set of semiconductor or electromechanical switches with a common output that can select one of a number of input signals and that you commonly use to increase the number of signals measured by one ADC.

N

Named Channel — Channel configuration that specifies a DAQ device; a hardware-specific channel string; channel attributes such as input limits, input mode, and actuator type; and a scaling formula for making a measurement or generating a signal in terms of your actual physical quantity

NI-488 — Driver-level software to control and communicate with GPIB cards and devices.

NI-DAQ — Driver-level software to control and communicate with GPIB cards and devices.

nonlatched digital I/O — Type of digital acquisition/generation where the DIO control updates the digital lines or port states immediately or returns the digital value of an input line. Also called *immediate digital I/O*.

nonreferenced signal — Signal sources with voltage signals that are not connected to an absolute reference or system ground. Also called *floating signal sources*. Common example of non-referenced signal sources include batteries, transformers, and thermocouples.

nonreferenced single-ended — Measurement where the voltage can vary with respect to the measurement system ground. All measurements are made with respect to a common reference.

NRSE — Nonreferenced Single-Ended.

O

object — Software tool for accomplishing tasks in different programming environments. An object can have properties, methods, and events. You change an object's state by changing the values of its properties. An object's behavior consists of the operations (methods) that can be performed on it and the accompanying state changes.

See property, method, event.

object browser — Dialog window that displays the available properties and methods for the controls that are loaded. The object browser shows the hierarchy within a group of objects. To activate the object browser in Visual Basic, press <F2>.

OCX — OLE Control eXtension. Another name for ActiveX controls, reflected by the .OCX file extension of ActiveX control files.

OLE — Object Linking and Embedding. *See* ActiveX.

OLE control — *See* ActiveX control.

onboard channels — Channels on the plug-in data acquisition board.

OUT output pin — Counter output pin where the counter can generate various TTL pulse waveforms.

output limits — Upper and lower voltage or current outputs for an analog output channel. The output limits determine the polarity and voltage reference settings for a board.

P

parallel mode	Type of SCXI operating mode in which the module sends each of its input channels directly to a separate input channel of the device to the module.
pattern	One update (input or output) on a digital port. The number of updates on a buffered digital operation is measured in number of patterns.
pattern generation	Type of handshaked (latched) digital I/O in which internal counters generate the handshaked signal, which in turn initiates a digital transfer. Because counters output digital pulses at a constant rate, you can generate and retrieve patterns at a constant rate because the handshaked signal is produced at a constant rate.
pause condition	Condition on a data acquisition process that determines when the acquisition is paused. The condition can be a digital hardware signal or the state of an analog hardware signal relative to set limits.
PFI	Programmable Function Input. Input and output lines on the I/O controller of E-series data acquisition devices.
PGIA	Programmable Gain Instrumentation Amplifier.
Plot	CWGraph group of methods that displays a new set of data while deleting any previous data on the graph. A plot also refers to one of the traces (data lines) on a graph representing the data in one row or column of an array. Each plot on the graph has its own properties, such as color, style, and so on.
Plug and Play devices	Devices that do not require dip switches or jumpers to configure resources on the devices. Also called *switchless devices*.
PnP	*See* Plug and Play devices.
Pointer	Indicator on a CWSlide or CWKnob object. You can use a collection of pointers to display different values on the same object. In the collection, each pointer is referenced by an index in the collection and each individual pointer has its own properties such as color, style, mode, and so on.
postriggering	Technique used on data acquisition boards to acquire a programmed number of samples after trigger conditions are met.

pretriggering — Technique used on data acquisition boards to keep a continuous buffer filled with data. When the trigger conditions are met, the sample includes the data leading up to the trigger condition.

property — Attribute that controls the appearance or behavior of an object. The property can be a specific value or another object with its own properties and methods. For example, a value property is the color (property) of a plot (object), while an object property is a specific Y axis (property) on a graph (object). The Y axis itself is another object with properties, such as minimum and maximum values.

pulse — Physical signal generated by the Pulse control. Pulse uses TTL levels on most data acquisition devices.

pulse delay — Amount of time from the start of a pulse generation until the active phase of the pulse, measured in seconds. In a positive polarity pulse, the output from the counter is low for the duration of the pulse delay before going high.

pulse period — Period of a continuous or finite pulse train. The period is defined as the amount of time between two consecutive rising or falling edges of the signal. The period is the inverse of the frequency of the pulse train.

pulse trains — Multiple pulses.

pulse width — Width of a pulse generated by a counter on a data acquisition device; measured in seconds.

pulsed output — Form of counter signal generation by which a pulse is output when a counter reaches a certain value.

R

reference — Link to an external code source in Visual Basic. References are anything that add additional code to your program, such as OLE controls, DLLs, objects, and type libraries. You can add references by selecting the **Tools»References…** menu.

referenced single-ended — Measurement made with respect to a common reference (RSE) or a ground. Also called a *grounded measurement system*.

RMS — Root Mean Square.

row-major order — Systematic way to organize the data in a 2D array by rows.

RS-232	Standard serial bus on a computer used to communicate with instruments. Commonly referred to as serial communication.
RSE	Referenced Single-Ended.
RTD	Resistance Temperature Detector. A temperature-sensing device in which resistance increases with increases in temperature.
RTSI	Real-Time System Integration bus. The National Instruments timing bus that interconnects data acquisition boards directly, by means of connectors on top of the boards, for precise synchronization of functions.

S

sample	Single (one and only one) analog or digital input or output data point.
sample counter	Clock that counts the output of the channel clock (the number of samples taken). On boards with simultaneous sampling, this counter counts the output of the scan clock and hence the number of scans.
scan	One or more analog or digital input samples. Typically, the number of input samples in a scan is equal to the number of channels in the input group. For example, one pulse from the scan clock produces one scan that acquires one new sample from every analog input channel in the group.
scan clock	Clock controlling the time interval between scans. On boards with interval scanning support (for example, the AT-MIO-16F-5), this clock gates the channel clock on and off. On boards with simultaneous sampling, this clock clocks the track-and-hold circuitry.
scan rate	Number of scans per second. For example, at a scan rate of 10Hz, each channel is sampled 10 times per second.
scan width	Number of channels in the channel list or number of ports in the port list you use to configure an analog or digital input group.
SCXI	Signal Conditioning eXtensions for Instrumentation. The National Instruments product line for conditioning low-level signals within an external chassis near sensors, so only high-level signals in a noisy environment are sent to data acquisition boards.
sec	Seconds.

