ComponentWorks™

Getting Results with ComponentWorks™ Autotuning PID

National Instruments™

August 1998 Edition
Part Number 322064A-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

Important Information

Contents

About This Manual

Organization of This Manual xi
    Part I, Building ComponentWorks Applications xi
    Part II, Using the ComponentWorks Autotuning PID Control xii
    Part III, PID Algorithms and Control Strategies xiii
    Appendices, Glossary, and Index xiii
Conventions Used in This Manual xiii
Related Documentation xiv
Customer Communication xv

Chapter 1 Introduction to ComponentWorks Autotuning PID

What Is ComponentWorks Autotuning PID? 1-1
System Requirements 1-2
Installing ComponentWorks 1-3
    Installing From Floppy Disks 1-3
    Installed Files 1-4
About the ComponentWorks Controls 1-4
    Properties, Methods, and Events 1-4
    Object Hierarchy 1-5
    Collection Objects 1-6
Setting the Properties of an ActiveX Control 1-7
    Using Property Pages 1-7
    Changing Properties Programmatically 1-9
    Item Method 1-10
    Working with Control Methods 1-10
    Developing Event Handler Routines 1-11
Learning the Properties, Methods, and Events 1-11

Chapter 2 Getting Started with ComponentWorks

Explore the ComponentWorks Documentation 2-1
    Getting Results with ComponentWorks Autotuning PID Manual 2-1
    ComponentWorks Autotuning PID Online Reference 2-2
        Accessing the Online Reference 2-2
        Finding Specific Information 2-2

Become Familiar with the Examples Structure 2-3
Develop Your Application 2-3
Seek Information from Additional Sources 2-5

PART I
Building ComponentWorks Applications

Chapter 3
Building ComponentWorks Applications with Visual Basic

Developing Visual Basic Applications 3-1
Loading ComponentWorks Controls into the Toolbox 3-2
Building the User Interface Using ComponentWorks 3-2
Using Property Pages 3-3
Using Your Program to Edit Properties 3-4
Working with Control Methods 3-5
Developing Control Event Routines 3-5
Using the Object Browser to Build Code in Visual Basic 3-6
Pasting Code into Your Program 3-8
Adding Code Using Visual Basic Code Completion 3-9

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 4-3
Building the User Interface Using ComponentWorks 4-4
Programming with the ComponentWorks Controls 4-5
Using Properties 4-6
Using Methods 4-8
Using Events 4-8

Chapter 5
Building ComponentWorks Applications with Delphi

Running Delphi Examples 5-1
Developing Delphi Applications 5-1
Loading ComponentWorks Controls into the Component Palette 5-2
Building the User Interface 5-3
Placing Controls 5-4
Using Property Pages 5-4

Programming with ComponentWorks....5-6
Using Your Program to Edit Properties....5-6
Using Methods....5-7
Using Events....5-7

PART II
Using the ComponentWorks Autotuning PID Control

Chapter 6
Using the PID Control

PID Control....6-1
PID Object....6-2
Parameters Collection....6-2
Parameter Object....6-3
Autotune Object....6-3
Autotune Wizard....6-4
PID Events....6-4
LeadLag Control....6-4
Ramp Control....6-5
Tutorial: Using the PID Control....6-5
Part 1: Develop a PID System....6-5
Designing the Form....6-5
Setting the PID Properties....6-6
Developing the Code....6-7
Testing Your Program....6-8
Part 2: Autotune PID Parameters....6-8
Designing the Form....6-8
Setting PID Properties....6-9
Developing the Code....6-9
Testing Your Program....6-9

Chapter 7
PID Examples

AutotunePID....7-1
Cascade and Selector....7-1
General PID....7-2
LeadLag....7-2
PID with MIO Board....7-2
RampDemo....7-3
Tank Level....7-3

PART III
PID Algorithms and Control Strategies

Chapter 8
Algorithms
PID Algorithm..8-1
Gain Scheduling...8-4
Autotuning Algorithm..8-5
Tuning Formulas...8-6

Chapter 9
Designing Control Strategies
Setting Timing...9-2
Manual Tuning Techniques 9-2
Closed-Loop (Ultimate Gain) Tuning Procedure 9-3
Open-Loop (Step Test) Tuning Procedure 9-4

Appendices, Glossary, and Index

Appendix A
Error Codes

Appendix B
Distribution and Redistributable Files

Appendix C
Customer Communication

Glossary

Index

Figures and Tables

Figures

Figure 1-1. PID Control Object Hierarchy 1-6
Figure 1-2. Visual Basic Default Property Sheets 1-8
Figure 1-3. ComponentWorks Custom Property Pages 1-8

Figure 3-1. Visual Basic Property Pages 3-3
Figure 3-2. ComponentWorks Custom Property Pages in Visual Basic 3-4
Figure 3-3. Selecting Events in the Code Window 3-6
Figure 3-4. Viewing CWPID in the Object Browser 3-7
Figure 3-5. Viewing the Parameters Sub-Object in the Object Browser 3-8
Figure 3-6. Visual Basic 5 Code Completion 3-9

Figure 4-1. New Dialog Box 4-2
Figure 4-2. MFC AppWizard—Selecting a Dialog-Based Application 4-3
Figure 4-3. CWPID Control Property Pages in Visual C++ 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 4-9

Figure 5-1. Delphi Import ActiveX Control Dialog Box 5-2
Figure 5-2. ComponentWorks Controls on a Delphi Form 5-4
Figure 5-3. Delphi Object Inspector 5-5
Figure 5-4. ComponentWorks PID Control Property Pages in Delphi 5-5
Figure 5-5. Delphi Object Inspector Events Tab 5-7

Figure 6-1. PID Control Object Hierarchy 6-1
Figure 6-2. Simple PID Example Form 6-6
Figure 6-3. Testing the Simple PID Example 6-8

Figure 7-1. Resistor-Capacitor Network 7-2

Figure 8-1. Nonlinear Multiple for Integral Action (SP_{rng} = 100) 8-3
Figure 8-2. Process under PID Control with Setpoint Relay 8-5

Figure 9-1. Control Flowchart 9-1
Figure 9-2. Output and Process Variable Strip Chart 9-4

Tables

Table 2-1. Chapters about Specific Programming Environments 2-4

Table 8-1. Tuning Formula under P-only Control (fast) 8-6
Table 8-2. Tuning Formula under P-only Control (normal) 8-6
Table 8-3. Tuning Formula under P-only Control (slow) 8-6
Table 8-4. Tuning Formula under PI Control (fast) 8-6
Table 8-5. Tuning Formula under PI Control (normal).. 8-7
Table 8-6. Tuning Formula under PI Control (slow) 8-7

Table 9-1. Closed-Loop–Quarter-Decay Ratio Values 9-3
Table 9-2. Open-Loop–Quarter-Decay Ratio Values... 9-5

Table A-1. ComponentWorks Errors A-1

About This Manual

The *Getting Results with ComponentWorks Autotuning PID* manual contains the information you need to get started with the ComponentWorks Autotuning PID control. ComponentWorks adds the instrumentation-specific tools for designing control strategies 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 98/95 or Windows NT.

Organization of This Manual

The *Getting Results with ComponentWorks Autotuning PID* manual is organized as follows.

- Chapter 1, *Introduction to ComponentWorks Autotuning PID*, contains an overview of ComponentWorks, lists the ComponentWorks Autotuning PID system requirements, describes how to install the software, and presents basic information about ComponentWorks ActiveX controls.
- Chapter 2, *Getting Started with ComponentWorks*, describes approaches to help you get started using ComponentWorks Autotuning PID, depending on your application needs, your experience using ActiveX controls in your particular programming environment, and your specific goals in using ComponentWorks.

Part I, Building ComponentWorks Applications

This section describes how to use ActiveX controls in the most commonly used programming environments—Visual Basic, Visual C++, and Borland Delphi.

Part I, *Building ComponentWorks Applications*, contains the following chapters.

- Chapter 3, *Building ComponentWorks Applications with Visual Basic*, describes how you can use the ComponentWorks controls with Visual Basic; 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.

Part II, Using the ComponentWorks Autotuning PID Control

This section describes the ComponentWorks PID, LeadLag, and Ramp controls. These chapters contain overviews of the controls and their most commonly used properties, methods, and events; short code segments to illustrate programmatic control; and tutorials that lead you through building an application with the controls.

Part II, *Using the ComponentWorks Autotuning PID Control*, contains the following chapters.

- Chapter 6, *Using the PID Control*, describes the Autotuning PID control and includes a tutorial with step-by-step instructions for using the control. This chapter also introduces the LeadLag and Ramp controls.
- Chapter 7, *PID Examples*, describes examples included with the ComponentWorks Autotuning PID software.

Part III, *PID Algorithms and Control Strategies*

This section describes the Autotuning PID algorithms and basic control design systems. Part III, *PID Algorithms and Control Strategies*, contains the following chapters.

- Chapter 8, *Algorithms*, explains the PID algorithm, the Autotuning algorithm, and how these algorithms are applied to control systems.
- Chapter 9, *Designing Control Strategies*, describes how you can design and implement control strategies with the PID, LeadLag, and Ramp controls.

Appendices, Glossary, and Index

- Appendix A, *Error Codes*, lists the error codes returned by ComponentWorks.
- Appendix B, *Distribution and Redistributable Files*, contains information about ComponentWorks Autotuning PID redistributable files and distributing applications that use ComponentWorks controls.
- Appendix C, *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, and dialog box options.
bold italic	Bold italic text denotes a note.
<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, 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 document contains information you might find useful as you read this manual:

- ComponentWorks Autotuning PID Online Reference (available by selecting **Programs»National Instruments ComponentWorks» Autotuning PID»ComponentWorks PID Reference** from the Windows **Start** menu)

If you have purchased and installed one of the ComponentWorks development systems, you also have access to the following documentation:

- *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 C, *Customer Communication*, at the end of this manual.

Introduction to ComponentWorks Autotuning PID

This chapter contains an overview of the ComponentWorks Autotuning PID control, lists the ComponentWorks Autotuning PID system requirements, describes how to install the software, and presents introductory information about ComponentWorks ActiveX controls.

What Is ComponentWorks Autotuning PID?

ComponentWorks Autotuning PID is a collection of ActiveX controls for developing PID control applications 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»Autotuning PID»ComponentWorks PID Reference** from the Windows **Start** menu.

PID (Proportional-Integral-Derivative) is the most common control algorithm used in industry. PID is often used to control processes such as heating and cooling systems, fluid level monitoring, flow control, and pressure control. The system parameter being controlled is referred to as the *process variable* (for example, temperature, pressure, or flow rate). The operator must specify a *setpoint* or desired value for the process variable that is to be controlled. A PID controller determines a *controller output* value (for example, heater power or valve position). The controller output value is applied to the system which in turn affects the process variable and drives it toward the setpoint value.

With ComponentWorks Autotuning PID, you can develop control applications based on proportional-integral-derivative (PID) controllers:

- Proportional (P); proportional-integral (PI); proportional-derivative (PD); and proportional-integral-derivative (PID) algorithms
- Gain-scheduled PID
- PID autotuning
- Error-squared PID
- Lead-Lag compensation
- Setpoint ramp generation
- Multiloop cascade control
- Feedforward control
- Override (minimum/maximum selector) control
- Ratio/bias control

The ComponentWorks Autotuning PID package contains the following components:

- PID Control—ActiveX control for implementing PID process control algorithms in a physical system.
- LeadLag Control—ActiveX control for calculating the dynamic compensator in feedforward control schemes.
- Ramp Control—ActiveX control for generating setpoint ramps.

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 other applications that support them, including Visual C++ and Delphi.

System Requirements

To use the ComponentWorks Autotuning PID ActiveX controls, your computer must meet the following minimum requirements:

- Personal computer using at least a 33 MHz 80486 or higher microprocessor (National Instruments recommends a 90 MHz Pentium or higher microprocessor)
- Microsoft Windows 98/95 or Windows NT version 4.0
- VGA resolution (or higher) video adapter

- 32-bit ActiveX control container such as Visual Basic 4.0 or greater, Visual C++ 4.*x* or greater, or Delphi
- Minimum of 16 MB of memory
- Minimum of 10 MB of free hard disk space
- Microsoft-compatible mouse

Installing ComponentWorks

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

1. Insert the ComponentWorks Autotuning PID CD in the CD drive of your computer. From the CD startup screen, click **Install ComponentWorks Autotuning PID**. If the CD startup screen does not appear, use the Windows Explorer to run the `SETUP.EXE` program in the `\Setup` directory on the CD.
2. 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 if you are unsure about which settings to choose. 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, you can copy the files to floppy disks and install the controls from those disks, as described by the following steps.

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.

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: `cwpid.ocx`, `cwpid.dep`, `cwpid.hlp`, `cwpid.cnt`
- Example programs and applications

  Directory: `\ComponentWorks\Samples\...`
- Tutorial programs

  Directory: `\ComponentWorks\Tutorials-PID\...`
- Miscellaneous files

  Directory: `\ComponentWorks\`

Note ***You select the location of the*** `\ComponentWorks\...` ***directory during installation.***

About the ComponentWorks Controls

This section presents background information about the ComponentWorks ActiveX controls. Make sure you understand these concepts before continuing. You also should refer to your programming environment documentation for more information about using ActiveX controls in that 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 is affected by the current property values of the control.

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

☞ **Note** ***Use the 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»Autotuning PID»ComponentWorks PID Reference** ***from the Windows*** **Start** ***menu.***

Object Hierarchy

The three parts of an ActiveX control—properties, methods, and events—are stored in a *software object*. Because some ActiveX controls are very complex and contain many properties, 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 specific parts (properties) and functionality (methods and events) of the ActiveX control. The relationships among 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 a hierarchy is not limited.

Figure 1-1 shows the object hierarchy of the ComponentWorks Autotuning PID control. The PID object contains some of its own properties, such as `Name` and `Setpoint`. It also contains the `Parameters` property, which is a separate object. The Parameters object contains individual Parameter objects, each describing one PID parameter. Each Parameter object has properties, such as `IntegralTime` and `ProportionalGain`, while the Parameters collection object has a property `Count`. The Parameters collection object is a special type of object referred to as a collection, which is described in the *Collection Objects* section.

