

LabVIEW™ Basics I

Development Course Manual

Course Software Version 8.0

October 2005 Edition

Part Number 320628N-01

Copyright

© 1993–2005 National Instruments Corporation. All rights reserved.

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

In regards to components used in USI (Xerces C++, ICU, and HDF5), the following copyrights apply. For a listing of the conditions and disclaimers, refer to the `USICopyrights.chm`.

This product includes software developed by the Apache Software Foundation (<http://www.apache.org/>).

Copyright © 1999 The Apache Software Foundation. All rights reserved.

Copyright © 1995–2003 International Business Machines Corporation and others. All rights reserved.

NCSA HDF5 (Hierarchical Data Format 5) Software Library and Utilities

Copyright 1998, 1999, 2000, 2001, 2003 by the Board of Trustees of the University of Illinois. All rights reserved.

Trademarks

National Instruments, NI, ni.com, and LabVIEW are trademarks of National Instruments Corporation. Refer to the *Terms of Use* section on `ni.com/legal` for more information about National Instruments trademarks.

Other product and company names mentioned herein are trademarks or trade names of their respective companies.

Members of the National Instruments Alliance Partner Program are business entities independent from National Instruments and have no agency, partnership, or joint-venture relationship with National Instruments.

Patents

For patents covering National Instruments products, refer to the appropriate location: **Help»Patents** in your software, the `patents.txt` file on your CD, or `ni.com/legal/patents`.

Worldwide Technical Support and Product Information

ni.com

National Instruments Corporate Headquarters

11500 North Mopac Expressway Austin, Texas 78759-3504 USA Tel: 512 683 0100

Worldwide Offices

Australia 1800 300 800, Austria 43 0 662 45 79 90 0, Belgium 32 0 2 757 00 20, Brazil 55 11 3262 3599, Canada 800 433 3488, China 86 21 6555 7838, Czech Republic 420 224 235 774, Denmark 45 45 76 26 00, Finland 385 0 9 725 725 11, France 33 0 1 48 14 24 24, Germany 49 0 89 741 31 30, India 91 80 51190000, Israel 972 0 3 6393737, Italy 39 02 413091, Japan 81 3 5472 2970, Korea 82 02 3451 3400, Lebanon 961 0 1 33 28 28, Malaysia 1800 887710, Mexico 01 800 010 0793, Netherlands 31 0 348 433 466, New Zealand 0800 553 322, Norway 47 0 66 90 76 60, Poland 48 22 3390150, Portugal 351 210 311 210, Russia 7 095 783 68 51, Singapore 1800 226 5886, Slovenia 386 3 425 4200, South Africa 27 0 11 805 8197, Spain 34 91 640 0085, Sweden 46 0 8 587 895 00, Switzerland 41 56 200 51 51, Taiwan 886 02 2377 2222, Thailand 662 278 6777, United Kingdom 44 0 1635 523545

For further support information, refer to the *Additional Information and Resources* appendix. To comment on National Instruments documentation, refer to the National Instruments Web site at ni.com/info and enter the info code `feedback`.

Contents

Student Guide

A. About This Manual	viii
B. What You Need to Get Started	viii
C. Installing the Course Software.....	ix
D. Course Goals.....	x
E. Course Conventions.....	x

Lesson 1

Problem Solving

A. Software Development Method	1-2
B. Scenario	1-2
C. Design	1-2
D. Implementation	1-7
E. Testing	1-8
F. Maintenance.....	1-8
Exercise 1-1 Software Development Method.....	1-9
Exercise 1-2 Project: State Transition Diagram	1-11
G. Project.....	1-14