serial	Standard serial bus on a computer used to communicate with instruments. Also known as RS-232.
settling time	Amount of time required for a voltage to reach its final value within specified limits.
signal conditioning	Manipulation of analog signals to prepare them for digitizing.
signal divider	Performing frequency division on an external signal.
single-ended inputs	Analog inputs that you measure with respect to a common ground.
Snap Mode	Mode that controls the available coordinates for cursors to line up on a plot.
software analog triggering	Method of triggering in which you simulate an analog trigger using software.
SOURCE input pin	Counter input pin where the counter counts the signal transitions.
start condition	Condition on a data acquisition process that determines when the actual acquisition starts. The condition can be a software trigger, an analog hardware trigger, or a digital hardware trigger.
STC	System Timing Controller. *See* DAQ-STC.
stop condition	Condition on a data acquisition process that determines when the acquisition stops. The condition can be none (the acquisition stops when all points have been acquired), continuous (the acquisition runs continuously), software analog trigger, hardware analog trigger, or hardware digital trigger.
strain gauge	Thin conductor attached to a material that detects stress or vibrations in that material.
Style	Display style of a GUI object. An object can have different display styles while maintaining the same functionality. For example, the button can be an LED, toggle switch, vertical or horizontal slide, push button, command button, and more.
switchless device	Devices that do not require dip switches or jumpers to configure resources on the devices. Also called *Plug and Play devices*.

synchronous — Property or operation that begins an operation and returns control to the program only when the operation is complete.

syntax — Set of rules to which statements must conform in a particular programming language.

T

TC — Terminal Count. The highest value of a counter.

thermocouple — Template-sensing device that measures temperature by the changing electrical potential between two different metals.

toggled output — Form of counter signal generation by which the output changes the state of the output signal from high to low or from low to high when the counter reaches a certain value.

trace — Data line on a graph. Traces are generated using the Plot or Chart methods of the graph. Also called a *plot*.

track-and-hold — Circuit that tracks an analog voltage and holds the value on command.

Track Mode — Property of a graph that specifies how the mouse interacts with the graph. Use the Track Mode to turn on zooming and panning for the graph.

transducer excitation — Type of signal conditioning that uses external voltages and currents to excite the circuitry of a signal conditioning system into measuring physical phenomena.

trigger — Condition for starting or stopping clocks.

TTL — Transistor-Transistor Logic. Specifications for digital signals.

U

UI — User Interface.

unipolar — Signal range that is always positive, for example, 0 to 10 V.

update — One or more analog or digital output samples. Typically, the number of output samples in an update is equal to the number of channels in the output group. For example, one pulse from the update clock produces one update that sends one new sample to every analog output channel in the group.

update clock	Clock that sets the update rate for the AO, DI, and DO controls.
update rate	Number of output updates per second.
update width	Number of channels in the channel list or number of ports in the port list you use to configure an analog or digital output group.
URL	Uniform Resource Locator. A standard that uses a descriptive string to identify the source or connection for a data transfer used with DataSocket, FTP, HTTP, and file I/O. The URL includes the communication protocol (scheme), name or identifier of the computer being accessed, and scheme-specific information identifying a file, Web page, or, when using DSTP, the data item.

V

V	Volts.
Value Pairs	Pair that consists of a name and a value that you can use for custom ticks, labels, and grid lines on the axis of a knob, slide, or graph.
Value Pairs Only control	Control whose only valid values are its value pairs.
VB	Microsoft Visual Basic.
VC++	Microsoft Visual C++.
VDC	Volts, Direct Current.
VISA	Driver software architecture developed by National Instruments to unify instrumentation software—VXI, GPIB, and serial port. It has been accepted as a standard for VXI by the VXI*plug&play* Systems Alliance.
VISA Resource	String identifying a specific instrument connected to the computer and controlled using the VISA control and driver.
Vref	Voltage reference.
VXI	VME eXtension for Instrumentation. Instrumentation architecture and bus based on the VME standard. Used in high-end test applications.

W

waveform	Multiple voltage readings taken at a specific sampling rate.
Web Server	See *HTTP server*.

Index

Numbers

1D and 2D operations (table), 10-3

A

AcquireData method, 7-11 to 7-12
ActiveX controls. *See also* events; methods; properties.
    getting started with using, 2-5 to 2-6
    installing, 1-4
    setting properties, 1-10 to 1-15
Advanced Analysis Library, 10-2
advanced application development. *See* application development.
AI control, 7-9 to 7-14
    AI object, 7-10
    asynchronous acquisition, 7-10 to 7-11
    ChannelClock object, 7-12 to 7-13
    error handling, 7-12
    methods and events, 7-10 to 7-12
    object hierarchy (figure), 7-9
    PauseCondition object, 7-13 to 7-14
    ScanClock object, 7-12 to 7-13
    StartCondition object, 7-13 to 7-14
    StopCondition object, 7-13 to 7-14
    synchronous acquisition, 7-11 to 7-12
    tutorial, 7-14 to 7-18
        developing code, 7-16 to 7-17
        form design, 7-15
        setting DAQ properties, 7-16
        testing the program, 7-18
AI object, 7-10
AIPoint control, 7-6 to 7-8
    AIPoint object, 7-6 to 7-7
    Channel object, 7-8
    ChannelClock object, 7-8
    Channels collection, 7-7 to 7-8
    object hierarchy (figure), 7-6
    tutorial, 7-14 to 7-18
        developing code, 7-16 to 7-17
        form design, 7-15
        setting DAQ properties, 7-16
        testing the program, 7-18
AIPoint object, 7-6 to 7-7
analog input, single point. *See* AIPoint control.
analog output, single point. *See* AOPoint control.
Analysis Library controls
    controls, 10-12 to 10-13
    descriptions in online reference manual, 10-13
    error codes (table), D-20 to D-23
    error messages, 10-13
    function tree (table)
        CWArray control, 10-3 to 10-4
        CWComplex control, 10-4 to 10-5
        CWDSP control, 10-8 to 10-12
        CWMatrix control, 10-5 to 10-6
        CWStat control, 10-6 to 10-7
    list of controls (table), 10-1
    overview, 1-2
    purpose and use, 1-15
    questions and answers, C-14
    statistics function tutorial, 10-14 to 10-17
        developing code, 10-16 to 10-17
        form design, 10-15
        testing the program, 10-17
    versions of the library, 10-2
analysis of variance functions (table), 10-7
AO control, 7-20 to 7-24
    AO object, 7-21
    IntervalClock object, 7-22 to 7-23
    methods and events, 7-21 to 7-22
    object hierarchy (figure), 7-20