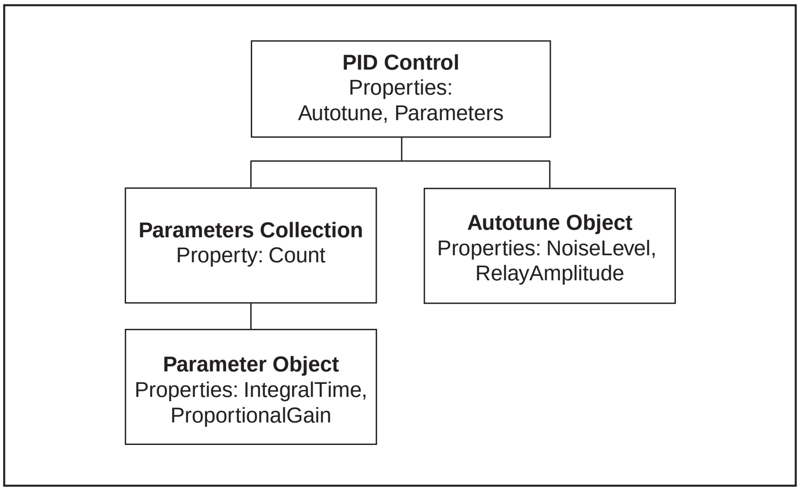


Figure 1-1. PID Control Object Hierarchy

Collection Objects

One object can contain several objects of the same type. For example, the PID object uses several Parameter objects, each representing one PID parameter. 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 parameters). 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. You can consider a collection 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 Parameter objects is referred to as Parameters. In the ComponentWorks software, the terms *object* and *collection* are rarely used, only the type names Parameter and Parameters are listed.

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 accessing 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

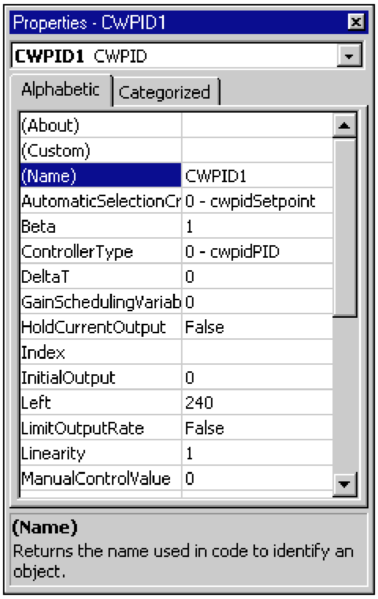
Property pages are common throughout the Windows 98/95 and Windows NT interface. When you want to change the appearance or options of a particular object, right click the object and select **Properties**. A property page or tabbed dialog box appears with a variety of properties that you can set for that 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 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.

Figure 1-2 shows the Visual Basic default property sheet for the PID control.



Properties - CWPID1

CWPID1 CWPID

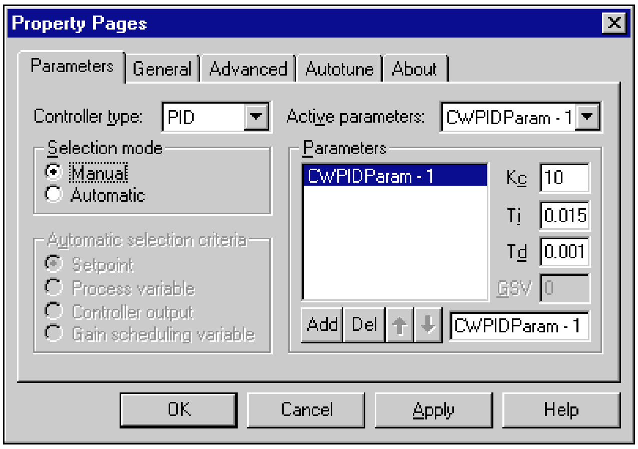
Alphabetic | Categorized

(About)	
(Custom)	
(Name)	CWPID1
AutomaticSelectionCr	0 - cwpidSetpoint
Beta	1
ControllerType	0 - cwpidPID
DeltaT	0
GainSchedulingVariab	0
HoldCurrentOutput	False
Index	
InitialOutput	0
Left	240
LimitOutputRate	False
Linearity	1
ManualControlValue	0

(Name)
Returns the name used in code to identify an object.

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. Figure 1-3 shows the custom property page for the Autotuning PID control.



Property Pages

Parameters | General | Advanced | Autotune | About

Controller type: PID

Active parameters: CWPIDParam - 1

Selection mode: [✓ Manual] Automatic

Automatic selection criteria: Setpoint, Process variable, Controller output, Gain scheduling variable

Parameters: CWPIDParam - 1

Kc 10

Ti 0.015

Td 0.001

GSV 0

Add | Del | CWPIDParam - 1

OK | Cancel | Apply | Help

Figure 1-3. ComponentWorks Custom Property Pages

Changing Properties Programmatically

You also can set or read the properties of your controls programmatically. For example, you can specify the type of controller—proportional-integral-derivative, proportional-derivative, or proportional—with the `ControllerType` property of the PID control.

Note ***The exact syntax for reading and writing property values depends on the programming language. Refer to the appropriate*** *Building ComponentWorks Applications* ***chapter for information about using ComponentWorks in your programming environment. In this discussion, code examples are written in 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 set the `ControllerType` property using the following line of code, where `cwpidPID` is a constant specifying the PID controller.

```
CWPID1.ControllerType = cwpidPID
```

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 PID control.

```
CWPID1.Autotune.ControlDesign = cwpidNormal
```

In the above code, `Autotune` is a property of the PID control and refers to an Autotune object. `ControlDesign` is one of several Autotune properties.

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

```
Print CWPID1.ControllerType
```

You can display the setpoint of the PID control in a Visual Basic text box with the following code.

```
Text1.Text = CWPID1.Setpoint
```

Item Method

To access an object or its properties in a collection, use the `Item` method on the collection object. For example, you can set the proportional gain of a PID control with the following code.

```
CWPID1.Parameters.Item(2).ProportionalGain= 10
```

The term `CWPID1.Parameters.Item(2)` refers to the second Parameter object in the Parameters collection of the PID object. The parameter of the `Item` method is an integer representing the (one-based) index of the object in the collection.

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

```
CWPID1.Parameters(2).ProportionalGain = 10
```

is programmatically equivalent to

```
CWPID1.Parameters.Item(2).ProportionalGain= 10
```

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.

Methods can have required and optional parameters in some programming environments, such as Visual Basic. You can omit these optional parameters if you want to use their default values. Other programming environments require all parameters to be passed explicitly.

Depending on your programming environment, parameters might be enclosed in parentheses. If the function or method is not assigned a return variable, Visual Basic does not use parentheses to pass parameters. When returning a value to a variable, enclose parameters in parentheses, as in the following example.

```
output = CWPID1.NextOutput(ProcessVariable)
```

Developing Event Handler Routines

After configuring 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 PID control has an `AutotuneComplete` event that *fires* (occurs) when the autotuning process has completed.

To develop the event routine code, most programming environments generate a skeleton function to handle each event. For information about generating these function skeletons, refer to the appropriate *Building ComponentWorks Applications* chapter. For example, the Visual Basic environment generates the following function skeleton into which you insert the functions to call when the `AutotuneComplete` event occurs.

```
Private Sub CWPID1_AutotuneComplete
  (ByVal NewParameterPosition As Long)

End Sub
```

In most cases, the event also returns some data to the event handler that can be used in your event handler routine.

Learning the Properties, Methods, and Events

The ComponentWorks PID online reference contains detailed information about each control and its associated properties, methods, and events. 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»Autotuning PID»ComponentWorks PID 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.

2

Getting Started with ComponentWorks

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

Explore 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 this manual to learn how to develop simple applications. It also contains information you can use in specific circumstances, such as debugging particular problems.

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

Getting Results with ComponentWorks Autotuning PID Manual

This manual contains three different parts:

- Part I, *Building ComponentWorks Applications*—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.

- Part II, *Using the ComponentWorks Autotuning PID Control*—These chapters describe the ComponentWorks PID, LeadLag, and Ramp controls. These chapters contain overviews of the controls and their most commonly used properties, methods, and events; short code segments to illustrate programmatic control; and tutorials that lead you through building an application with the controls.
- Part III, *PID Algorithms and Control Strategies*—These chapters describe the Autotuning PID algorithms and suggest different control strategies.

ComponentWorks Autotuning PID Online Reference

The ComponentWorks Autotuning PID Online Reference includes complete reference information for all controls—all properties, methods, and events for every control—as well as the text from this manual.

To use the online reference efficiently, you should understand the material presented in this manual about using ComponentWorks ActiveX controls. After going through this manual and its 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» Autotuning PID»ComponentWorks PID 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, `CWPID`). Property, method, and event names are identical to those used in the code.

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 Autotuning PID*.

Become Familiar with the Examples Structure

The examples installed with ComponentWorks Autotuning PID software show you how to use the control 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, organized by programming environment (`\Visual Basic`, `\Visual C++`, and so on), and grouped in the PID folder under each language. Within these directories, the examples are further subdivided by functionality.

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

Develop Your Application

Depending on your experience with your programming environment, ActiveX controls, and ComponentWorks, you can get started using ComponentWorks Autotuning PID 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 about ActiveX controls in Chapter 1, *Introduction to ComponentWorks Autotuning PID*, 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 (`www.natinst.com/support`) for current information about particular environments.

Table 2-1. Chapters about 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. Part II provides basic information about the PID control and describes its most commonly used properties, methods, and events. Chapter 6, *Using the PID Control*, also offers a tutorial to help you become more familiar with using the controls. The tutorial solution is installed with your software (`\ComponentWorks\Tutorials-PID`).

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 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. Generally, the examples avoid presenting complex operations on more than one control. To become familiar with a control, look at the example for that control. Then, you can combine different programming concepts from the different controls 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 differ from other examples; however, the values of the properties change the example significantly.

Seek 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 Autotuning PID* Appendices—The appendices include error descriptions and distribution information.
- ComponentWorks Autotuning PID Online Reference—The online reference includes complete reference documentation and text from the *Getting Results with ComponentWorks Autotuning PID* manual. If you cannot 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.

Part I

Building ComponentWorks Applications

This section describes 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.

Part I, *Building ComponentWorks Applications*, contains the following chapters.

- Chapter 3, *Building ComponentWorks Applications with Visual Basic*, describes how you can use the ComponentWorks controls with Visual Basic; 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.

3

Building ComponentWorks Applications with Visual Basic

This chapter describes how you can use the ComponentWorks controls with Visual Basic; 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.

Note ***The descriptions and figures in this chapter apply specifically to the Visual Basic 5 environment.***

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 the control while editing your application and the Visual Basic code editor opens to a default event handler routine.

Loading ComponentWorks Controls into the Toolbox

Before building an application using ComponentWorks controls, you must add them to the Visual Basic toolbox.

1. In a new Visual Basic project, right click the toolbox and select **Components**.
2. Place a checkmark in the box next to **National Instruments CW PID**.

   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 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.

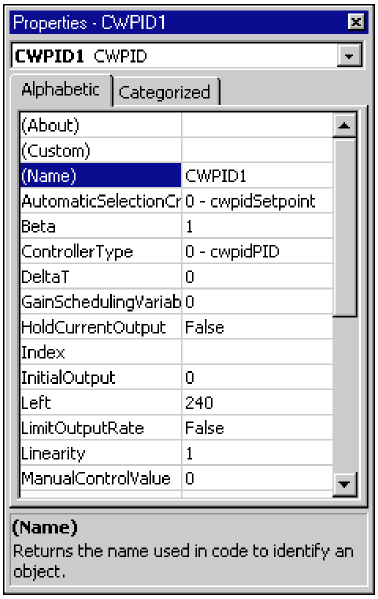
Building the User Interface Using ComponentWorks

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 creating the control, you can move it using the mouse. To move a control, click and hold the mouse on the control and drag the control to the desired location. You cannot resize the PID, LeadLag, or Ramp icons after placing them on the form. They also are not visible during 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 Pages

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.



(About)	
(Custom)	
(Name)	CWPID1
AutomaticSelectionCr	0 - cwpidSetpoint
Beta	1
ControllerType	0 - cwpidPID
DeltaT	0
GainSchedulingVariab	0
HoldCurrentOutput	False
Index	
InitialOutput	0
Left	240
LimitOutputRate	False
Linearity	1
ManualControlValue	0

(Name)
Returns the name used in code to identify an object.

Figure 3-1. Visual Basic Property Pages

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

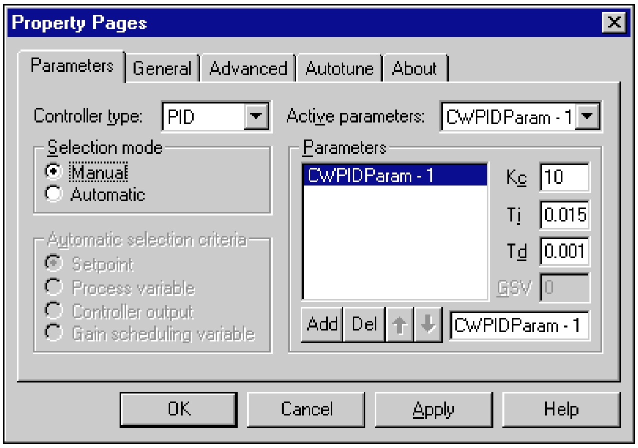


Figure 3-2. ComponentWorks Custom Property Pages in Visual Basic

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, you can set the `Setpoint` property of a PID control using the following line of code, where `CWPID1` is the default name of the PID control.

```
CWPID1.Setpoint = 50
```

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 PID control.

```
CWPID1.Parameters(1).ProportionalGain = 10
```

In the above code, `Parameter` is a property of the PID control and refers to a Parameter object. `ProportionalGain` is one of several Parameter properties.

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

```
Print CWPID1.Setpoint
```

You can display the proportional gain of a parameter in a Visual Basic text box with the following code.

```
Text1.Text = CWPID1.Parameters(1).ProportionalGain
```

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 `Reset` method on the PID control.

```
CWPID1.Reset
```

Methods can have parameters that you pass to the method and return values that pass information back to your program. 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. If you assign the return value of a method to a return variable, enclose the parameters in parentheses.

```
Output = CWPID1.NextOutput(ProcessVariable)
```

Developing Control Event Routines

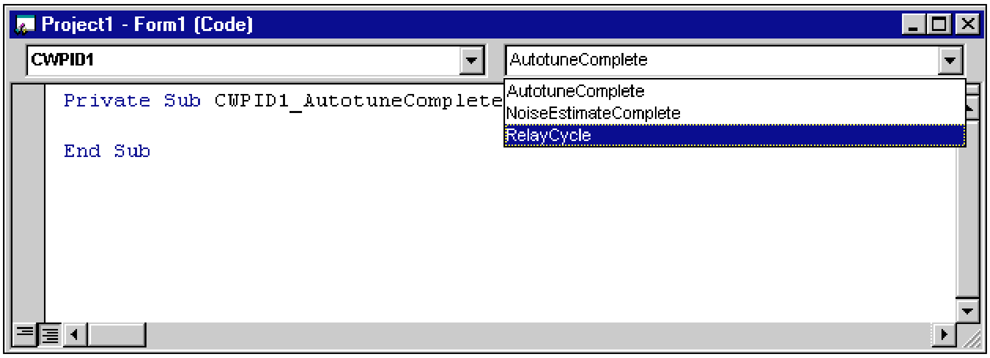
After configuring controls in the forms editor, you can write Visual Basic code to respond to events on the controls. The controls generate these events in response to user interactions with the controls or in response to some other occurrence in the control.