Lesson 2

Navigating LabVIEW

A. Virtual Instruments (VIs).....	2-2
B. Parts of a VI	2-2
C. Starting a VI.....	2-4
D. Project Explorer	2-8
E. Front Panel.....	2-12
F. Block Diagram.....	2-18
Exercise 2-1 Concept: Exploring a VI.....	2-25
G. Searching for Controls, VIs, and Functions.....	2-26
Exercise 2-2 Concept: Navigating Palettes	2-28
H. Selecting a Tool	2-29
Exercise 2-3 Concept: Selecting a Tool	2-35
I. Dataflow.....	2-39
Exercise 2-4 Concept: Dataflow.....	2-41
J. Building a Simple VI.....	2-42
Exercise 2-5 Simple AAP VI.....	2-46
K. LabVIEW Help Utilities	2-57
Exercise 2-6 Concept: Using Help	2-61

Lesson 3**Implementing a VI**

A. Front Panel Design.....	3-2
Exercise 3-1 Project: Weather Station UI VI	3-9
B. Block Diagram Terminals.....	3-14
C. Documenting Code	3-22
Exercise 3-2 Project: Determine Warnings VI.....	3-25
D. While Loops.....	3-32
Exercise 3-3 Auto Match VI.....	3-35
E. For Loops	3-41
Exercise 3-4 Concept: While Loops versus For Loops	3-44
F. Timing a VI.....	3-47
G. Iterative Data Transfer	3-48
Exercise 3-5 Average Temperature VI.....	3-51
H. Plotting Data	3-55
Exercise 3-6 Temperature Multiplot VI	3-61
I. Case Structures	3-66
Exercise 3-7 Project: Determine Warnings VI.....	3-71
J. Formula Nodes.....	3-78
Exercise 3-8 Optional: Square Root VI.....	3-80
Exercise 3-9 Optional: Determine Warnings Challenge VI	3-84
Exercise 3-10 Optional: Determine More Warnings VI.....	3-87

Lesson 4**Relating Data**

A. Arrays.....	4-2
Exercise 4-1 Concept: Manipulating Arrays	4-8
B. Clusters	4-15
Exercise 4-2 Concept: Clusters.....	4-21
Exercise 4-3 Project: Weather Station UI VI	4-26
C. Type Definitions	4-29
Exercise 4-4 Project: Weather Station CTL	4-33

Lesson 5**Debugging VIs**

A. Correcting Broken VIs.....	5-2
B. Debugging Techniques	5-4
C. Undefined or Unexpected Data.....	5-11
D. Error Checking and Error Handling.....	5-12
Exercise 5-1 Concept: Debugging.....	5-16

Lesson 6**Developing Modular Applications**

A. What is Modularity?	6-2
B. Icon and Connector Pane	6-4
C. Using SubVIs	6-9
Exercise 6-1 Project: Determine Warnings VI	6-11
Self-Review: Quiz	6-17
Self-Review: Quiz Answers	6-19

Lesson 7**Measurement Fundamentals**

A. Computer-Based Measurement Systems	7-2
B. Measurement Concepts	7-4
C. Increasing Measurement Quality	7-12
Exercise 7-1 Concepts: Measurement Fundamentals	7-18

Lesson 8**Data Acquisition**

A. Hardware	8-2
B. Software Architecture	8-5
C. Simulating a DAQ Device	8-8
Exercise 8-1 Concept: MAX	8-9
D. Analog Input	8-15
Exercise 8-2 Triggered Analog Input VI	8-17
E. Analog Output	8-22
F. Counters	8-23
Exercise 8-3 Count Events VI	8-25
G. Digital I/O	8-28
Exercise 8-4 Optional: Digital Count VI	8-29
Self-Review: Quiz	8-33

Lesson 9**Instrument Control**

A. Instrument Control	9-2
B. GPIB	9-2
C. Serial Port Communication	9-3
D. Using Other Interfaces	9-6
E. Software Architecture	9-7
Exercise 9-1 Concept: GPIB Configuration with MAX	9-9

F. Instrument I/O Assistant	9-12
Exercise 9-2 Concept: Instrument I/O Assistant	9-14
G. VISA	9-23
Exercise 9-3 VISA Write & Read VI	9-26
H. Instrument Drivers	9-29
Exercise 9-4 Concept: Instrument Driver	9-32