StartCondition object, 7-23 to 7-24
UpdateClock object, 7-22 to 7-23
AO object, 7-21
AOPoint control, 7-18 to 7-20
AOPoint object, 7-19
methods, 7-19 to 7-20
object hierarchy (figure), 7-19
tutorial, 7-24 to 7-28
developing code, 7-25 to 7-27
form design, 7-24 to 7-25
testing the program, 7-27 to 7-28
AOPoint object, 7-19
application development
ActiveX controls, 2-5 to 2-6
advanced features of ComponentWorks, 12-1 to 12-12
Virtual Data Logger application, 12-9 to 12-12
Virtual Oscilloscope application, 12-1 to 12-4
Virtual Spectrum Meter application, 12-4 to 12-9
Delphi applications, 5-1 to 5-9
distributing applications using ComponentWorks, C-2
examples included with ComponentWorks, 2-4 to 2-5
installing for specific programming environment, 2-5
learning to use controls, 2-4
using as basis for applications, 2-6 to 2-7
getting started, 2-5 to 2-7
testing and debugging, 12-12 to 12-18
breakpoints, 12-17 to 12-18
Debug.Print command, 12-17
error and warning events, 12-15 to 12-16
error checking, 12-12 to 12-14
exceptions, 12-13 to 12-14
GetErrorText function, 12-16 to 12-17
return codes, 12-14 to 12-15
single stepping, 12-18
Step Into mode, 12-18
Step Over mode, 12-18
watch window, 12-18
Visual Basic applications, 3-1 to 3-12
Visual C++ applications, 4-1 to 4-10
array operation functions (table)
1D and 2D operations, 10-3
multidimensional array operations, 10-4
multidimensional element operations, 10-3 to 10-4
ASRL (Serial) object, 9-8
asynchronous acquisition methods and events, 7-10 to 7-11
asynchronous I/O
GPIB control, 8-8 to 8-9
message-based communication, 9-10
Serial control, 8-14
AT-A2150 data acquisition card, C-9
AT-DSP2200 data acquisition card, C-9
attributes, CWData object, 11-12
axes, Virtual Data Logger application
formats, 12-11 to 12-12
multiple, 12-10 to 12-11
Axes collection, 6-20
Axis object, 6-4 to 6-5, 6-20

B

Base Analysis Library, 10-2
Borland Delphi. *See* Delphi applications.
breakpoints, 12-17 to 12-18
buffered measurements, Counter control, 7-50
buffered waveform digital input. *See* DI control.
buffers, data acquisition programming considerations, B-4
bulletin board support, F-1

Button control
events, 6-13
Graph and Button control tutorial, 6-22 to 6-25
developing program code, 6-24 to 6-25
form design, 6-22 to 6-23
testing the program, 6-25
purpose and use, 6-12

C

calibration functions, 7-60
channel clocks, B-4
Channel object, 7-8
channel strings
devices requiring reverse list of channels (note), 7-4
purpose and use, 7-3 to 7-4
SCXI channel strings
data acquisition controls, 7-4 to 7-5
data acquisition programming considerations, B-6 to B-7
Channel Wizard, B-3
ChannelClock object
AI control, 7-12 to 7-13
AIPoint control, 7-8
channels
data acquisition programming considerations, B-3 to B-4
questions and answers, C-9 to C-10
Channels collection, 7-7 to 7-8
charting data, 6-13. *See also* Graph control.
ChartXvsY method, 6-16
ChartXY method, 6-16
ChartY method, 6-16, 6-25
clearing instruments, with GPIB control, 8-9
Clock objects
ChannelClock
AI control, 7-12 to 7-13
AIPoint control, 7-8
IntervalClock, 7-22 to 7-23
ScanClock, 7-12 to 7-13
UpdateClock
AO control, 7-22 to 7-23
DI control, 7-36
DO control, 7-39 to 7-40
Clock (Source) input, 7-45
clocks
channel clocks, B-4
data acquisition programming considerations, B-4
scan clocks, B-5
collection objects, 1-9 to 1-10
collections
Axes, 6-20
Channels, 7-7 to 7-8
Cursors, 6-18 to 6-19
definition, 1-9
Lines, 7-32
Plots, 6-16 to 6-18
Pointers, 6-4
Ports, 7-31
ValuePairs collection, 6-5 to 6-6
common questions. *See* questions about ComponentWorks.
complex number functions (table), 10-4 to 10-5
complex operation functions, multidimensional (table), 10-5
ComponentWorks
application development overview, 2-5 to 2-7
components, 1-1 to 1-2
examples installed with, 2-4 to 2-5
general error codes, D-24 to D-26
information sources
additional sources, 2-7
documentation, 2-2 to 2-3
online reference information, 2-3 to 2-4

installing and configuring driver software, 2-1 to 2-2
overview, 1-1 to 1-2
questions about, C-1 to C-14
system requirements, 1-3
configuration, for data acquisition, B-2 to B-3
DAQ Channel Wizard, B-3
device number, B-2 to B-3
overview, 7-2
SCXI hardware, B-2
Configuration method, Pulse control, 7-53
Configure functions, 7-60
Configure method
AO control, 7-21 to 7-22
asynchronous acquisition, AI control, 7-10 to 7-11
Counter control, 7-48
DO control, 7-40
Connect method, DataSocket control, 11-2, 11-9
ConnectTo method, DataSocket control, 11-3
control methods. *See* methods.
Conversion functions, 7-60
CopyFrom method, 11-10
Counter control, 7-46 to 7-50
buffered measurements, 7-50
methods and events, 7-48 to 7-49
object hierarchy (figure), 7-46
tutorial, 7-55 to 7-59
code development, 7-56 to 7-58
form design, 7-55 to 7-56
testing the program, 7-59
Counter object, 7-46 to 7-48
buffered measurements, 7-50
measurement types (table), 7-47
counter/timer hardware
Counter control, 7-46 to 7-50
Pulse control, 7-51 to 7-54
questions and answers, C-11
tutorial, 7-55 to 7-59
Cursor object, 6-19
cursors, Virtual Spectrum Meter application, 12-7 to 12-8
Cursors collection, 6-18 to 6-19
curve fitting functions (table), 10-7
custom property page
CWGraph control example (figure), 1-11
definition, 1-11
customer communication, *xxiii*, F-1 to F-2
CWArray control function tree, 10-3 to 10-4
1D and 2D operations, 10-3
multidimensional array operations, 10-4
multidimensional element operations, 10-3 to 10-4
CWButton control (table), 6-2
CWComplex control function tree, 10-4 to 10-5
complex numbers, 10-4 to 10-5
multidimensional complex operations, 10-5
CWData object, 11-11 to 11-13
standalone CWData objects, 11-12 to 11-13
working with attributes, 11-12
CWDSP control function tree, 10-8 to 10-12
FIR digital filters, 10-10 to 10-11
frequency domain signal processing, 10-8 to 10-9
IIR digital filters, 10-9 to 10-10
measurement, 10-11
signal generation, 10-8
time domain signal processing, 10-9
windows, 10-11
CWGPIB object, 8-7
CWGraph control
associated styles (table), 6-2
custom property page (figure), 1-11
CWKnob control (table), 6-2
CWMatrix control function tree, 10-5 to 10-6
CWNumEdit control (table), 6-2
CWPattern object, 8-5

CWSerial object, 8-12 to 8-13
CWSlide control (table), 6-2
CWStat control function tree, 10-6 to 10-7
analysis of variance, 10-7
curve fitting, 10-7
interpolation, 10-7
nonparametric statistics, 10-7
probability distributions, 10-6 to 10-7
statistics, 10-6
CWTask object, 8-4
CWToken object, 8-6