To develop the event handler routine code for an ActiveX control in Visual Basic, double click 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 PID control. This event routine (`AutotuneComplete`) is called when the autotuning process has completed.

```
Private Sub CWPID1_AutotuneComplete
   (ByVal NewParameterPosition As Long)
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 Figure 3-3.



```
Private Sub CWPID1_AutotuneComplete

End Sub
```

Figure 3-3. Selecting Events in the Code Window

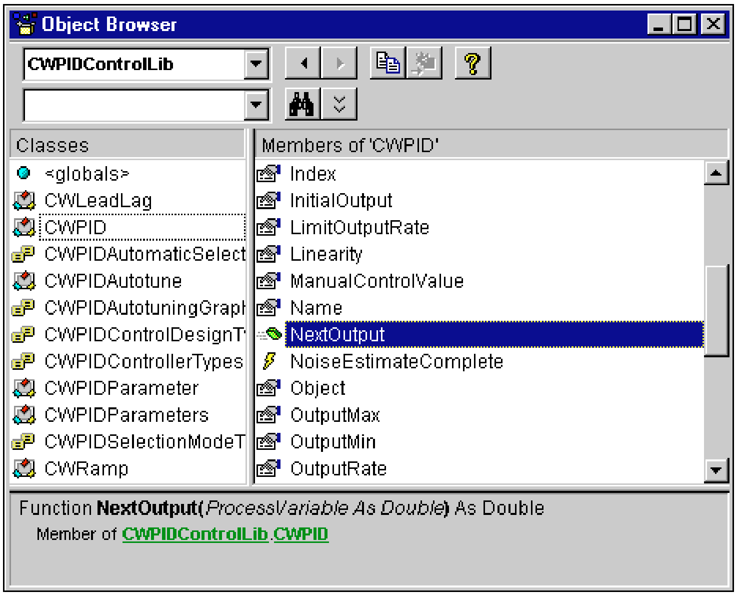
Use the left pull-down menu in the code window to change to another control without going back to the form window.

Using the 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 while creating your program. The Object Browser displays a detailed list of the available properties, methods, and events for a particular control. 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. 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 PID control file selected in the Object Browser. The Classes list shows the PID control, LeadLag control, Ramp control, 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.

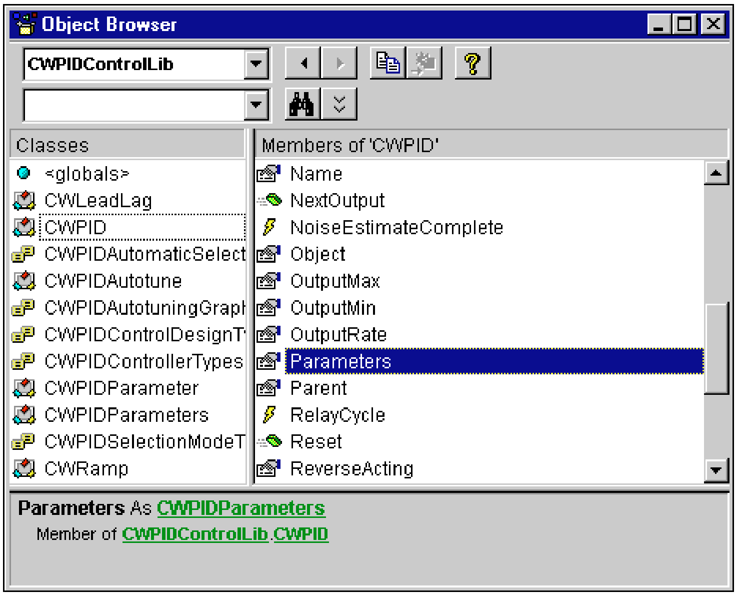


Function **NextOutput(***ProcessVariable As Double***)** As Double
Member of CWPIDControlLib.CWPID

Figure 3-4. Viewing CWPID 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 CWPID control is selected from the Classes list. For this control, the `NextOutput` 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 CWPID control selected in the Classes list and its `Parameters` property selected in the Members list.



Parameters As CWPIDParameters
Member of CWPIDControlLib.CWPID

Figure 3-5. Viewing the Parameters Sub-Object in the Object Browser

The Parameters object on the CWPID control is a separate object, so the description at the bottom of the dialog window lists the Parameters property as CWPIDParameters. CWPIDParameters is the type name of the Parameters collection object, and you can select CWPIDParameters 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 CWPID 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>.

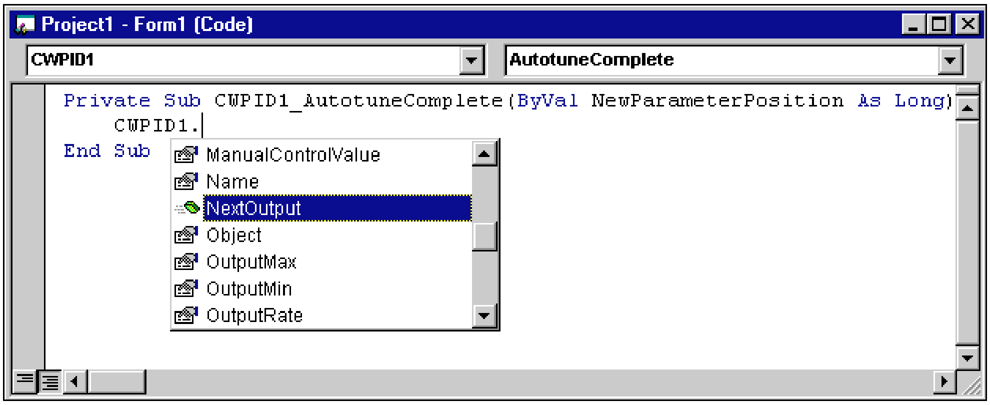
Use this method repeatedly to build a more complex reference to a property of a lower-level object in the object hierarchy. For example, you can create a reference to

```
CWPID1.Parameters.Item(1).ProportionalGain
```

by typing in the name of the control (`CWPID1`) and then using the Object Browser to add each section of the property reference.

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.



```
Private Sub CWPID1_AutotuneComplete(ByVal NewParameterPosition As Long)
    CWPID1.
End Sub
```

Figure 3-6. Visual Basic 5 Code Completion

You can select from the list of 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.

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.

Note ***The descriptions and figures in this chapter apply specifically to the Visual C++ 5 environment.***

Developing Visual C++ Applications

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

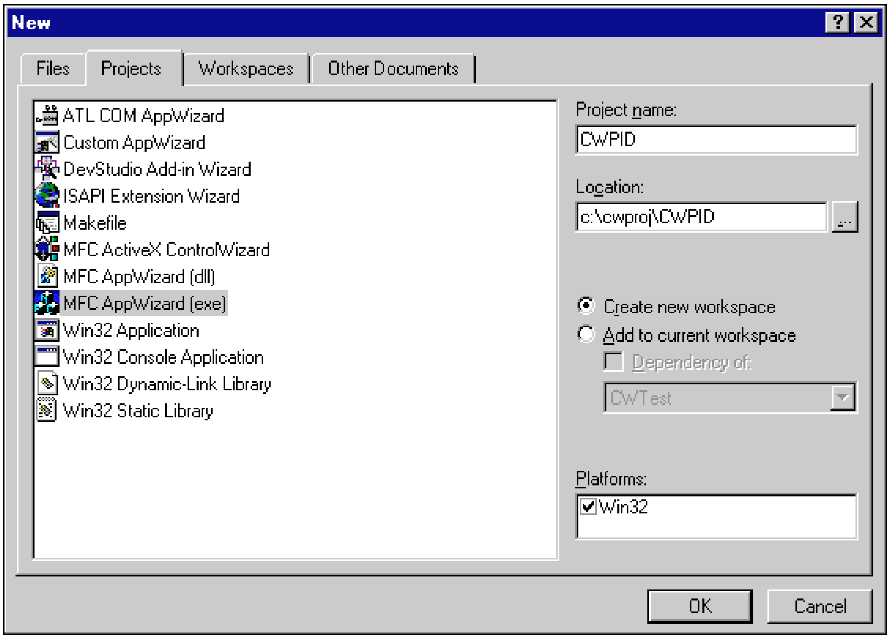
1. Create a new workspace or project in Visual C++.
2. To create a project compatible with the ComponentWorks ActiveX controls, use the Visual C++ MFC AppWizard to create a skeleton project and program.
3. After building the skeleton project, add the ComponentWorks controls to the controls toolbar. From 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).
2. On the **Projects** tab, select the MFC AppWizard (exe) and enter the project name and the directory.

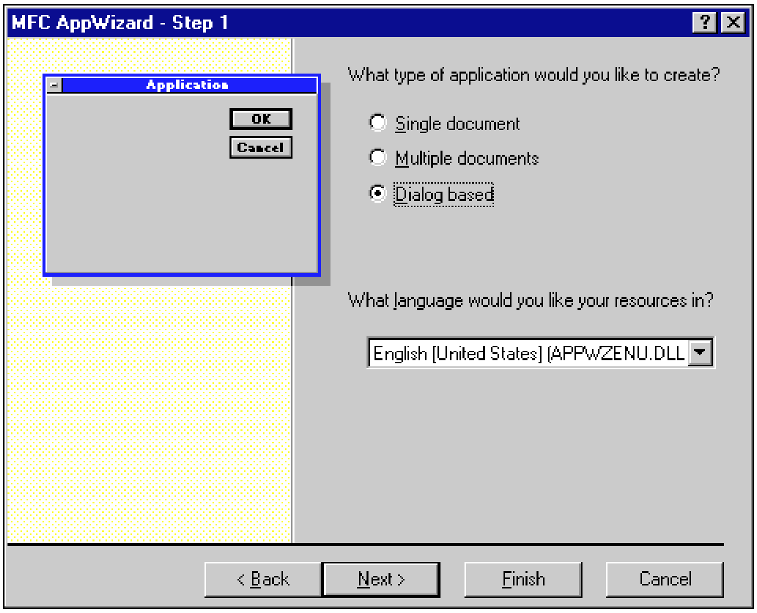


Project name: CWPID
Location: c:\cwproj\CWPID
[✓ Create new workspace]
Add to current workspace
Dependency of: CWTest
Platforms: [✓ Win32]

Figure 4-1. New Dialog Box

3. Click **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.



What type of application would you like to create?

( ) Single document
( ) Multiple documents
(●) Dialog based

What language would you like your resources in?

English [United States] (APPWZENU.DLL

Figure 4-2. MFC AppWizard—Selecting a Dialog-Based Application

5. Click 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 is automatically generated 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 one of the Dialog entries.
4. If necessary, right click 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 Registered ActiveX Controls.
2. Select the ComponentWorks PID control and click the **Insert** button.
3. Click **OK** in the following dialog windows. Repeat Steps 1 through 3 to add other controls.
4. When you have inserted the controls, click **Close** in the Components and Controls Gallery.

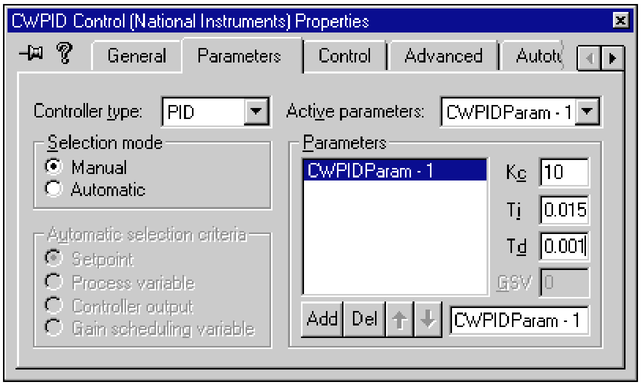
Building the User Interface Using ComponentWorks

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 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 CWPID control property pages.



Controller type: PID
Active parameters: CWPIDParam - 1
Selection mode: Manual [✓] / Automatic
Automatic selection criteria: Setpoint / Process variable / Controller output / Gain scheduling variable
Parameters: CWPIDParam - 1
Kc 10
Ti 0.015
Td 0.001
GSV 0
CWPIDParam - 1

Figure 4-3. CWPID Control Property Pages in Visual C++

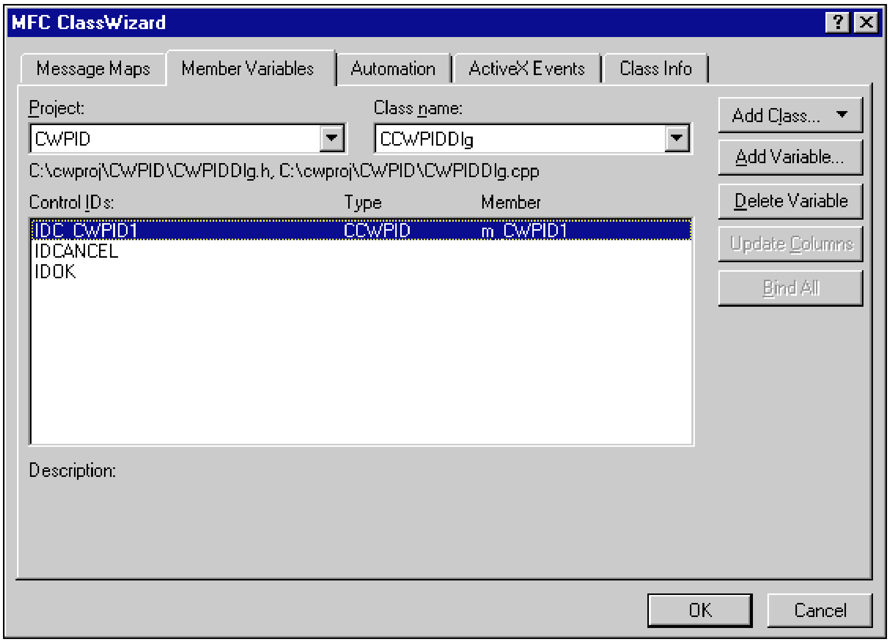
Programming with the ComponentWorks Controls

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

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 the control and select **ClassWizard**. In the **MFC Class Wizard** window, activate the **Member Variables** tab, as shown in Figure 4-4.

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 the PID control.



Project: CWPID

Class name: CCWPIDDlg

C:\cwproj\CWPID\CWPIDDlg.h, C:\cwproj\CWPID\CWPIDDlg.cpp

Control IDs:	Type	Member
IDC_CWPID1	CCWPID	m_CWPID1
IDCANCEL		
IDOK		

Figure 4-4. MFC ClassWizard—Member Variable Tab

Using Properties

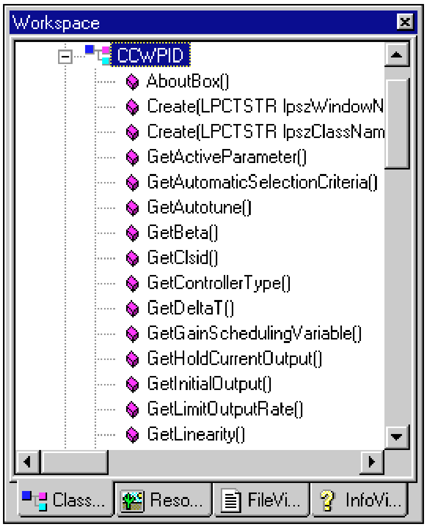
Unlike Visual Basic, you do not read or set the properties of ComponentWorks controls directly in Visual C++. Instead, the wrapper class of each control contains functions to read and write the value of each property. These functions are named starting with either `Get` or `Set` followed by the name of the property. For example, to set the `Setpoint` property of a PID control, use the `SetSetpoint` function of the wrapper class for the PID control. In the source code, the function call is preceded by the member variable name of the control to which it applies.