Lesson 10

Analyzing and Storing Measurement Data

A. Analyzing and Processing Numeric Data	10-2
Exercise 10-1 Concept: Analysis Types	10-7
B. Reading and Writing Data to File	10-9
Exercise 10-2 Optional: Read VCard VI	10-13
Self-Review: Quiz Answers	10-29

Lesson 11

Common Design Techniques and Patterns

A. Sequential Programming	11-2
B. State Programming	11-4
C. State Machines	11-5
Exercise 11-1 Project: Temperature Weather Station	11-12
D. Parallelism	11-28
Summary	11-29

Appendix A

Additional Information and Resources

Index

Course Evaluation

Student Guide

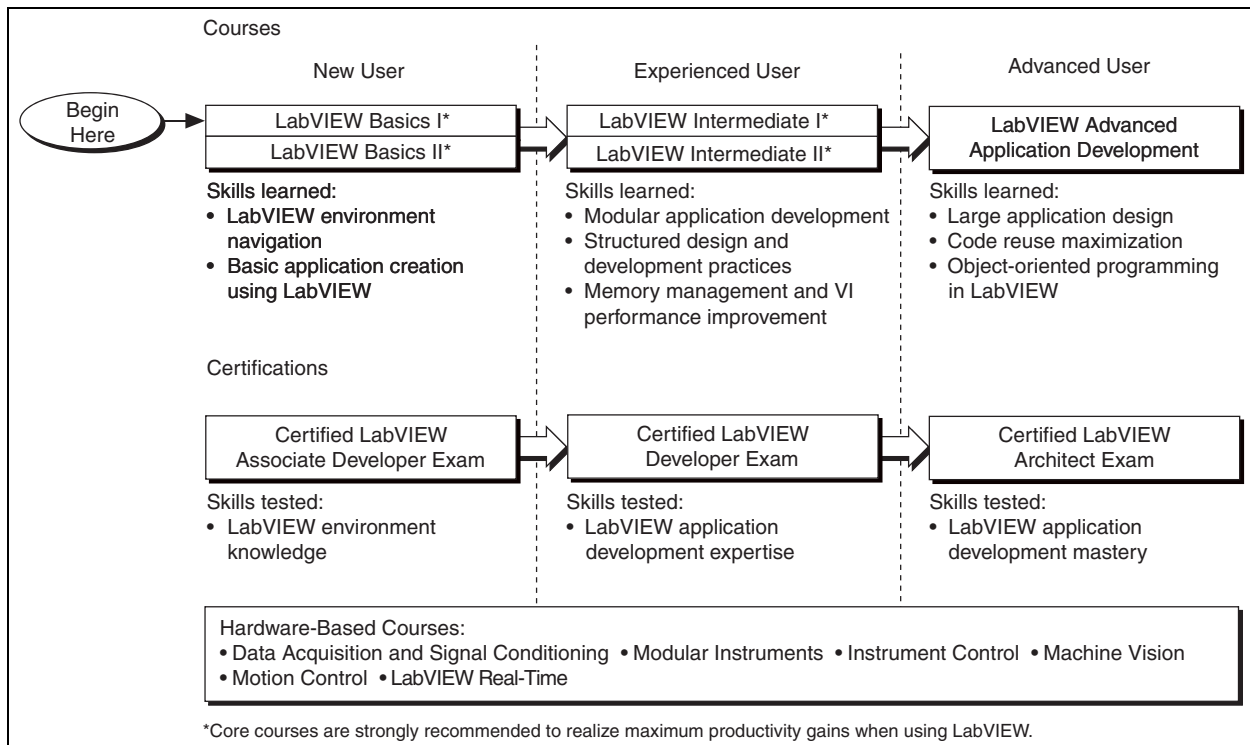
Thank you for purchasing the *LabVIEW Basics I: Introduction* course kit. You can begin developing an application soon after you complete the exercises in this manual. This course manual and the accompanying software are used in the three-day, hands-on *LabVIEW Basics I: Introduction* course.