D

DAQ Channel Wizard, B-3
DAQ controls. *See* data acquisition controls.
DAQCard-700, C-11
DAQCard-1200, C-11
DAQTools, 7-60 to 7-61
function groups, 7-60
using DAQTools functions, 7-61
data acquisition
configuration, B-2 to B-3
DAQ Channel Wizard, B-3
device number, B-2 to B-3
overview, 7-2
SCXI hardware, B-2
hardware, B-7 to B-9
PFI, B-9
RTSI, B-8
SCXI, B-7 to B-8
installing hardware and software, B-1 to B-2
programming considerations, B-3 to B-9
buffers, B-4
channel clocks, B-5
clocks, B-5
device numbers and channels, B-3 to B-4
scan clocks, B-5
SCXI channel string, B-6 to B-7
Data Acquisition controls, 7-1 to 7-61
AI control, 7-9 to 7-14
AI object, 7-10
asynchronous acquisition, 7-10 to 7-11
ChannelClock object, 7-12 to 7-13
error handling, 7-12
methods and events, 7-10 to 7-12
PauseCondition object, 7-13 to 7-14
ScanClock object, 7-12 to 7-13
StartCondition object, 7-13 to 7-14
StopCondition object, 7-13 to 7-14
synchronous acquisition, 7-11 to 7-12
AIPoint and AI DAQ control tutorial, 7-14 to 7-18
developing code, 7-16 to 7-17
form design, 7-15
setting DAQ properties, 7-16
testing the program, 7-18
AIPoint control, 7-6 to 7-8
AIPoint object, 7-6 to 7-7
Channel object, 7-8
ChannelClock object, 7-8
Channels collection, 7-7 to 7-8
AO control, 7-20 to 7-24
AO object, 7-21
IntervalClock object, 7-22 to 7-23
methods and events, 7-21 to 7-22
object hierarchy (figure), 7-20
StartCondition object, 7-23 to 7-24
UpdateClock object, 7-22 to 7-23
AOPoint control, 7-18 to 7-20
AOPoint object, 7-19
methods, 7-19 to 7-20
AOPoint control tutorial, 7-24 to 7-28
developing code, 7-25 to 7-27
form design, 7-24 to 7-25
testing the program, 7-27 to 7-28

Counter and Pulse control tutorial, 7-55 to 7-59
code development, 7-56 to 7-58
form design, 7-55 to 7-56
testing the program, 7-59
Counter control, 7-46 to 7-50
buffered measurements, 7-50
Counter object, 7-46 to 7-48
measurement types (table), 7-47
methods and events, 7-48 to 7-49
counter/timer hardware, 7-45 to 7-54
DAQTools, 7-60 to 7-61
function groups, 7-60
using DAQTools functions, 7-61
data acquisition configuration, 7-2
DI control, 7-34 to 7-37
DI object, 7-35
methods and events, 7-36 to 7-37
UpdateClock object, 7-36
digital controls and hardware, 7-28 to 7-41
DIO control, 7-29 to 7-34
common properties and methods, 7-32 to 7-34
DIO object, 7-30 to 7-31
Ports collection and Port object, 7-31
DIO control tutorial, 7-41 to 7-45
developing code, 7-42 to 7-44
form design, 7-42
testing the program, 7-44 to 7-45
DO control, 7-37 to 7-41
DO object, 7-38 to 7-39
methods and events, 7-40 to 7-41
UpdateClock object, 7-39 to 7-40
error codes (table), D-1 to D-16
Lines collection and Line object, 7-32
common properties and methods, 7-32 to 7-34
list of DAQ controls, 7-2
object hierarchy and common properties, 7-2 to 7-6
channel strings, 7-3 to 7-4
Device, DeviceName, and DeviceType, 7-3
ExceptionOnError and ErrorEventMask, 7-5 to 7-6
SCXI channel strings, 7-4 to 7-5
overview, 1-1, 7-1 to 7-2
Pulse control, 7-51 to 7-54
FSK and ETS pulse generation, 7-54
methods, 7-53
Pulse object, 7-51 to 7-53
questions and answers, C-8 to C-11
data acquisition utility functions (DAQTools), 7-60 to 7-61
Data Logger application. *See* Virtual Data Logger application.
data source, for DataSocket control. *See* DataSocket control.
DataAsString property
GPIB control and Serial control, 8-2
VISA control, 9-4
DataSocket control, 11-1 to 11-17
basic principles, 11-2 to 11-3
CWData object, 11-11 to 11-13
standalone CWData objects, 11-12 to 11-13
working with attributes, 11-12
disconnecting from data source, 11-5 to 11-6
locating data source, 11-3
OnDataUpdated event, 11-4 to 11-5
OnStatusUpdated event, 11-5 to 11-6
overview, 1-2, 11-1 to 11-2
reading data from data source, 11-3 to 11-6
setting up DataSocket Server, 11-13 to 11-15

tutorial for reading waveform, 11-6 to 11-9
code development, 11-7 to 11-8
form design, 11-6 to 11-7
testing the program, 11-8 to 11-9
updating data, 11-5
automatically, 11-5
writing data to data target, 11-9 to 11-10
automatically updating data target, 11-10
updating data target, 11-10
DataSocket Server, 11-13 to 11-15
connecting to and reading data items, 11-15
creating data items, 11-14
distributing applications, E-3
requirements for running, 11-14
status checking, 11-14
tutorial for sharing data between applications, 11-15 to 11-17
configuring DataSocket Server, 11-16 to 11-17
procedural steps, 11-15 to 11-16
debugging and testing applications, 12-12 to 12-18
breakpoints, 12-17 to 12-18
Debug.Print command, 12-17
error and warning events, 12-15 to 12-16
error checking, 12-12 to 12-14
exceptions, 12-13 to 12-14
GetErrorText function, 12-16 to 12-17
return codes, 12-14 to 12-15
single stepping, 12-18
Step Into mode, 12-18
Step Over mode, 12-18
watch window, 12-18
default property sheet
definition, 1-10
Visual Basic example (figure), 1-11
DeleteAttribute method, 11-12
Delphi 2, A-6 to A-7
Delphi applications, 5-1 to 5-9
building user interface, 5-4 to 5-6
editing properties programmatically, 5-6 to 5-7
events, 5-8 to 5-9
loading ComponentWorks controls, 5-2 to 5-4
methods, 5-7 to 5-8
online help for learning controls, 5-9
placing controls, 5-4 to 5-5
programming with ComponentWorks controls, 5-6 to 5-9
property sheets, 5-5 to 5-6
using Delphi 2, A-6 to A-7
developing applications. *See* application development.
device numbers
configuration, B-2 to B-3
programming considerations, B-3 to B-4
Device property, 7-3
DeviceName property, 7-3
DeviceType property, 7-3
DI control, 7-34 to 7-37
methods and events, 7-36 to 7-37
object hierarchy (figure), 7-34
UpdateClock object, 7-36
DI object, 7-35
digital controls and hardware, 7-28 to 7-41
common properties and methods, 7-32 to 7-34
DI control—buffered waveform digital input, 7-34 to 7-37
DI object, 7-35
DIO control—single point digital input and output, 7-29 to 7-34
DIO object, 7-30 to 7-31
DO control—buffered waveform digital output, 7-37 to 7-41
DO object, 7-38 to 7-39