```
m_CWPID1.SetSetpoint(50);
```

Some values passed to properties need to be of variant type. Convert the value passed to the property to a variant using `COleVariant()`. For example, set the `Item` property of the CWParameters object.

```
m_CWPID1.GetParameters().Item(COleVariant(1.0));
```

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 PID object as listed in the Workspace. These are created automatically when you add a control to the Controls toolbar in you project.



CCWPID
AboutBox()
Create(LPCTSTR lpszWindowN
Create(LPCTSTR lpszClassNam
GetActiveParameter()
GetAutomaticSelectionCriteria()
GetAutotune()
GetBeta()
GetClsid()
GetControllerType()
GetDeltaT()
GetGainSchedulingVariable()
GetHoldCurrentOutput()
GetInitialOutput()
GetLimitOutputRate()
GetLinearity()

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 set the `RelayAmplitude` on the PID control.

```
#include "cwpidautotune.h"
CCWPIDAutotune Autotune;
Autotune = m_CWPID1.GetAutotune();
Autotune.SetRelayAmplitude(10);
```

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

```
#include "cwpidautotune.h"
m_CWPID1.GetAutotune().SetRelayAmplitude(10);
```

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 proportional gain of the first Parameter object on a PID control, use the following code.

```
#include "cwpidparameters.h"
#include "cwpidparameter.h"
m_CWPID1.GetParameters().Item(COleVariant(1.0)).
   SetProportionalGain(15);
```

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_CWPID1.Reset();
```

Some 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 `Remove` method of the Parameters object requires a scalar value as variant.

```
m_CWPID1.GetParameters().Remove(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, to call `StartAutotune` on the Autotune object, use the following line of code.

```
#include "cwpidautotune.h"
m_CWPID1.GetAutotune().StartAutotune(10);
```

Using Events

After placing a control on your form, you can start defining event handler functions for the control in your code. Controls generate events automatically at run time when they respond to specific conditions.

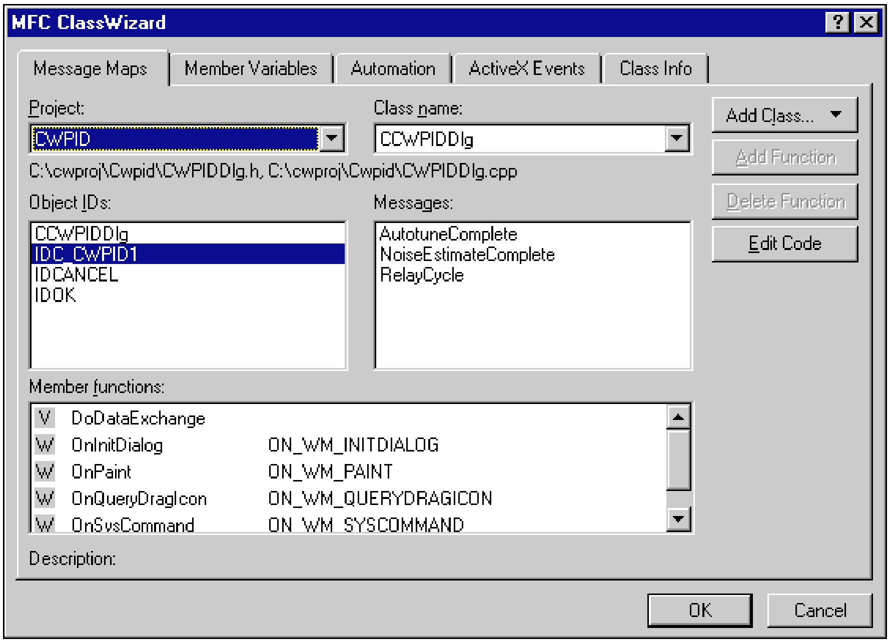
Use the following procedure to create an event handler.

1. Right click 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.)

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 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 `AutotuneComplete` event of the PID control.

```
void CCWPIDDlg::OnAutotuneCompleteCwpid1(long
  NewParameterPosition)
{
    m_CWPID1.SetActiveParameter(m_CWPID1.GetParameters().
      Item(COleVariant(NewParameterPosition)));
}
```



Project:
CWPID

Class name:
CCWPIDDlg

C:\cwproj\Cwpid\CWPIDDlg.h, C:\cwproj\Cwpid\CWPIDDlg.cpp

Object IDs:
CCWPIDDlg
IDC_CWPID1
IDCANCEL
IDOK

Messages:
AutotuneComplete
NoiseEstimateComplete
RelayCycle

Member functions:

V	DoDataExchange	
W	OnInitDialog	ON_WM_INITDIALOG
W	OnPaint	ON_WM_PAINT
W	OnQueryDragIcon	ON_WM_QUERYDRAGICON
W	OnSysCommand	ON_WM_SYSCOMMAND

Description:

Figure 4-6. Event Handler

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.

Note ***The descriptions and figures in this chapter apply specifically to the Delphi 3 environment. 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 controls into the Delphi environment. See the section on *Loading ComponentWorks Controls into the Component Palette* for more information about loading the controls.

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.

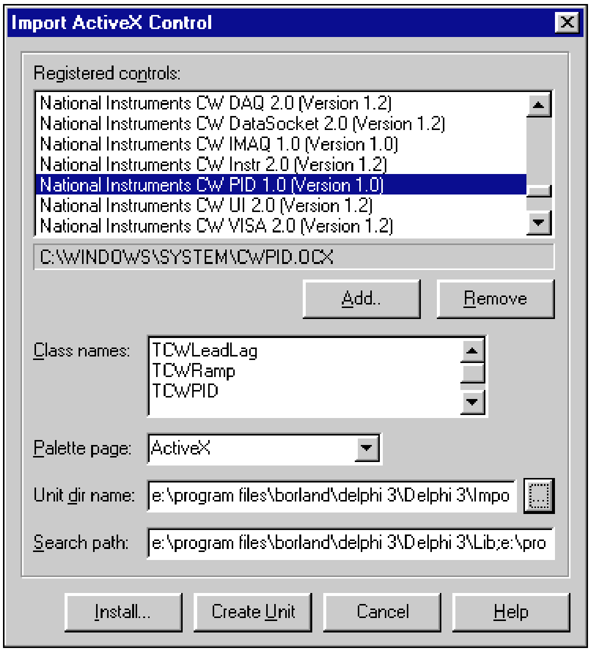
Loading ComponentWorks Controls into the Component Palette

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.


Import ActiveX Control

Registered controls:
National Instruments CW DAQ 2.0 (Version 1.2)
National Instruments CW DataSocket 2.0 (Version 1.2)
National Instruments CW IMAQ 1.0 (Version 1.0)
National Instruments CW Instr 2.0 (Version 1.2)
National Instruments CW PID 1.0 (Version 1.0)
National Instruments CW UI 2.0 (Version 1.2)
National Instruments CW VISA 2.0 (Version 1.2)
C:\WINDOWS\SYSTEM\CWPID.OCX

Class names: TCWLeadLag, TCWRamp, TCWPID
Palette page: ActiveX
Unit dir name: e:\program files\borland\delphi 3\Delphi 3\Impo
Search path: e:\program files\borland\delphi 3\Delphi 3\Lib;e:\pro

Figure 5-1. Delphi Import ActiveX Control Dialog Box

2. Select **National Instruments CW PID** to add the PID controls to the Component palette.
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 **OK** to add the controls to the Delphi user's components package.
5. In the following dialog, click **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 the **Add** button.
   b. Select the **Import ActiveX** tab.
   c. Select the ActiveX control you want to add.
   d. Click **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. 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 controls to customize the interface, as in Figure 5-2. After you place the controls, you can change their default property values by using the default property sheet (Object Inspector) and custom property sheets.

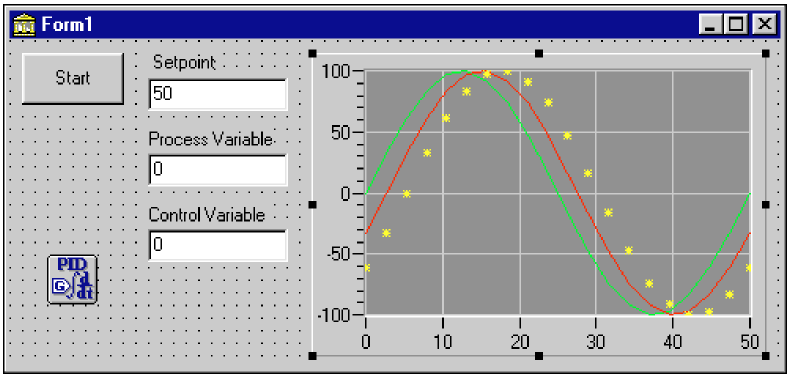
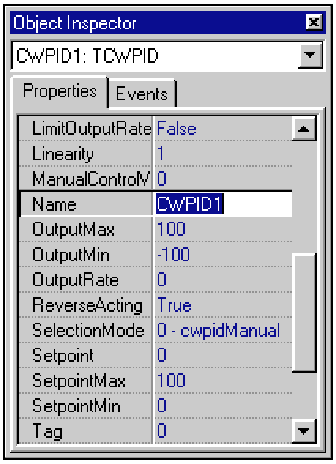


Figure 5-2. ComponentWorks Controls on a Delphi Form

Using Property Pages

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.



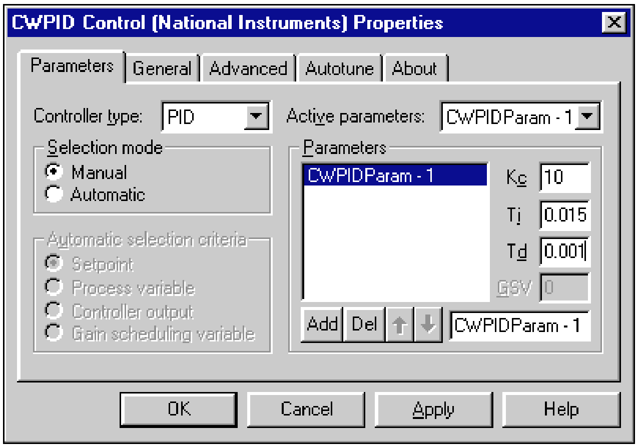
Object Inspector

CWPID1: TCWPID

LimitOutputRate	False
Linearity	1
ManualControlV	0
Name	CWPID1
OutputMax	100
OutputMin	-100
OutputRate	0
ReverseActing	True
SelectionMode	0 - cwpidManual
Setpoint	0
SetpointMax	100
SetpointMin	0
Tag	0

Figure 5-3. Delphi Object Inspector

To open the custom property pages of a control, double click the control or right click the control and select **Properties**. You can edit most control properties from the custom property pages. The following figure shows the ComponentWorks PID control property page.



CWPID Control (National Instruments) Properties

Controller type: PID

Active parameters: CWPIDParam - 1

Selection mode: [✓ Manual] Automatic

Automatic selection criteria: Setpoint, Process variable, Controller output, Gain scheduling variable

Parameters: CWPIDParam - 1

Kc 10

Ti 0.015

Td 0.001

GSV 0

Figure 5-4. ComponentWorks PID Control Property Pages in Delphi

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.

The syntax for setting the `Value` property in Delphi is

```
name.property := new_value;
```

For example, you can set the `Setpoint` property of a PID control using the following line of code, where `CWPID1` is the default name of the PID control.

```
CWPID1.Setpoint := 50;
```

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 PID control. `Autotune` is both a property of the PID control and an object itself. `RelayAmplitude` is a property of the Autotune object. As an object of the PID control, Autotune itself has several additional properties.

```
CWPID1.Autotune.RelayAmplitude := 2;
```

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

```
Edit1.Text := CWPID1.Autotune.RelayAmplitude;
```

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.

```
CWPID1.Parameters.Item(1).ProportionalGain := 10;
```

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.

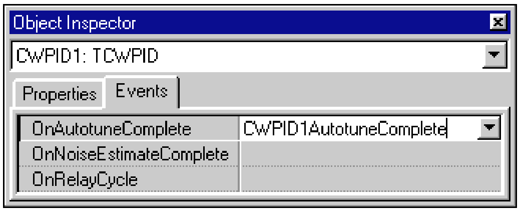
```
CWPID1.Reset;
```

Some methods require parameters. 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.

Using Events

Use event handler routines in your source code to respond to and process events generated by the ComponentWorks controls. Events are generated by user interaction with an object 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 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.



OnAutotuneComplete	CWPID1AutotuneComplete
OnNoiseEstimateComplete	
OnRelayCycle	

Figure 5-5. Delphi Object Inspector Events Tab

To specify your own event handler name, click 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. The following is an example of the event handler function for the `AutotuneComplete` event of the PID control.