Lines collection and Line object, 7-32
methods and events
DI control, 7-36 to 7-37
DO control, 7-40 to 7-41
Ports collection and Port object, 7-31
UpdateClock object
DI control, 7-36
DO control, 7-39 to 7-40
Digital Signal Processing Analysis Library, 10-2
digital signal processing and signal generation functions
FIR digital filters (table), 10-10 to 10-11
frequency domain signal processing (table), 10-8 to 10-9
IIR digital filters (table), 10-9 to 10-10
measurement functions (table), 10-11
signal generation functions (table), 10-8
time domain signal processing functions (table), 10-9
Virtual Spectrum Meter application, 12-5 to 12-7
windows functions (table), 10-11
DIO control
common properties and methods, 7-32 to 7-34
Lines collection and Line object, 7-32
object hierarchy (figure), 7-30
Ports collection and Port object, 7-31
single point digital input and output, 7-29 to 7-34
tutorial, 7-41 to 7-45
developing code, 7-42 to 7-44
form design, 7-42
testing the program, 7-44 to 7-45
DIO object
common properties and methods, 7-32 to 7-34
purpose and use, 7-30 to 7-31
DisableEvent method, 9-16
DiscardEvent method, 9-16
disconnecting from data source, 11-5 to 11-6
distributing applications, E-1 to E-5
automatic installers, E-2
ComponentWorks evaluation mode, E-4
DataSocket Server, E-3
general considerations, E-1
instrument drivers, E-3
manual installation, E-2 to E-3
questions and answers, C-2
required files, E-1
run-time licenses, E-4
troubleshooting, E-4 to E-5
DO control, 7-37 to 7-41
methods and events, 7-40 to 7-41
object hierarchy (figure), 7-38
UpdateClock object, 7-39 to 7-40
DO object, 7-38 to 7-39
documentation
conventions used in manual, *xxii-xxiii*
organization of manual, *xix-xxii*, 2-2 to 2-3
related documentation, *xxiii*
driver software for hardware I/O controls, installing, 2-1 to 2-2
DSP functions. *See* digital signal processing and signal generation functions.
dstp: (DataSocket transfer protocol) scheme, 11-3

E

EISA-A2000 data acquisition card, C-9
electronic support services, F-1 to F-2
e-mail support, F-2
EnableEvent method, 9-17
equivalent time sampling (ETS) pulse generation, 7-54
error and warning events, 12-15 to 12-16
error checking/handling
AI control, 7-12
DAQ controls, 7-5 to 7-6

GetErrorText function, 7-60, 12-16 to 12-17
testing applications, 12-12 to 12-14
error codes
Analysis controls (table), D-20 to D-23
data acquisition controls (table), D-1 to D-16
general ComponentWorks codes, D-24 to D-26
VISA controls (table), D-17 to D-19
error messages
Analysis Library controls, 10-13
exceptions, 12-13 to 12-14
ErrorEventMask property, 7-5 to 7-6
ETS (equivalent time sampling) pulse generation, 7-54
event handler routines
developing, 1-14 to 1-15
Visual Basic applications, 3-6 to 3-7
event handling with event queue, VISA control, 9-15 to 9-17
checking events, 9-16
disabling events, 9-16 to 9-17
discarding events, 9-16
event types, VISA control, 9-14 to 9-15
events
AO control, 7-21 to 7-22
asynchronous acquisition, 7-10 to 7-11
Button control, 6-13
Counter control, 7-48 to 7-49
CWTask object, 8-4
definition, 1-7
DI control, 7-36 to 7-37
DO control, 7-40 to 7-41
error and warning events, 12-15 to 12-16
GPIB control, 8-8 to 8-9
Graph control, 6-21
Knob and Slide controls, 6-6 to 6-7
Numeric Edit Box control, 6-7 to 6-8
OnDataUpdated, 11-4 to 11-5
OnStatusUpdated, 11-5 to 11-6
Serial control, 8-13 to 8-14
synchronous acquisition, 7-11 to 7-12
using in applications
Delphi applications, 5-8 to 5-9
Visual C++ applications, 4-9 to 4-10
VISA control, 9-14 to 9-17
examples included with ComponentWorks, 2-4 to 2-5
installing for specific programming environment, 2-5
learning about controls, 2-4
using for application development, 2-6 to 2-7
ExceptionOnError property
GPIB control and Serial control, 8-2
purpose and use, 7-5 to 7-6
VISA control, 9-4
exceptions, in application testing, 12-13 to 12-14

F

fax and telephone support numbers, F-2
Fax-on-Demand support, F-2
file: (local files) scheme, 11-3
file input/output functions, Virtual Data Logger application, 12-12
files
installed files, 1-6
required files for distributing applications, E-4 to E-5
FIR digital filters (table), 10-10 to 10-11
forms. *See also* tutorials.
Delphi applications, 5-2
Visual Basic applications, 3-1 to 3-2
FOUT functions, 7-60
frequency domain signal processing (table), 10-8 to 10-9
FSK (frequency shift keying) pulse generation, 7-54

ftp: (file transfer protocol) scheme, 11-3
FTP support, F-1

G

Gate input, 7-45
Get functions, 7-60
GetAttribute method, 11-12
GetErrorText function, 7-60, 12-16 to 12-17
GPIB control, 8-6 to 8-9
    common properties, 8-2
    CWGPIB object, 8-7
    methods and events, 8-8 to 8-9
        asynchronous I/O, 8-8 to 8-9
        other GPIB operations, 8-9
        synchronous I/O, 8-8
    object hierarchy, 8-2
    object hierarchy (figure), 8-7
    overview, 1-1
    parsing, 8-2 to 8-6
        advanced features, 8-2 to 8-6
        CWPattern object, 8-5
        CWTask object, 8-4
        CWToken object, 8-6
    purpose and use, 8-1
    questions and answers, C-12 to C-13
    tutorial, 8-9 to 8-11
        code development, 8-11
        form design, 8-10
        setting properties, 8-10 to 8-11
        testing the program, 8-11
GPIB object, 9-9
Graph and Button control tutorial, 6-22 to 6-25
    developing program code, 6-24 to 6-25
    form design, 6-22 to 6-23
    testing the program, 6-25
graph axes, Virtual Data Logger application
    formats, 12-11 to 12-12
    multiple, 12-10 to 12-11
Graph control, 6-13 to 6-21
    Axes collection, 6-20
    Axis object, 6-20
    chart methods, 6-16
    Cursor object, 6-19
    Cursors collection, 6-18 to 6-19
    events, 6-21
    Graph object, 6-14 to 6-16
    hierarchy of (figure), 6-14
    overview, 1-7 to 1-8
    panning and zooming, 6-21
    plot methods, 6-15
    Plot object, 6-17 to 6-18
    Plots collection, 6-16 to 6-18
    PlotTemplate object, 6-18
    purpose and use, 6-13 to 6-14
    tutorial, 6-22 to 6-25
Graph object, 6-14 to 6-16
graph track mode, Virtual Spectrum Meter application, 12-8 to 12-9
Graphical User Interface (GUI) controls. *See* User Interface controls.

H

HasAttribute method, 11-12
Help button, 1-15
help files, online. *See* online reference.
http: (hypertext transfer protocol) scheme, 11-3

I

ICtr functions, 7-60
IIR digital filters (table), 10-9 to 10-10
In method, 9-11, 9-12
installation, 1-3 to 1-6
    ActiveX controls, 1-4
    from CD-ROM, 1-3

distributed applications, E-2 to E-3
automatic installers, E-2
manual installation, E-2 to E-3
driver software for hardware I/O controls, 2-1 to 2-2
from floppy disks, 1-4
hardware and software for data acquisition, B-1 to B-2
installed files, 1-6
instrument driver DLLs, 1-5
Instrument Driver Factory, 1-5
questions about ComponentWorks, C-1 to C-2
system requirements, 1-3
Instrument controls. *See* GPIB control; Serial control; VISA control.
instrument driver DLLs
installing, 1-5
using in applications, 1-15
Visual Basic applications, 3-7 to 3-8
Instrument Driver Factory
installing, 1-5
overview, 1-2
instrument drivers
definition, 3-7
overview, 1-2
interchannel delay, 7-12
interpolation functions (table), 10-7
IntervalClock object, 7-22 to 7-23
Item method, 1-13

K

Knob and Slide controls, 6-3 to 6-7
Axis object, 6-4 to 6-5
events, 6-6 to 6-7
hierarchy of (figure), 6-3
Knob and Slide object, 6-3 to 6-4
Labels object, 6-5
Pointer object, 6-4
Pointers collection, 6-4
Statistics object, 6-6
Ticks object, 6-5
tutorial, 6-8 to 6-11
developing program code, 6-10 to 6-11
form design, 6-9
testing the program, 6-11
ValuePair object, 6-6

L

Labels object, 6-5
Lab-PC+ card, C-11
Line object, 7-32
Lines collection, 7-32

M

MapAddress method, 9-13
matrix algebra functions (table), 10-5 to 10-6
measurement functions (table), 10-11
measurement types, Counter object (table), 7-47
message-based communication, 9-9 to 9-13
asynchronous I/O, 9-10
synchronous I/O, 9-10
tutorial, 9-17 to 9-19
code development, 9-19
form design, 9-17 to 9-18
setting properties, 9-18 to 9-19
testing the program, 9-19
methods. *See also* specific methods.
AI object, 7-10 to 7-12
asynchronous acquisition, 7-10 to 7-11
synchronous acquisition, 7-11 to 7-12
AO control, 7-21 to 7-22
AOPoint object, 7-19 to 7-20
asynchronous acquisition, 7-10 to 7-11
Counter control, 7-48 to 7-49

CWTask object, 8-4
definition, 1-7
DI control, 7-36 to 7-37
DIO control, 7-32 to 7-34
DO control, 7-40 to 7-41
functions as methods of corresponding control, 10-12
GPIB control, 8-8 to 8-9
Lines collection and Line object, 7-32 to 7-34
parameters, 3-6
Pulse control, 7-53
Serial control, 8-13 to 8-14
synchronous acquisition, 7-11 to 7-12
using in applications
Delphi applications, 5-7 to 5-8
Visual Basic applications, 3-5 to 3-6
Visual C++ applications, 4-8 to 4-9
VISA control, 9-9 to 9-13
working with control methods, 1-13 to 1-14
MoveIn method, 9-12
MoveOut method, 9-12
multidimensional array operations (table), 10-4
multidimensional complex operations (table), 10-5
multidimensional element operations (table), 10-3 to 10-4
multiple graph axes, Virtual Data Logger application, 12-10 to 12-11

N

NI-DAQ driver configuration utility, 7-2
nonparametric statistics functions (table), 10-7
Numeric Edit Box control, 6-7 to 6-8
events, 6-7 to 6-8
purpose and use, 6-7
tutorial, 6-8 to 6-11
developing program code, 6-10 to 6-11
form design, 6-9
testing the program, 6-11

O

Object Browser
Visual Basic 4, A-2 to A-3
Visual Basic 5, 3-8 to 3-10
object hierarchy
AI control (figure), 7-9
AIPoint control (figure), 7-6
AO control (figure), 7-20
AOPoint control (figure), 7-19
Counter control (figure), 7-46
DAQ controls, 7-3 to 7-6
DI control (figure), 7-34
DIO control (figure), 7-30
DO control (figure), 7-38
GPIB and Serial controls, 8-2
GPIB control (figure), 8-7
Graph control (figure), 6-14
Knob and Slide controls (figure), 6-3
Pulse control (figure), 7-51
purpose and use, 1-8 to 1-9
Serial control (figure), 8-12
similarity in different controls, 6-2
Slide object example, 1-9
VISA control, 9-2 to 9-3
OnDataUpdated event, 11-4 to 11-5
one dimensional operations (table), 10-3
online reference, 2-3 to 2-4
accessing, 2-4
descriptions of analysis functions, 10-13
finding specific information, 2-4

learning about ComponentWorks controls, 1-15
Delphi applications, 5-9
Visual Basic applications, 3-12
Visual C++ applications, 4-10
Web Site support, 2-7
OnStatusUpdated event, 11-5 to 11-6
Oscilloscope application. *See* Virtual Oscilloscope application.
Out method, 9-12
Out output, 7-45

P

panning and zooming graphs, 6-21
parallel polling, with GPIB control, 8-9
parameters for methods, 3-6
Parse method, 8-3
parsing
GPIB control, 8-2 to 8-6
advanced features, 8-2 to 8-6
CWPattern object, 8-5
CWTask object, 8-4
CWToken object, 8-6
questions and answers, C-13
VISA control, 9-5
PauseCondition object, 7-13 to 7-14
Peek*XX* method, 9-13
PFIs (Programmable Function Inputs), B-9
Plot object, 6-17 to 6-18
Plots collection, 6-16 to 6-18
PlotTemplate object, 6-18
plotting data, 6-13. *See also* Graph control.
PlotXvsY method, 6-15
PlotXY method, 6-15
PlotY method
changing properties programmatically (example), 1-13 to 1-14
format for accepting data, 6-15
Graph and Button control tutorial, 6-25
Pointer object, 6-4
Pointers collection, 6-4
Poke*XX* method, 9-13
polling, with GPIB control, 8-9
Port object, 7-31
ports, definition, 7-29
Ports collection, 7-31
pretriggering modes, Virtual Oscilloscope application, 12-3
probability distribution functions (table), 10-6 to 10-7
Programmable Function Inputs (PFI), B-9
programming considerations, for data acquisition, B-3 to B-9. *See also* application development.
buffers, B-4
channel clocks, B-5
clocks, B-5
device numbers and channels, B-3 to B-4
scan clocks, B-5
SCXI channel string, B-6 to B-7
properties. *See also* specific property names.
channel strings, 7-3 to 7-4
definition, 1-7
Device, DeviceName, and DeviceType, 7-3
DIO control, 7-32 to 7-34
editing programmatically
Delphi applications, 5-6 to 5-7
Visual Basic applications, 3-4 to 3-5
ExceptionOnError and ErrorEventMask, 7-5 to 7-6
GPIB control, 8-2
Instrument controls, 8-2
Lines collection and Line object, 7-32 to 7-34
Port object, 7-32 to 7-34
SCXI channel strings, 7-4 to 7-5
Serial control, 8-2