```
procedure TForm1.CWPID1AutotuneComplete(Sender:
   TObject;NewParameterPosition: Integer);
begin
end;
```

Part II

Using the ComponentWorks Autotuning PID Control

This section describes the ComponentWorks PID, LeadLag, and Ramp controls. These chapters contain overviews of the controls and their most commonly used properties, methods, and events; short code segments to illustrate programmatic control; and tutorials that lead you through building an application with the controls.

Part II, *Using the ComponentWorks Autotuning PID Control*, contains the following chapters.

- Chapter 6, *Using the PID Control*, describes the Autotuning PID control and includes a tutorial with step-by-step instructions for using the control. This chapter also introduces the LeadLag and Ramp controls.
- Chapter 7, *PID Examples*, describes examples included with the ComponentWorks PID software.

6 Using the PID Control

This chapter describes the Autotuning PID control and includes a tutorial with step-by-step instructions for using the control. This chapter also introduces the LeadLag and Ramp controls.

You can find additional information about the PID control in the ComponentWorks Autotuning PID Online Reference, available by selecting **Programs»National Instruments ComponentWorks» Autotuning PID»ComponentWorks PID Reference** from the Windows **Start** menu.

PID Control

The PID control implements the PID algorithm, which you can find information about in Chapter 8, *Algorithms*. The PID control, CWPID, is built from a hierarchy of object, as illustrated in Figure 6-1.

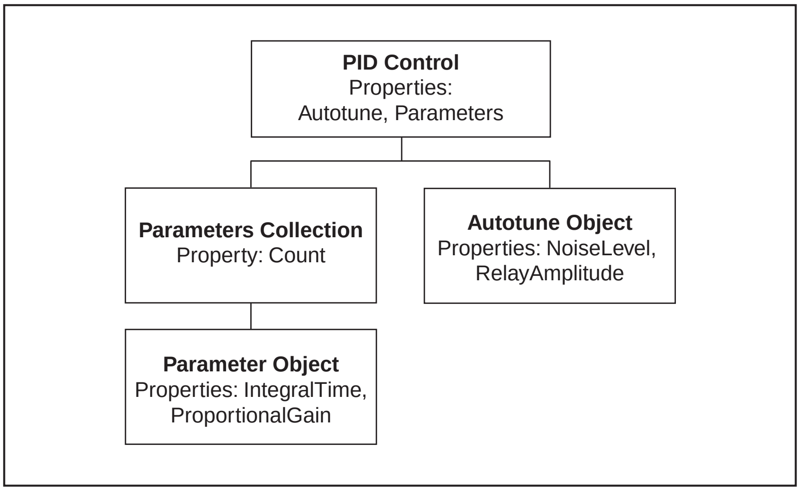


Figure 6-1. PID Control Object Hierarchy

PID Object

The PID object contains several properties and methods you can use to configure the PID control, and you can set properties in the property pages during design and change them programmatically at run time. You need to set the proportional gain, derivative time, and integral time in the property pages before run time.

`NextOutput` returns the next output from the PID algorithm given the controls current settings and a process variable.

```
Output = CWPID1.NextOutput(ProcessVariable)
```

You can have the object automatically determine the amount of time between successive calls to the control, or specify it yourself using the `DeltaT` property. If you use the built-in time keeper, the object calculates new timing information at each call, measuring the time since the last call and using that difference in its calculations.

Note ***If you use the built-in timer, remember that tick timer resolution is limited to 1 ms on Windows 95 and NT. Do not try to run the PID methods faster than 10 Hz when using the built-in timer.***

Parameters Collection

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

```
NumParameters = CWPID1.Parameters.Count
```

Use the `Add`, `Remove`, and `RemoveAll` methods to programmatically change the number of Parameter objects in the PID control. When calling the `Remove` method, include the index of the Parameter object to be removed.

```
CWPID1.Parameters.Add "param1", 10.0, 0.015, 0.001, 0
CWPID1.Parameters.Remove 3
```

Use the `Item` method of the Parameters collection to access a specific Parameter object in the collection.

Parameter Object

The Parameter object represents an individual parameter in the PID control. The Parameter object contains a number of different properties that you can use to determine the next output of the control.

You can set `IntegralTime`, `ProportionalGain`, and `DerivativeTime` as shown in the following code.

```
CWPID1.Parameters.Item(1).ProportionalGain = 10.0
CWPID1.Parameters.Item(1).Integraltime = 0.015
CWPID1.Parameters.Item(1).DerivativeTime = 0.001
```

Autotune Object

The Autotune object has four properties needed to autotune the PID parameters. For more information about autotuning or the autotuning algorithm, refer to Chapter 8, *Algorithms*.

- `NoiseLevel`—Amount of noise in the system.
- `RelayAmplitude`—Amplitude the Autotune object should use when modifying the setpoint during the relay process.
- `ControlDesign`—Design of the controller for which the Autotune object should calculate new PID parameters. Valid values include Fast, Normal, and Slow.
- `ControlType`—Type of controller for which the Autotune object should calculate new PID parameters. Valid values include PID, PI, and P.

Several methods can help you autotune PID parameters:

- `ShowDialog`—Displays the Autotune Wizard that steps you through the autotune process. Refer to *Autotune Wizard* for more information.
- `StartNoiseEstimate`—Begins a process that determines the amount of noise in the system.
- `StartRelay`—Begins the relay process and stabilizes it. For more information about the relay process, see Chapter 8, *Algorithms*.
- `StartAutotune`—Tunes the PID parameters.

Note ***You must call the*** `StartRelay` ***method before you can call the*** `StartAutotune` ***method.***

Autotune Wizard

The Autotune Wizard performs the following steps to guide you through the following autotuning process, which you can cancel anytime without affecting the current control state.

1. Determine the amount of noise in the system. If you know the amount of noise, you can specify it, and the wizard will not calculate it.
2. Begin the relay process. The wizard prompts you for information needed to autotune the PID parameters. After you enter this information, the wizard starts the relay process.
3. Autotune the PID parameters. When you specify that the relay process is stabilized, the wizard begins autotuning the PID parameters.
4. Display the new PID parameters. At this point, you can accept the new PID parameters or cancel the process.

PID Events

To signal when the noise estimate and autotuning process has finished, the CWPID object generates two events: `AutotuningComplete` and `NoiseEstimateComplete`. The control also generates an event every time the relay process completes a cycle.

LeadLag Control

The LeadLag control, CWLeadLag, calculates the dynamic compensator in feedforward control schemes. Properties of CWLeadLag include `Gain`, `LagTime`, and `LeadTime`. The `NextOutput` method returns the next value given the current state of the control.

You can have the object automatically determine the amount of time between successive calls to the control, or specify it yourself using the `DeltaT` property. If you use the built-in time keeper, the object calculates new timing information at each call, measuring the time since the last call and using that difference in its calculations.

Note ***If you use the built-in timer, remember that tick timer resolution is limited to 1 ms on Windows 95 and NT. Do not try to run the PID methods faster than 10 Hz when using the built-in timer.***

Ramp Control

The Ramp control, CWRamp, generates a setpoint ramp. Use the `Rate` property to determine how fast the value is ramped to the `Setpoint` property.

You can have the object automatically determine the amount of time between successive calls to the control, or specify it yourself using the `DeltaT` property. If you use the built-in time keeper, the object calculates new timing information at each call, measuring the time since the last call and using that difference in its calculations.

Note ***If you use the built-in timer, remember that tick timer resolution is limited to 1 ms on Windows 95 and NT. Do not try to run the PID methods faster than 10 Hz when using the built-in timer.***

Tutorial: Using the PID Control

This tutorial shows you how to configure the PID control and autotune the PID parameters. This tutorial is divided into two parts:

- Develop a PID system
- Autotune PID parameters

Although the code and examples in the tutorial use Visual Basic syntax, you can apply the concepts and implement the steps in any programming environment. Remember to adjust all code to your specific programming language.

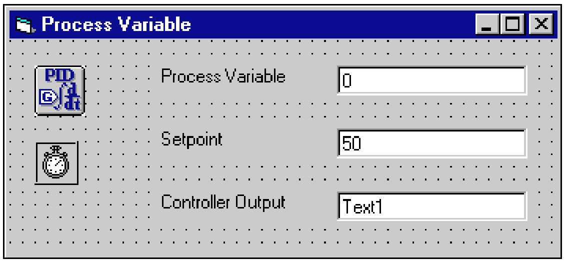
Part 1: Develop a PID System

In the first part of this tutorial, you use a CWPID object to control a known process.

Designing the Form

1. Open a new project and form. In Visual C++, select a dialog-based application and name your project `SimplePIDExample`.
2. Load the ComponentWorks PID control into your programming environment.
3. Place a CWPID control on the form. Keep its default name, `CWPID1`. You configure its properties in the next section, *Setting the PID Properties*.

4. Place three Visual Basic TextBoxes on the form and name them `ProcessVariable`, `Setpoint`, and `ControllerOutput`. Add a corresponding label to each text box.
5. Set `ProcessVariable` to `0` and `Setpoint` to `50`.
6. Place a Visual Basic Timer control on the form and set its `Interval` to `100` ms.



Process Variable	0
Setpoint	50
Controller Output	Text1

Figure 6-2. Simple PID Example Form

Setting the PID Properties

You normally configure the default property values of the different controls before you develop your program code. When using the PID control, you will set most properties, if not all, during design and will not change them during program execution. Use this program to start and stop the PID process only. If necessary, you can edit the properties of the PID control at run time.

1. Open the custom property pages for the CWPID control on the form by right clicking on the control and selecting **Properties**.
2. On the Parameters page, click the **Add** button in the Parameters group, and set the following parameters.

Proportional Gain (Kc)	10
Integral Time (Ki)	0.015
Derivative Time (Kd)	0.001

3. On the General page, change the `Setpoint` value to `50`.

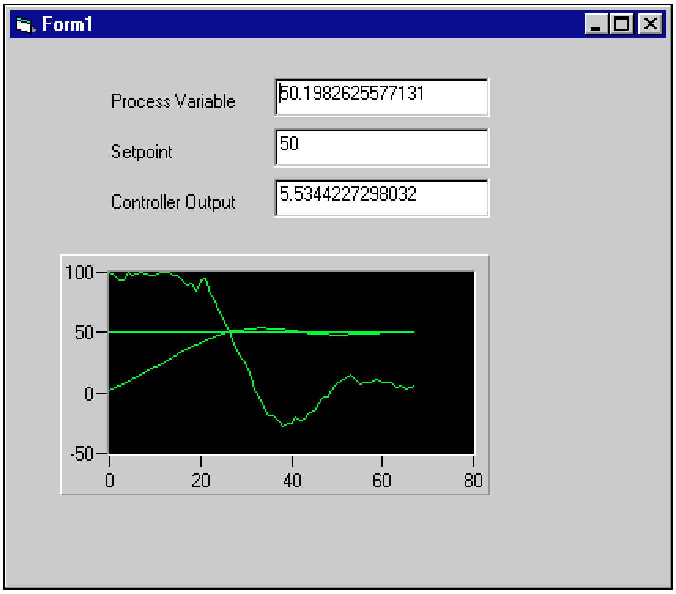
Developing the Code

In the following steps, you add the code to apply the PID algorithm. Normally, the PID control calculates a new output (controller output) that you apply to your physical system. In this tutorial, you simulate a real process by calculating a new process variable using the controller output and a simple formula. In real systems the process variable is measured from the physical system after applying the controller output.

1. Create an event handler for the `Change` event of the Setpoint textbox. Add the following code to the event handler to update the setpoint when it changes.
   ```
   CWPID1.Setpoint = Val(Setpoint.Text)
   ```
2. Create the event handler for the `Timer` event of the Timer control. In the `Timer` event, calculate the next controller output using the PID control and current process variable. Then calculate the corresponding process variable using a simulation formula. Add the following code to the event handler.
   ```
   ' Calculate controller output
   ControllerOutput =
     CWPID1.NextOutput(ProcessVariable)
   ```
   ```
   ' Calculate new (simulated) process variable
   ' from controller output
   ProcessVariable = ProcessVariable +
     (ControllerOutput * 0.02) + (Rnd – 0.5)
   ```
   If you have a graph control to display the data, you can add a line of code to display the current values of the three variables. For example, if you have the ComponentWorks graph control, you can use the following code:
   ```
   CWGraph1.ChartY Array(ProcessVariable,
     ControllerOutput, Setpoint), 1, False
   ```

Testing Your Program

Run your program. Once the program starts, the PID control drives the process variable to the setpoint. Your application should look similar to the following figure. Notice the process variable converging on the setpoint and oscillating around the setpoint when it reaches the setpoint value. Better PID parameters decrease the amplitude of the oscillation.



Process Variable 50.1982625577131
Setpoint 50
Controller Output 5.5344227298032

Figure 6-3. Testing the Simple PID Example

When you change the setpoint value, the PID controller drives the process variable to the new setpoint value.

Part 2: Autotune PID Parameters

In the second part of this tutorial, you add autotuning to improve the PID parameters for your system or process.

Designing the Form

Add a button and change its caption to `Autotune`.

Setting PID Properties

1. On the Autotune property page for the PID control, change **Noise level** to `0.25` and **Relay amplitude** to `2`.
2. Check the **Apply autotune results to the active parameters** box.

Developing the Code

Define an event handler routine for the `Click` event of the **Autotune** button. In the event handler, call the `ShowDialog` method of the CWPIDAutotune property object with the parameter `cwpid20ms`.

```
CWPID1.Autotune.ShowDialog cwpid20ms
```

Testing Your Program

Run your program. Allow the process to settle and then press the **Autotune** button. The Autotune Wizard is displayed, and you can follow the directions in the wizard to optimize the PID parameters.

7 PID Examples

This chapter describes examples included with the ComponentWorks Autotuning PID software.

AutotunePID

The AutotunePID example is an extension of the GeneralPID example, which is described in the *General PID* section. Press the **Autotune** button to display the Autotune Wizard. The Autotune Wizard leads you through the autotuning process. When the autotuning process is complete, the parameter controls are updated with the autotuned parameters.

Cascade and Selector

This example demonstrates a cascade and selector control, which simulates a compressor driven by a motor with a tachometer requiring a PID loop to control the speed (the downstream loop). The flow and pressure from the compressor pass to individual PID controllers.

You want to control the flow, but if the pressure exceeds a specified setpoint, the pressure becomes the controlled variable. This calls for a low select function to combine the two upstream controller outputs. The lower of the two outputs becomes the setpoint for the compressor speed.

☞ **Note** ***All variables in this simulation are percentages. In a real application, you might want to normalize all the input and setpoint values to percentages before passing them to the PID controllers.***

The downstream loop (also known as the inner loop), which is the compressor speed control, must be faster than the outer loops. You should use a factor of 10 to prevent oscillation. In this simulation, the compressor lag is smaller (faster) than that of the outer loops.

General PID

The General PID example demonstrates how to use the PID control in a simple Plant Simulator scenario. Like the Tank Level example, this pressure controller simulation uses the Plant Simulator subroutines.

LeadLag

The LeadLag example uses the LeadLag method. Excitation for the LeadLag example is either a sine wave or a square wave. The waveform is synchronized to the cycle time you choose. By varying the tuning parameters, you can see the time-domain response of the LeadLag example. A large lead setting causes a wild ringing on the output, while a large lag setting heavily filters the signal, almost making it disappear.

PID with MIO Board

The PID with MIO Board example turns your computer into a single-loop PID controller through the use of a National Instruments data acquisition board. Connect the analog output to the analog input through the resistor-capacitor network shown on the picture below.

Note ***You must have the appropriate hardware to run this example. The pin numbers shown in Figure 7-1 correspond to the standard MIO pinouts, such as a standard 50-pin cable from an E-series DAQ board.***

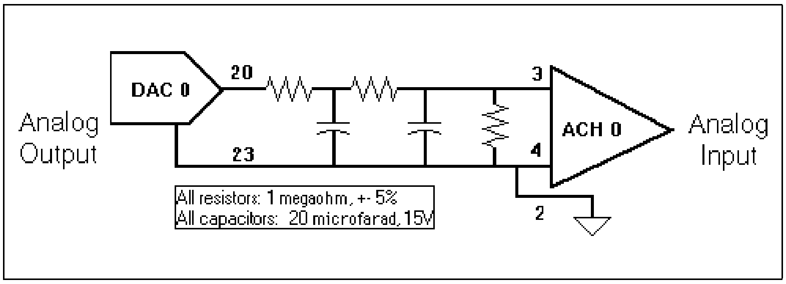

DAC 0
Analog Output
20
23
3
4
2
ACH 0
Analog Input
All resistors: 1 megaohm, +- 5%
All capacitors: 20 microfarad, 15V

Figure 7-1. Resistor-Capacitor Network

This example adjusts the analog output so that the input (process variable or PV) equals the setpoint (SP). The example displays SP and PV on a strip chart. You can experiment with different controller tuning methods and try for the fastest settling time with the least overshoot. The default tuning parameters are optimum for the network shown in Figure 7-1.

The input span is –10 to 10V, and the output span is 0 to 10V. PV and SP are expressed in volts. Set your I/O board for differential input to ±10V range, and the output to bipolar 10V range (these are all factory defaults). If you use other settings, change the example constants for the data acquisition board configuration.

The recommended network has a DC gain of 0.33, an effective deadtime of about 5 s, and an effective time constant of about 30s.

To customize this demonstration example, add alarm limits that set the digital output lines on the I/O board, remote setpoint (by using one of the analog inputs), and remote automatic to manual switching (by using one of the digital inputs).

Note ***Try using the autotuning functionality to see the effect of autotuning the controller.***

RampDemo

The RampDemo example uses the Ramp control to drive an initial value to a setpoint using a specified rate and cycle time. While the initial value is driven to the setpoint, you can change the rate and cycle time to vary the response of the control.

Tank Level

The Tank Level example is a process simulation for tank level. A level controller adjusts the flow into a tank. To represent a change in process loading, click the on/off value that serves as a drain. With this example, you also can switch between having the controller automatically determine the output or using the manual control to select the output.

The Plant Simulator subroutine, which simulates this process, reads and delays the previous valve position and scales it according to the process gain. The gain represents how fast the tank fills versus the position of the valve. The process load value depends on the state of HV-101, the drain valve. When you open the valve, the tank level drops.

Part III

PID Algorithms and Control Strategies

This section describes the Autotuning PID algorithms and basic control design systems.

Part III, *PID Algorithms and Control Strategies*, contains the following chapters.

- Chapter 8, *Algorithms*, explains the PID algorithm, the Autotuning algorithm, and how these algorithms are applied to control systems.
- Chapter 9, *Designing Control Strategies*, describes how you can design and implement control strategies with the PID, LeadLag, and Ramp controls.

Algorithms

This chapter explains the PID algorithm, the Autotuning algorithm, and how these algorithms are applied to control systems.

PID Algorithm

In the PID (Proportional-Integral-Derivative) controller, the setpoint is compared to the process variable to obtain the error

$$e = SP - PV$$

You can then calculate the controller action theoretically as

$$u(t) = K_c\left(e + \frac{1}{T_i}\int_0^t e dt + T_d \frac{de}{dt}\right)$$

where K_c is controller gain. If the error and the controller output have the same range, that is –100% to 100%, controller gain is the reciprocal of *proportional band*. T_i is the integral time in minutes (also called *reset time*), and T_d is the derivative time in minutes (also called *rate time*). The proportional action is

$$u_p(t) = K_c e$$

the integral action is

$$u_I(t) = \frac{K_c}{T_i}\int_0^t e dt$$

and the derivative action is

$$u_D(t) = K_c T_d \frac{de}{dt}$$

The PID control implements the positional PID algorithm as described in the following sections.

PV filtering—Process variable filtering minimizes the effects of noise.

$$PV_f = 0.5PV + 0.25PV(k-1) + 0.175PV(k-2) + 0.075PV(k-3)$$

Error calculation—The current error used in calculating *integral action* and *derivative action* is

$$e(k) = (SP - PV_f)(L + (1-L)^* \frac{|SP - PV_f|}{SP_{rng}})$$

The error for calculating *proportional action* is

$$eb(k) = (\beta^* SP - PV_f)(L + (1-L)^* \frac{|\beta SP - PV_f|}{SP_{rng}})$$

where SP_{rng} is the range of the setpoint, β (beta) is the setpoint factor (for the *Two Degree of Freedom PID* algorithm described under *Proportional Action*), and L is the linearity factor that produces a nonlinear gain term in which the controller gain increases with the magnitude of the error. If L is 1, the controller is linear. A value of 0.1 makes the minimum gain of the controller 10% K_c. This use of a nonlinear gain term is referred to as an Error-Squared PID algorithm.

Proportional Action—In applications, setpoint changes are normally greater and more rapid than load disturbances, while load disturbances appear as a slow departure of the controlled variable from the setpoint. PID tuning for good load-disturbance responses often results in setpoint responses of unacceptable oscillation. On the other hand, tuning for good setpoint responses often yields sluggish load-disturbance responses. The factor β, when set to less than 1, reduces the setpoint-response overshoot without affecting the load-disturbance response. This is referred to as a *Two Degree of Freedom PID* algorithm. Intuitively, β is an index of the setpoint response importance, from 0 to 1. For example, if you consider load response the most important loop performance, set β to 0. Conversely, if you want the process variable to follow the setpoint change quickly, set β to 1.

$$u_P(k) = (K_c^* eb(k))$$

Trapezoidal Integration—Trapezoidal integration is used to avoid sharp changes in integral action when there is a PV or SP jump; **nonlinear adjustment of integral action** is used to counteract overshoot. The larger the error, the smaller the integral action, as shown in the following formula and Figure 8-1.

$$u_I(k) = \frac{K_c}{T_i} \sum_{i=1}^{k} \left[\frac{e(i) + e(i-1)}{2}\right] \Delta t \left[\frac{1}{1 + \frac{10 * e(i)^2}{SP_{rng}^{\;2}}}\right]$$

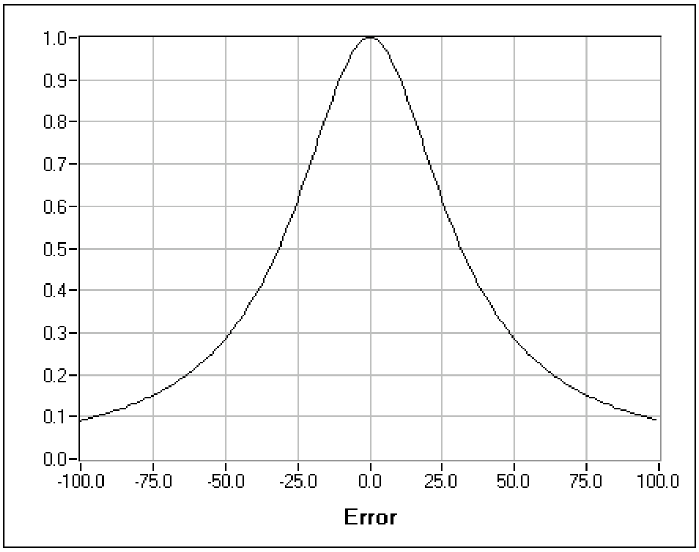


Figure 8-1. Nonlinear Multiple for Integral Action ($SP_{rng} = 100$)

Partial Derivative Action—Because of abrupt changes in setpoint (SP), derivative action is applied only to a *filtered* PV (not the error *e*) to avoid *derivative kick.*

$$u_D(k) = -K_c \frac{T_d}{\Delta_t} (PV_f(k) - PV_f(k-1))$$

Controller Output—Controller output is the summation of the proportional, integral, and derivative action.

$$u(k) = u_P(k) + u_I(k) + u_D(k)$$

Output Limiting—The actual controller output is limited to the range specified for control output.

$$\text{If } u(k) \geq u_{max} \text{ then } u(k) = u_{max}$$

and

$$\text{if } u(k) \leq u_{min} \text{ then } u(k) = u_{min}$$

the practical model of the PID controller is

$$u(t) = K_c\left[(\beta SP - PV) + \frac{1}{T_i}\int_0^t (SP - PV)dt - T_d\frac{dPV_f}{dt}\right]$$

The PID control uses an *integral sum correction algorithm* that facilitates *anti-windup* and *bumpless* automatic to manual and manual to automatic transfers. Anti-windup is the upper limit of the controller output, for example, 100%. Once the error *e* decreases, the controller output decreases and steps out of the windup area. This algorithm prevents abrupt controller output changes when you switch from automatic to manual mode or from manual to automatic mode or change any other parameters.

The default ranges for the parameters **setpoint**, **process variable**, and **output** correspond to percentage values; however, you can use actual engineering units. Adjust corresponding ranges accordingly. **Reverse acting** (also called increase-decrease) is the *normal* controller mode in which the output decreases if the **process variable** is greater than the **setpoint**. The parameters T_i and T_d are specified in minutes. Switching to hold mode or manual mode freezes the output at the current value. In the manual model, you can increase or decrease the output by changing the manual input. All transfers, from automatic to manual or automatic to hold, and from manual to automatic or hold to automatic, are bumpless.

Note ***As a general rule, manually drive the process variable until it meets or approaches the setpoint before you perform the manual to automatic transfer.***

Gain Scheduling

Gain scheduling describes a system where controller parameters are changed depending on measured operating conditions. For instance, the scheduling variable can be the setpoint, the process variable, a controller output, or an external signal. For historical reasons, the word *gain scheduling* is used even if other parameters such as **derivative time** or **integral time** change. Gain scheduling effectively controls a system whose dynamics change with the operating conditions.

In this software, you can define unlimited sets of PID parameters for gain scheduling. For each schedule you can run autotuning to update the PID parameters.

Autotuning Algorithm

Autotuning is used to improve performance. Often, many controllers are poorly tuned—some too aggressive, some too sluggish. When you are not sure about disturbance or process dynamic characteristics, tuning a PID controller is difficult; therefore, the need for autotuning arises.

Figure 8-2 illustrates the autotuning procedure excited by the *setpoint relay experiment*, which connects a relay and an extra feedback signal with the setpoint. The existing controller remains in the loop.

Note ***Although it might be poorly tuned, a stable controller must be established before autotuning.***

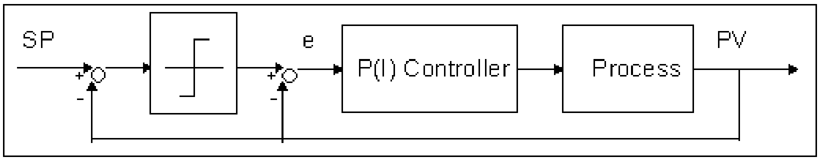


Figure 8-2. Process under PID Control with Setpoint Relay

For most systems, a limiting cycle generates because of the nonlinear relay characteristic. From this cycle, the autotuning algorithm identifies the relevant information needed for PID tuning:

- If the existing controller is proportional only, ultimate gain K_u and ultimate period T_u.
- If the existing model is PI or PID, dead time τ and time constant T_p, which are two parameters in the integral-plus-deadtime model

$$G_P(s) = \frac{e^{-\tau s}}{T_p s}$$

Tuning Formulas

This package uses Ziegler and Nichols' heuristic methods for determining the parameters of a PID controller. When autotuning, select one of three types of loop performance: fast (1/4 damping ratio), normal (some overshoot), and slow (little overshoot). Refer to the following tuning formula tables for each type of loop performance.

Table 8-1. Tuning Formula under P-only Control (fast)

Controller	K_c	T_i	T_d
P	$0.5K_u$		
PI	$0.4K_u$	$0.8T_u$	
PID	$0.6K_u$	$0.5T_u$	$0.12T_u$

Table 8-2. Tuning Formula under P-only Control (normal)

Controller	K_c	T_i	T_d
P	$0.2K_u$		
PI	$0.18K_u$	$0.8T_u$	
PID	$0.25K_u$	$0.5T_u$	$0.12T_u$

Table 8-3. Tuning Formula under P-only Control (slow)

Controller	K_c	T_i	T_d
P	$0.13K_u$		
PI	$0.13K_u$	$0.8T_u$	
PID	$0.15K_u$	$0.5T_u$	$0.12T_u$

Table 8-4. Tuning Formula under PI Control (fast)

Controller	K_c	T_i	T_d
P	T_p/τ		
PI	$0.9T_p/\tau$	3.33τ	
PID	$1.1T_p/\tau$	2.0τ	0.5τ

Table 8-5. Tuning Formula under PI Control (normal)

Controller	K_c	T_i	T_d
P	$0.44T_p/\tau$		
PI	$0.4T_p/\tau$	5.33τ	
PID	$0.53T_p/\tau$	4.0τ	0.8τ

Table 8-6. Tuning Formula under PI Control (slow)

Controller	K_c	T_i	T_d
P	$0.26T_p/\tau$		
PI	$0.24T_p/\tau$	5.33τ	
PID	$0.32T_p/\tau$	4.0τ	0.8τ

Note ***During tuning, the process remains under closed-loop (PID) control. You do not need to switch off the existing controller and perform the experiment under open-loop conditions. In the setpoint relay experiment, the SP signal mirrors the SP for the PID controller.***

9 Designing Control Strategies

This chapter describes how you can design and implement control strategies with the PID, LeadLag, and Ramp controls.

When designing a control strategy, sketch a process flowchart showing control elements (for example, valves) and measurements. Add feedback and any required computations. Then translate the flowchart into code using the ComponentWorks PID control. Figure 9-1 shows a process flowchart.

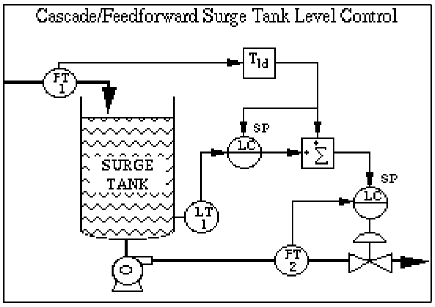


Figure 9-1. Control Flowchart

The following Visual Basic code implements the process shown in Figure 9-1.