setting, 1-10 to 1-15. *See also* tutorials.
Analysis Library and instrument driver DLLs, 1-15
changing properties programmatically, 1-12
developing event handler routines, 1-14 to 1-15
help file, 1-15
Item method, 1-13
property sheets, 1-10 to 1-11
working with control methods, 1-13 to 1-14
VISA object, 9-5 to 9-7
Visual C++ applications, 4-6 to 4-8
property pages, 1-10 to 1-11
custom property page, 1-11
default property sheet, 1-10
Delphi applications, 5-5 to 5-6
setting default values, 1-10 to 1-11
Visual Basic applications, 3-3 to 3-4
Pulse control, 7-51 to 7-54
FSK and ETS pulse generation, 7-54
methods, 7-53
object hierarchy (figure), 7-51
tutorial, 7-55 to 7-59
code development, 7-56 to 7-58
form design, 7-55 to 7-56
testing the program, 7-59
Pulse object, 7-51 to 7-53
pulse type operations (table), 7-52

Q

questions about ComponentWorks, C-1 to C-14
analysis controls, C-14
data acquisition controls, C-8 to C-11
GPIB, Serial, and VISA controls, C-12 to C-13
installation and getting started, C-1 to C-2
user interface controls, C-4 to C-7

R

RdWrt object, 9-7 to 9-8
Read method
parsing, 8-3
synchronous I/O, message-based communication, 9-10
ReadAsynch method
asynchronous I/O, message-based communication, 9-10
parsing, 8-3
ReadCounter method, 7-48
ReadMeasurement method, 7-48
Reconfigure method, 7-53
register-based communication, 9-11 to 9-13
high-level register accesses, 9-12
low-level register accesses, 9-13
moving blocks of data, 9-12
tutorial, 9-20 to 9-22
code development, 9-21 to 9-22
form design, 9-20 to 9-21
setting properties, 9-21
testing the program, 9-22
RemoveAll method, 8-6
Reset functions, 7-60
Reset method
AIPoint object, 7-7
AO control, 7-22
AOPoint object, 7-19, 7-20
asynchronous acquisition, 7-10
Counter control, 7-48
DO control, 7-40
Pulse control, 7-53
return codes, testing applications, 12-14 to 12-15
RTSI bus
data acquisition considerations, B-8
programming, B-8
run-time licenses, E-4

S

scan clocks, data acquisition
programming considerations, B-4
scan rate, 7-12
ScanClock object, 7-12 to 7-13
SCXI channel strings
data acquisition controls, 7-4 to 7-5
data acquisition programming
considerations, B-6 to B-7
SCXI hardware
configuring, B-2
data acquisition considerations,
B-7 to B-8
Serial (ASRL) object, 9-8
Serial control, 8-12 to 8-14
common properties, 8-2
CWSerial object, 8-12 to 8-13
methods and events, 8-13 to 8-14
asynchronous I/O, 8-14
synchronous I/O, 8-13
object hierarchy, 8-2
object hierarchy (figure), 8-12
overview, 1-1
parsing, 8-2 to 8-6
advanced features, 8-2 to 8-6
CWPattern object, 8-5
CWTask object, 8-4
CWToken object, 8-6
purpose and use, 8-1
questions and answers, C-12 to C-13
tutorial, 8-14 to 8-17
code development, 8-17
form design, 8-14 to 8-15
setting properties, 8-15 to 8-16
testing the program, 8-17
serial polling, with GPIB control, 8-9
ServiceRequest event, VISA control, 9-14
Set functions, 7-60
SetAttribute method, 11-12
setting properties. *See* properties, setting.
signal processing functions. *See* digital signal
processing and signal generation functions.
single point analog input. *See* AIPoint control.
single point analog output. *See*
AOPoint control.
single stepping, 12-18
SingleRead method
AIPoint object, 7-7
DIO, Port, and Line objects, 7-32 to 7-33
SingleWrite method
AOPoint object, 7-19 to 7-20
DIO, Port, and Line objects, 7-33
Slide control. *See* Knob and Slide controls.
software objects, 1-8
Source input, 7-45
Spectrum Meter application. *See*
Virtual Spectrum Meter application.
Start method
AO control, 7-21
asynchronous acquisition,
AI control, 7-10
Counter control, 7-48
DO control, 7-40
Pulse control, 7-53
StartCondition object
AI control, 7-13 to 7-14
AO control, 7-23 to 7-24
statistics functions
analysis of variance functions
(table), 10-7
curve fitting functions (table), 10-7
interpolation functions (table), 10-7
nonparametric statistics functions
(table), 10-7
probability distribution functions (table),
10-6 to 10-7
simple statistics (table), 10-6
tutorial, 10-13 to 10-17
developing code, 10-16 to 10-17
form design, 10-15
testing the program, 10-17

Statistics object, 6-6
step into mode, 12-18
step over mode, 12-18
stop condition modes, Virtual Oscilloscope application, 12-2 to 12-3
Stop method
    asynchronous acquisition, 7-10
    Counter control, 7-48
    Pulse control, 7-53
StopCondition object, 7-13 to 7-14
SwapBytes property
    GPIB control and Serial control, 8-2
    VISA control, 9-4
synchronous acquisition methods and events, 7-11 to 7-12
synchronous I/O
    GPIB control, 8-8
    message-based communication, 9-10
    Serial control, 8-13
system requirements, 1-3

T

technical support, F-1 to F-2
telephone and fax support numbers, F-2
testing applications. *See* debugging and testing applications.
Ticks object, 6-5
time domain signal processing functions (table), 10-9
TrackMode property
    Graph object, 6-14
    panning and zooming, 6-21
    Virtual Spectrum Meter application, 12-8 to 12-9
Trigger event, VISA control, 9-14 to 9-15
triggering instruments, with GPIB control, 8-9
tutorials
    AI control, 7-14 to 7-18
    AIPoint control, 7-14 to 7-18
    AOPoint control, 7-24 to 7-28
    Counter control, 7-55 to 7-59
    DataSocket control, 11-6 to 11-9
    DataSocket Server, 11-15 to 11-17
    DIO control, 7-41 to 7-45
    GPIB control, 8-9 to 8-11
    Graph and Button control tutorial, 6-22 to 6-25
    Knob and Slide controls, 6-8 to 6-11
    message-based communication, 9-17 to 9-19
    Numeric Edit Box control, 6-8 to 6-11
    Pulse control, 7-55 to 7-59
    register-based communication, 9-20 to 9-22
    Serial control, 8-14 to 8-17
    statistics functions, 10-13 to 10-17
two dimensional operations (table), 10-3