```
SP1 = CWLeadLag1.NextOutput(FT1)
CWPID1.Setpoint = SP1
Output1 = CWPID1.NextOutput(LT1)
CWPID2.Setpoint = Output1 + SP1
Valve = CWPID2.NextOutput(FT2)
```

You can handle the inputs and outputs through data acquisition (DAQ) boards, FieldPoint I/O modules, GPIB instruments, or serial I/O ports.

Setting Timing

The `PID`, `LeadLag`, and `RampGenerator` methods are time dependent. These methods can acquire timing information through a value you supply or through a built-in time keeper. With the built-in time keeper, the methods calculate new timing information each time they are called. At each call, the method measures the time since the last call and uses that difference in its calculations. Tick timer resolution is limited to 1 ms on Windows 95 and NT. Because of this limitation, do not try to run the PID methods faster than 10 Hz when using the built-in time keeper.

If you specify the timing information manually, the method uses the value you specify in the calculations, regardless of the elapsed time. This method should be used for fast loops, such as when controller input is timed with acquisition hardware.

According to control theory, a sampled control system must run about 10 times faster than the fastest time constant in the plant under control. For example, a temperature control loop is probably quite slow—a time constant of 60 s is common in a small system. In this case, a cycle time of about 6 s is sufficient. Faster cycling offers no improvement in performance.

If the control application does not contain graphics that must update frequently, the PID control can execute at kilohertz (kHz) rates. Remember that actions such as mouse activity and window scrolling interfere with these rates.

Manual Tuning Techniques

The following controller tuning procedures are based on the work of Ziegler and Nichols, the developers of the Quarter-Decay Ratio tuning techniques derived from a combination of theory and empirical observations. For different processes, one method might be easier or more accurate than the other. Some techniques you can use with online controllers cannot stand the gross upsets described here.

To perform these tests, set up your control strategy with the process variable (PV), setpoint, and output displayed on a large strip chart with the axes showing the values versus time. Perturb the loop as described in the *Closed-Loop (Ultimate Gain) Tuning Procedure* and *Open-Loop (Step Test) Tuning Procedure* sections of this chapter and determine the response from the graph.

Closed-Loop (Ultimate Gain) Tuning Procedure

Although the closed-loop (ultimate gain) tuning procedure is very accurate, you must put your process in steady-state oscillation and observe the **process variable** on a strip chart. To perform the closed-loop tuning procedure, complete the following steps:

1. With the controller in automatic mode, carefully increase the proportional gain (K_c) in small steps. Disturb the loop after each step by making a small change in the setpoint. The process variable should start oscillating as you increase the K_c. Keep making changes until the oscillation is perfectly sustained, neither growing nor decaying over time.
2. Record the controller proportional band as PB_u as a percent, where $PB_u = 100 / K_c$.
3. Record the period of oscillation as T_u in minutes.
4. Multiply the measured values by the factors shown in Table 9-1, and enter the new tuning parameters into your controller. The table provides the proper values for a quarter-decay ratio.

   If you want less overshoot, reduce the gain K_c.

Table 9-1. Closed-Loop–Quarter-Decay Ratio Values

Controller	PB (percent)	Reset (minutes)	Rate (minutes)
P	$2.00PB_u$	—	—
PI	$2.22PB_u$	$0.83T_u$	—
PID	$1.67PB_u$	$0.50T_u$	$0.125T_u$

Note ***Proportional gain (K_c) is related to proportional band (PB) as $K_c = 100 / PB$.***

Open-Loop (Step Test) Tuning Procedure

The open-loop (step test) tuning procedure assumes that you can model any process as a first-order lag and a pure deadtime. This method requires more analysis than the closed-loop tuning procedure, but your process does not need to reach sustained oscillation. Therefore, the open-loop tuning procedure might be quicker and less hazardous for many processes. Observe the output and the process variable (PV) on a strip chart that shows time on the X axis. To perform the open-loop tuning procedure, complete the following steps:

1. Put the controller in manual mode, set the output to a nominal operating value, and allow the PV to settle completely. Record the PV and output values.
2. Make a step change in the output. Record the new output value.
3. Wait for the PV to settle. From the chart, determine the values as derived from the sample displayed in Figure 9-2.

   The values are as follows:

   - *Td*—Deadtime in minutes
   - *T*—Time constant in minutes
   - *K*—Process gain = $\frac{\text{change in output}}{\text{change in } PV}$

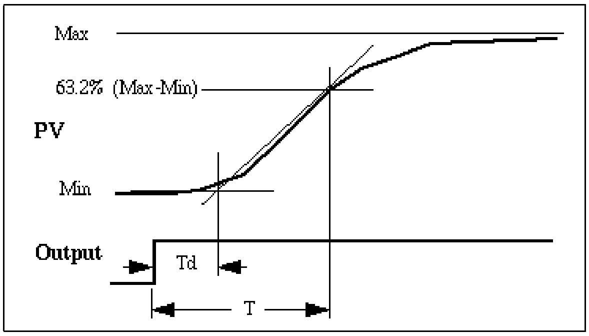


Figure 9-2. Output and Process Variable Strip Chart

4. Multiply the measured values by the factors shown in Table 9-2, and enter the new tuning parameters into your controller. The table provides the proper values for a quarter-decay ratio.

   If you want less overshoot, reduce the gain K_c.

Table 9-2. Open-Loop–Quarter-Decay Ratio Values

Controller	PB (percent)	Reset (minutes)	Rate (minutes)
P	$100\frac{KT_d}{T}$	—	—
PI	$110\frac{KT_d}{T}$	$3.33T_d$	—
PID	$80\frac{KT_d}{T}$	$2.00T_d$	$0.50T_d$

Error Codes

This appendix lists the error codes returned by ComponentWorks.

Table A-1. ComponentWorks Errors

Error Code	Description
–30000	Unexpected error.
–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	Divide by zero error.
–30005	Result of a calculation is an imaginary number.
–30006	Overflow error.
–30007	Out of memory.
–30008	You have called a function or method requiring a ComponentWorks product for which you do not have a license. For example, you might be using a method that is not supported in the base or standard Analysis package. To upgrade your product, contact National Instruments.

B

Distribution and Redistributable Files

This appendix contains information about ComponentWorks Autotuning PID 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\cwpid.ocx
```

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. 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.

Customer Communication

For your convenience, this appendix contains forms to help you gather the information necessary to help us solve your technical problems and a form you can use to comment on the product documentation. When you contact us, we need the information on the Technical Support Form and the configuration form, if your manual contains one, about your system configuration to answer your questions as quickly as possible.

National Instruments has technical assistance through electronic, fax, and telephone systems to quickly provide the information you need. Our electronic services include a bulletin board service, an FTP site, a fax-on-demand system, and e-mail support. If you have a hardware or software problem, first try the electronic support systems. If the information available on these systems does not answer your questions, we offer fax and telephone support through our technical support centers, which are staffed by applications engineers.

Electronic Services

Bulletin Board Support

National Instruments has BBS and FTP sites dedicated for 24-hour support with a collection of files and documents to answer most common customer questions. From these sites, you can also download the latest instrument drivers, updates, and example programs. For recorded instructions on how to use the bulletin board and FTP services and for BBS automated information, call 512 795 6990. You can access these services at:

United States: 512 794 5422
Up to 14,400 baud, 8 data bits, 1 stop bit, no parity

United Kingdom: 01635 551422
Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

France: 01 48 65 15 59
Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

FTP Support

To access our FTP site, log on to our Internet host, `ftp.natinst.com`, as `anonymous` and use your Internet address, such as `joesmith@anywhere.com`, as your password. The support files and documents are located in the `/support` directories.

Fax-on-Demand Support

Fax-on-Demand is a 24-hour information retrieval system containing a library of documents on a wide range of technical information. You can access Fax-on-Demand from a touch-tone telephone at 512 418 1111.

E-Mail Support (Currently USA Only)

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

`support@natinst.com`

Telephone and Fax Support

National Instruments has branch offices all over the world. Use the list below to find the technical support number for your country. If there is no National Instruments office in your country, contact the source from which you purchased your software to obtain support.

Country	Telephone	Fax
Australia	03 9879 5166	03 9879 6277
Austria	0662 45 79 90 0	0662 45 79 90 19
Belgium	02 757 00 20	02 757 03 11
Brazil	011 288 3336	011 288 8528
Canada (Ontario)	905 785 0085	905 785 0086
Canada (Québec)	514 694 8521	514 694 4399
Denmark	45 76 26 00	45 76 26 02
Finland	09 725 725 11	09 725 725 55
France	01 48 14 24 24	01 48 14 24 14
Germany	089 741 31 30	089 714 60 35
Hong Kong	2645 3186	2686 8505
Israel	03 6120092	03 6120095
Italy	02 413091	02 41309215
Japan	03 5472 2970	03 5472 2977
Korea	02 596 7456	02 596 7455
Mexico	5 520 2635	5 520 3282
Netherlands	0348 433466	0348 430673
Norway	32 84 84 00	32 84 86 00
Singapore	2265886	2265887
Spain	91 640 0085	91 640 0533
Sweden	08 730 49 70	08 730 43 70
Switzerland	056 200 51 51	056 200 51 55
Taiwan	02 377 1200	02 737 4644
United Kingdom	01635 523545	01635 523154
United States	512 795 8248	512 794 5678

Technical Support Form

Photocopy this form and update it each time you make changes to your software or hardware, and use the completed copy of this form as a reference for your current configuration. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

If you are using any National Instruments hardware or software products related to this problem, include the configuration forms from their user manuals. Include additional pages if necessary.

Name ____

Company ____

Address ____

____

Fax (___) ________________Phone (___) ____________________________

Computer brand____________ Model ________________Processor________________

Operating system (include version number) ____________________________

Clock speed ______MHz RAM _____MB Display adapter ____________________

Mouse ___yes ___no Other adapters installed____________________________

Hard disk capacity _____MB Brand____________________________

Instruments used ____

____

National Instruments hardware product model____________ Revision ______________

Configuration ____

National Instruments software product________________ Version ______________

Configuration ____

The problem is: ____

____

____

____

____

List any error messages: ____

____

____

The following steps reproduce the problem: ____________________________

____

____

____

____

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

Hardware revision ____

Interrupt level of hardware ____

DMA channels of hardware ____

Base I/O address of hardware ____

Programming choice ____

National Instruments software ____

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™ Autotuning PID*

Edition Date: August 1998

Part Number: 322064A-01

Please comment on the completeness, clarity, and organization of the manual.

____

____

____

____

____

____

____

If you find errors in the manual, please record the page numbers and describe the errors.

____

____

____

____

____

____

____

Thank you for your help.

Name ____

Title ____

Company ____

Address ____

____

E-Mail Address ____

Phone (___) ____________________________ Fax (___) ____________________________

Mail to: Technical Publications
National Instruments Corporation
6504 Bridge Point Parkway
Austin, Texas 78730-5039

Fax to: Technical Publications
National Instruments Corporation
512 794 5678

Glossary

Prefix	Meaning	Value
k-	kilo-	10^3
m-	milli-	10^{-3}
μ-	micro-	10^{-6}
n-	nano-	10^{-9}

Symbols

°	Degrees.
∞	Infinity.
Ω	Ohms.
%	Percent.

A

A	Amperes.
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 PID, DAQ, and UI tools in ComponentWorks are all ActiveX controls. An ActiveX control has properties, methods, objects, and events.
algorithm	A prescribed set of well-defined rules or processes for the solution of a problem in a finite number of steps.
anti-reset windup	A method that prevents the integral term of the PID algorithm from moving too far beyond saturation when an error persists.

B

bias	The offset added to a controller's output.
bumpless transfer	A process in which the next output always increments from the current output, regardless of the current controller output value; therefore, transfer from automatic to manual control is always bumpless.

C

C	Celsius.
cascade control	Control in which the output of one controller is the setpoint for another controller.
closed loop	A signal path which includes a forward path, a feedback path, and a summing point and which forms a closed circuit. Also called a *feedback loop*.
collection	Control property and object that contains a number of objects of the same type, such as PID parameters. The type name of the collection is the plural of the type name of the object in the collection. For example, a collection of CWPIDParameter objects is called CWPIDParameters. To reference an object in the collection, you must specify the object as part of the collection, usually by index. For example, `CWPID.Parameters.Item(2)` is the second parameter in the CWPIDParameters collection of the control.
controller output	*See* manipulated variable.

D

damping	The progressive reduction or suppression of oscillation in a device or system.
DC	Direct current.
dead time (T_d)	The interval of time, expressed in minutes, between initiation of an input change or stimulus and the start of the resulting observable response.
derivative (control) action	Control response to the time rate of change of a variable. Also called *rate action*.

deviation	Any departure from a desired value or expected value or pattern.
device	Plug-in data acquisition board that can contain multiple channels and conversion devices.
downstream loop	In a cascade, the controller whose setpoint is provided by another controller.
driver	Software that controls a specific hardware device, such as a data acquisition board.

E

EGU	Engineering units.
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.
exception	Error message generated by a control and sent directly to the application or programming environment containing the control.

F

FC	Flow controller.
feedback control	Control in which a measured variable is compared to its desired value to produce an actuating error signal that is acted upon in such a way as to reduce the magnitude of the error.
feedback loop	*See* closed loop.
form	Window or area on the screen on which you place controls and indicators to create the user interface for your program.

G

gain	For a linear system or element, the ratio of the magnitude (amplitude) of a steady-state sinusoidal output relative to the causal input; the length of a phasor from the origin to a point of the transfer locus in a complex plane. Also called the *magnitude ratio*.
General Purpose Interface Bus (GPIB)	The common name for the communications interface system defined in ANSI/IEEE Standard 488.1-1987 and ANSI/IEEE Standard 488.2-1987.

H

Hz	Hertz.

I

Instrument Society of America (ISA)	The organization that sets standards for process control instrumentation in the United States.
integral action rate	*See* reset rate.
integral (control) action	Control action in which the output is proportional to the time integral of the input. That is, the rate of change of output is proportional to the input.
I/O	Input/output.

K

K	Process gain.
K_c	Controller gain.
kHz	Kilohertz.