U

UI Controls. *See* User Interface controls.
Update method, 7-33 to 7-34
UpdateClock object
    AO control, 7-22 to 7-23
    DI control, 7-36
    DO control, 7-39 to 7-40
updating
    data, 11-5
    data target, 11-10
URL scheme, 11-3
URLs, for locating data source, 11-3
user interface, building. *See also* Graphical User Interface controls.
    Delphi applications, 5-4 to 5-6
    Visual Basic applications, 3-3 to 3-5
    Visual C++ applications, 4-4 to 4-5
User Interface controls.
    Button control, 6-12 to 6-13
    events, 6-13
    controls and associated styles (table), 6-2

Graph and Button control tutorial, 6-22 to 6-25
developing program code, 6-24 to 6-25
form design, 6-22 to 6-23
testing the program, 6-25
Graph control, 6-13 to 6-21
Axes collection, 6-20
Axis object, 6-20
chart methods, 6-16
Cursor object, 6-19
Cursors collection, 6-18 to 6-19
events, 6-21
Graph object, 6-14 to 6-16
hierarchy of (figure), 6-14
panning and zooming, 6-21
plot methods, 6-15
Plot object, 6-17 to 6-18
Plots collection, 6-16 to 6-18
PlotTemplate object, 6-18
tutorial, 6-22 to 6-25
Knob, Slide, and Numeric Edit Box control tutorial, 6-8 to 6-11
developing program code, 6-10 to 6-11
form design, 6-9
testing the program, 6-11
Knob and Slide controls, 6-3 to 6-7
Axis object, 6-4 to 6-5
events, 6-6 to 6-7
hierarchy of (figure), 6-3
Knob and Slide object, 6-3 to 6-4
Labels object, 6-5
Pointer object, 6-4
Pointers collection, 6-4
Statistics object, 6-6
Ticks object, 6-5
tutorial, 6-8 to 6-11
ValuePair object, 6-6
ValuePairs collection, 6-5 to 6-6
numeric edit box control, 6-7 to 6-8
events, 6-7 to 6-8
object hierarchy and common objects, 6-2
overview, 1-1
questions and answers, C-4 to C-7

V

value pairs, Virtual Oscilloscope application, 12-3 to 12-4
ValuePair object, 6-6
ValuePairs collection, 6-5 to 6-6
vector and matrix algebra functions (table), 10-5 to 10-6
Virtual Data Logger application, 12-9 to 12-12
file input/output, 12-12
graph axes formats, 12-11 to 12-12
multiple graph axes, 12-10 to 12-11
Virtual Oscilloscope application
data acquisition pretriggering, 12-3
data acquisition stop condition modes, 12-2 to 12-3
user interface value pairs, 12-3 to 12-4
Virtual Spectrum Meter application, 12-4 to 12-9
cursors, 12-7 to 12-8
DSP Analysis Library functions, 12-5 to 12-7
graph track mode, 12-8 to 12-9
VISA API, 9-1 to 9-2
advantages, 9-2
structure (figure), 9-1
VISA control, 9-1 to 9-22
common instrument control features, 9-4
error codes (table), D-17 to D-19
event handling with event queue, 9-15 to 9-17
checking events, 9-16
disabling events, 9-16 to 9-17
discarding events, 9-16
event types, 9-14 to 9-16

events, 9-14
GPIB object, 9-9
message-based communication, 9-9 to 9-13
asynchronous I/O, 9-10
synchronous I/O, 9-10
methods, 9-9 to 9-13
object hierarchy, 9-2 to 9-3
overview, 1-1
parsing, 9-5
purpose and use, 9-2
questions and answers, C-12 to C-13
RdWrt object, 9-7 to 9-8
register-based communication, 9-11 to 9-13
high-level register accesses, 9-12
low-level register accesses, 9-13
moving blocks of data, 9-12
Serial (ASRL) object, 9-8
tutorial
message-based communication, 9-17 to 9-19
register-based communication, 9-20 to 9-22
VISA object, 9-5 to 9-7
VXI object, 9-9
VISA object, 9-5 to 9-7
VISA Property Pages
General Page (figure), 9-6
RdWrt Page (figure), 9-7
Serial Page (figure), 9-8
VxiMemory Page (figure), 9-11
Visual Basic 4, A-1 to A-3
code completing lacking, A-3
creating default ComponentWorks project, A-3
menus and commands, A-1 to A-2
Object Browser, A-2 to A-3
Visual Basic applications, 3-1 to 3-12. *See also* Visual Basic 4.
automatic code completion feature, 3-11
building user interface, 3-3 to 3-5
default property sheet (figure), 1-11
developing event handler routines, 3-6 to 3-7
development procedure, 3-1 to 3-2
editing properties programmatically, 3-4 to 3-5
instrument driver DLLs, 3-7 to 3-8
loading ComponentWorks controls into toolbox, 3-2 to 3-3
Object Browser for building code, 3-8 to 3-10
online help for learning controls, 3-12
pasting code into programs, 3-10 to 3-11
property sheets, 3-3 to 3-4
questions about ComponentWorks, C-3
working with control methods, 3-5 to 3-6
Visual C++ 4.*x*, A-4 to A-6
adding ComponentWorks controls to toolbar, A-4 to A-5
building user interface and code, A-6
creating the application, A-4
Visual C++ applications, 4-1 to 4-10. *See also* Visual C++ 4.*x*.
adding ComponentWorks controls to toolbar, 4-4
building user interface, 4-4 to 4-5
creating applications, 4-1 to 4-3
events, 4-9 to 4-10
methods, 4-8 to 4-9
online help for learning controls, 4-10
programming with ComponentWorks controls, 4-5 to 4-6
properties, 4-6 to 4-8
VXI object, 9-9
VxiSignalProc event, VISA control, 9-15
VxiVmeInterrupt event, VISA control, 9-15

W

WaitOnEvent method, 9-16
warning events, 12-15 to 12-16
watch window, 12-18
waveform analog input. *See* AI control.
waveform analog output. *See* AO control.
waveform digital input, buffered. *See* DI control.
waveform tutorial, DataSocket control, 11-6 to 11-9
Web site support for ComponentWorks, 2-7
windows functions (table), 10-11
Write method
    AO control, 7-21
    synchronous I/O, message-based communication, 9-10
    UpdateClock object, 7-40, 7-41
WriteAsynch method, 9-10
writing data to data target, DataSocket control, 11-9 to 11-10

Z

zooming graphs, 6-21