L

lag	A lowpass filter or integrating response with respect to time.
loop cycle time	Time interval between calls to a control algorithm.

M

magnitude ratio	*See* gain.
manipulated variable	A quantity or condition that is varied as a function of the actuating error signal so as to change the value of the directly controlled variable. Also called *controller output*.
MB	Megabytes of memory.
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.
ms	Milliseconds.

N

noise	In process instrumentation, an unwanted component of a signal or variable. Noise may be expressed in units of the output or in percent of output span.

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.

output limiting	Preventing a controller's output from travelling beyond a desired maximum range.
overshoot	The maximum excursion beyond the final steady-state value of output as the result of an input change. Also called *transient overshoot*.

P

P	Proportional.
P controller	A controller which produces proportional control action only; that is, a controller that has only a simple gain response.
PC	Pressure controller.
PD	Proportional, derivative.
PD controller	A controller that produces proportional plus derivative (rate) control action.
PI	Proportional, integral.
PI controller	A controller that produces proportional plus integral (reset) control action.
PID	Proportional, integral, derivative.
PID controller	A controller that produces proportional plus integral (reset) plus derivative (rate) control action.
process gain (K)	For a linear process, the ratio of the magnitudes of the measured process response to that of the manipulated variable.
process variable (PV)	The measured variable (such as pressure or temperature) in a process to be controlled.
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 setpoint (property) of a Ramp (object), while an object property is the Autotune (property) on the PID (object). Autotune itself is another object with properties, such as noise level and controller type.

proportional band (PB)	The change in input required to produce a full range change in output due to proportional control action. PB = 100/K_c
proportional kick	The response of a proportional controller to a step change in the setpoint or process variable.

Q

Quarter Decay Ratio	A response in which the amplitude of each oscillation is one-quarter that of the previous oscillation.

R

ramp	The total (transient plus steady-state) time response resulting from a sudden increase in the rate of change from zero to some finite value of the input stimulus. Also called *ramp response*.
rate action	Control response to the time rate of change of a variable. Also called *derivative control action*.
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.
reset rate	Of proportional plus integral or proportional plus integral plus derivative control action devices: for a step input, the ratio of the initial rate of change of output due to integral control action to the change in steady-state output due to proportional control action. Of integral control action devices: for a step input, the ratio of the initial rate of change of output to the input change. Also called *integral action rate*.
reverse acting (increase-decrease) controller	A controller in which the value of the output signal decreases as the value of the input (measured variable) increases.
RPM	Revolutions per minute.

S

s	Seconds.
scope chart	Chart indicator modeled on the operation of an oscilloscope.
selector control	The use of multiple controllers and/or multiple process variables in which the connections may change dynamically depending on process conditions.
setpoint (SP)	An input variable that sets the desired value of the controlled process variable.
skeleton function	Applies a succession of thinning operations to an object until its width becomes one pixel.
span	The algebraic difference between the upper and lower range values.
strip chart	A plotting indicator modeled after a paper strip chart recorder, which scrolls as it plots data.

T

time constant (T)	In process instrumentation, the value T (in minutes) in an exponential response term, A exp (–t/T), or in one of the transform factors, such as 1+sT.
transient overshoot	*See* overshoot.

V

V	Volts.
valve dead band	In process instrumentation, the range through which an input signal may be varied, upon reversal of direction, without initiating an observable change in output signal.

W

While Loop	Post-iterative-test loop structure that repeats a section of code until a condition is met. Comparable to a Do loop or a Repeat-Until loop in conventional programming languages.

Index

A

ActiveX controls, 1-1. *See also* controls.
algorithms
- autotuning algorithm, 8-5 to 8-7
- PID algorithm, 8-1 to 8-5

anti-windup transfers, 8-4
application development
- Delphi, 5-1 to 5-7
  - building user interface, 5-3 to 5-5
  - editing properties programmatically, 5-6
  - events, 5-7
  - loading ComponentWorks controls into palette, 5-2 to 5-3
  - methods, 5-7
  - programming with ComponentWorks, 5-6 to 5-7
  - running examples, 5-1
- getting started, 2-3 to 2-5
- Visual Basic, 3-1 to 3-9
  - automatic code completion, 3-9
  - building user interface, 3-2 to 3-5
  - developing event routines, 3-5 to 3-6
  - loading ComponentWorks controls into toolbox, 3-2
  - pasting code into programs, 3-8 to 3-9
  - procedure overview, 3-1
  - using Object Browser, 3-6 to 3-8
  - working with methods, 3-5
- Visual C++, 4-1 to 4-9
  - adding ComponentWorks controls to toolbar, 4-3 to 4-4
  - building user interface, 4-4 to 4-5
  - events, 4-8 to 4-9
  - methods, 4-8
  - MFC AppWizard, 4-2 to 4-3
  - procedure overview, 4-1
  - programming with ComponentWorks controls, 4-5 to 4-6
  - properties, 4-6 to 4-8

Application Wizard, MFC, 4-2 to 4-3
automatic code completion, in Visual Basic, 3-9
Autotune object, 6-3
AutotunePID example, 7-1
Autotune Wizard, 6-4
autotuning algorithm, 8-5 to 8-7. *See also* tuning.
- setpoint relay experiment (figure), 8-5
- tuning formulas, 8-5 to 8-6

autotuning PID parameters, 6-8 to 6-9

B

bulletin board support, C-1
bumpless transfers, 8-4

C

C++. *See* Visual C++.
Cascade and Selector example, 7-1
closed-loop (ultimate gain) tuning procedure, 9-3
code completion, automatic, in Visual Basic, 3-9
collection objects, 1-6
ComponentWorks Autotuning PID. *See also* PID control.
- components, 1-2
- examples structure, 2-3
- exploring documentation, 2-1 to 2-3
- getting started, 2-1 to 2-6
- installing, 1-3 to 1-4
- online reference, 2-2 to 2-3
- overview, 1-1 to 1-2
- sources for additional information, 2-5 to 2-6
- system requirements, 1-2 to 1-3

ComponentWorks Support Web Site, 2-5 to 2-6
compressor (Cascade and Selector) example, 7-1
control strategy design, 9-1 to 9-5
- closed-loop (ultimate gain) tuning procedure, 9-3
- control flowchart (figure), 9-1
- manual tuning techniques, 9-2
- open-loop (step test) tuning procedure, 9-4 to 9-5
- setting timing, 9-2

controller output value, 1-1
controls, 1-4 to 1-6. *See also* events; methods; PID control; properties.
- collection objects, 1-6
- LeadLag control, 6-4
- loading into programming environments
  - Delphi, 5-2 to 5-3
  - Visual Basic, 3-2
  - Visual C++, 4-3 to 4-4
- object hierarchy, 1-5 to 1-6
- properties, methods, and events, 1-4 to 1-5
- Ramp control, 6-5

custom property page
- definition, 1-8
- example (figure), 1-8

customer communication, *xv,* C-1 to C-2

D

Delphi, 5-1 to 5-7
- building user interface, 5-3 to 5-5
- editing properties programmatically, 5-6
- events, 5-7
- loading ComponentWorks controls into palette, 5-2 to 5-3
- methods, 5-7
- newest version of Delphi required (note), 5-1
- programming with ComponentWorks, 5-6 to 5-7
- running examples, 5-1

designing control strategies. *See* control strategy design.
developing applications. *See* application development.
distribution and redistribution files, B-1 to B-4
- ComponentWorks evaluation, B-3
- distribution procedure, B-1 to B-3
  - automatic installers, B-2
  - manual installation, B-2 to B-3
- files required, B-1
- running on target computer, B-3
- run-time licenses, B-3
- troubleshooting, B-4

documentation. *See also* online reference.
- conventions used in manual, *xiii-xiv*
- exploring, 2-1 to 2-3
- organization of manual, *xi-xiii*
- related documentation, *xiv*

E

electronic support services, C-1 to C-2
e-mail support, C-2
error codes, A-1
event handler routines, developing
- Delphi, 5-7
- overview, 1-11
- Visual Basic applications, 3-5 to 3-6
- Visual C++ applications, 4-8 to 4-9

events
- definition, 1-5
- learning about, 1-11
- PID control, 6-4

examples
- AutotunePID, 7-1
- becoming familiar with, 2-3
- Cascade and Selector, 7-1
- General PID, 7-2

LeadLag, 7-2
location of examples, 2-3
PID with MIO board, 7-2 to 7-3
RampDemo, 7-3
Tank Level, 7-3

F

fax and telephone support numbers, C-2
Fax-on-Demand support, C-2
files installed on hard disk, 1-4
FTP support, C-1

G

gain scheduling, 8-4 to 8-5
General PID example, 7-2

H

help. *See* online reference.

I

installation, 1-3 to 1-4
Administrator privileges required (note), 1-3
distribution and redistribution files, B-2 to B-3
files installed on hard disk, 1-4
from floppy disks, 1-3
integral sum correction algorithm, 8-4
Item method, for setting properties, 1-10

L

LeadLag control, 6-4
LeadLag example, 7-2

M

manual. *See* documentation.
manual tuning techniques, 9-2. *See also* autotuning algorithm; tuning.
methods
definition, 1-4
learning about, 1-11
setting properties, 1-10
working with control methods
Delphi, 5-7
overview, 1-10
Visual Basic, 3-5
Visual C++, 4-8
Microsoft Foundation Classes Application (MFC) Wizard, 4-2 to 4-3
MIO board example, 7-2 to 7-3

O

Object Browser, in Visual Basic, 3-6 to 3-8
object hierarchy
example (figure), 1-6
overview, 1-5
PID control (figure), 6-1
objects
Autotune object, 6-3
collection objects, 1-6
Parameter object, 6-3
PID object, 6-2
OLE (Object Linking and Embedding) controls, 1-1. *See also* controls.
online reference
accessing, 1-1, 1-11, 2-2
finding specific information, 2-2 to 2-3
learning about properties, methods, and events, 1-11
overview, 2-2
searching complete text, 2-5

open-loop (step test) tuning procedure, 9-4 to 9-5
- open-loop–quarter-decay ratio values (table), 9-5
- output and process variable strip chart (figure), 9-4
- procedure, 9-4 to 9-5

P

Parameter object, 6-3
Parameters collection, 6-2
pasting code, in Visual Basic, 3-8 to 3-9
PID algorithm, 8-1 to 8-5
- controller output, 8-3
- error calculation, 8-2
- gain scheduling, 8-4 to 8-5
- integral sum correction algorithm, 8-4
- nonlinear adjustment of integral action, 8-3
- output limiting, 8-4
- partial derivative action, 8-3
- proportional action, 8-2
- PV filtering, 8-2
- trapezoidal integration, 8-3
- Two Degree of Freedom PID algorithm, 8-2

PID control, 6-1 to 6-5
- Autotune object, 6-3
- Autotune Wizard, 6-4
- events, 6-4
- object hierarchy (figure), 6-1
- overview, 1-1
- Parameter object, 6-3
- Parameters collection, 6-2

PID control tutorial, 6-5 to 6-9
- autotuning PID parameters, 6-8 to 6-9
- developing PID system, 6-5 to 6-8
  - code development, 6-7
  - form design, 6-5 to 6-6
  - setting PID properties, 6-6
  - testing your program, 6-8

PID examples, 7-1 to 7-3
- AutotunePID, 7-1
- Cascade and Selector, 7-1
- General PID, 7-2
- LeadLag, 7-2
- PID with MIO board, 7-2 to 7-3
- RampDemo, 7-3
- Tank Level, 7-3

PID object, 6-2
Plant Simulator subroutine, 7-3
process variable, 1-1
properties
- definition, 1-4
- editing programmatically
  - Delphi, 5-6
  - overview, 1-9
  - Visual Basic, 3-4 to 3-5
- learning about, 1-11
- setting, 1-7 to 1-11
  - developing event handler routines, 1-11
  - Item method, 1-10
  - using property pages, 1-7 to 1-8
  - working with methods, 1-10
- using in programming environments
  - Delphi, 5-4 to 5-5
  - Visual Basic, 3-3 to 3-5
  - Visual C++, 4-6 to 4-8

property pages
- custom property page
  - definition, 1-8
  - example (figure), 1-8
- setting properties for controls, 1-7 to 1-11
- Visual Basic default property sheets (figure), 1-8

proportional band, 8-1
Proportional-Integral-Derivative control. *See* PID control.

Q

Quarter-Decay Ratio tuning technique, 9-2

R

Ramp control, 6-5
RampDemo example, 7-3
rate time, 8-1
redistribution files.
  See distribution and redistribution files.
requirements for getting started, 1-2 to 1-3
reset time, 8-1
run-time licenses, B-3

S

Selector and Cascade example, 7-1
setpoint, 1-1
software object, 1-5
step test (open-loop) tuning procedure,
  9-4 to 9-5
system requirements, 1-2 to 1-3

T

Tank Level example, 7-3
technical support, C-1 to C-2
telephone and fax support numbers, C-2
timing of controls, setting, 9-2
troubleshooting distribution and redistribution
  files, B-4
tuning. *See also* autotuning algorithm.
  closed-loop (ultimate gain) tuning
    procedure, 9-3
  manual tuning technique, 9-2
  open-loop (step test) tuning procedure,
    9-4 to 9-5
Two Degree of Freedom PID algorithm, 8-2

U

ultimate gain (closed-loop) tuning
  procedure, 9-3
user interface, building
  Delphi applications, 5-3 to 5-5
    placing controls, 5-4
    using property pages, 5-4 to 5-5
  Visual Basic applications, 3-2 to 3-5
    editing properties programmatically,
      3-4 to 3-5
    using property pages, 3-3 to 3-4
  Visual C++ applications, 4-4 to 4-5

V

Visual Basic, 3-1 to 3-9
  automatic code completion, 3-9
  building user interface, 3-2 to 3-5
    editing properties programmatically,
      3-4 to 3-5
    using property pages, 3-3 to 3-4
  default property sheets (figure), 1-8
  developing event routines, 3-5 to 3-6
  loading ComponentWorks controls into
    toolbox, 3-2
  pasting code into programs, 3-8 to 3-9
  procedure overview, 3-1
  using Object Browser, 3-6 to 3-8
  working with methods, 3-5
Visual C++, 4-1 to 4-9
  adding ComponentWorks controls to
    toolbar, 4-3 to 4-4
  building user interface, 4-4 to 4-5
  events, 4-8 to 4-9
  methods, 4-8
  MFC AppWizard, 4-2 to 4-3
  procedure overview, 4-1
  programming with ComponentWorks
    controls, 4-5 to 4-6
  properties, 4-6 to 4-8

W

Web site for ComponentWorks support,
  2-5 to 2-6