You can apply the full purchase of this course kit toward the corresponding course registration fee if you register within 90 days of purchasing the kit. Visit ni.com/training for online course schedules, syllabi, training centers, and class registration.



Note For course manual updates and corrections, refer to ni.com/info and enter the info code `rdlvce`.

The *LabVIEW Basics I: Introduction* course is part of a series of courses designed to build your proficiency with LabVIEW and help you prepare for NI LabVIEW certification exams. The following illustration shows the courses that are part of the LabVIEW training series. Refer to ni.com/training for more information about NI Certification.



A. About This Manual

Use this manual to learn about LabVIEW programming concepts, techniques, features, VIs, and functions you can use to create test and measurement, data acquisition, instrument control, datalogging, measurement analysis, and report generation applications. This course manual assumes that you are familiar with Windows and that you have experience writing algorithms in the form of flowcharts or block diagrams.

The course manual is divided into lessons, each covering a topic or a set of topics. Each lesson consists of the following:

- An introduction that describes the purpose of the lesson and what you will learn
- A description of the topics in the lesson
- A set of exercises to reinforce those topics

Some lessons include optional and challenge exercise sections or a set of additional exercises to complete if time permits.

- A summary that outlines important concepts and skills taught in the lesson

Several exercises in this manual use one of the following National Instruments hardware products:

- A plug-in multifunction data acquisition (DAQ) device connected to a DAQ Signal Accessory containing a temperature sensor, function generator, and LEDs
- A GPIB interface connected to an NI Instrument Simulator

If you do not have this hardware, you still can complete the exercises. Alternate instructions are provided for completing the exercises without hardware. Exercises that explicitly require hardware are indicated with an icon, shown at left. You also can substitute other hardware for those previously mentioned. For example, you can use a GPIB instrument in place of the NI Instrument Simulator, or another National Instruments DAQ device connected to a signal source, such as a function generator.



B. What You Need to Get Started

Before you use this course manual, ensure you have all the following items:

- Windows 2000 or later installed on your computer. The course is optimized for Windows XP.
- Multifunction DAQ device configured as device 1 using Measurement & Automation Explorer (MAX)

- DAQ Signal Accessory, wires, and cable
- GPIB interface
- NI Instrument Simulator and power supply
- LabVIEW Full or Professional Development System 8.0 or later
- A serial cable
- A GPIB cable
- LabVIEW Basics I: Introduction* course CD, containing the following files

Filename	Description
Exercises	Folder for saving VIs created during the course and for completing certain course exercises; also includes subVIs necessary for some exercises and zip file (nidevsim.zip) containing the LabVIEW instrument driver for the NI Instrument Simulator
Solutions	Folder containing the solutions to all the course exercises

C. Installing the Course Software

Complete the following steps to install the course software.

1. Insert the course CD in your computer. The **LabVIEW Basics Course Material Setup** dialog box displays.
2. Click the **Next** button.
3. Choose the **Typical** setup type and click the **Install** button to begin the installation.
4. Click the **Finish** button to exit the Setup Wizard.

The installer places the `Exercises` and `Solutions` folders at the top level of the `C :` directory.

Repairing or Removing Course Material

You can repair or remove the course material using the **Add or Remove Programs** feature on the Windows **Control Panel**. Repair the course material to overwrite existing course material with the original, unedited versions of the files. Remove the course material if you no longer need the files on your machine.

D. Course Goals

This course prepares you to do the following:

- Understand front panels, block diagrams, icons, and connector panes
- Use the programming structures and data types that exist in LabVIEW
- Use various editing and debugging techniques
- Create and save VIs so you can use them as subVIs
- Display and log data
- Create applications that use plug-in DAQ devices
- Create applications that use serial port and GPIB instruments

This course does *not* describe any of the following:

- Every built-in VI, function, or object; refer to the *LabVIEW Help* for more information about LabVIEW features not described in this course
- Analog-to-digital (A/D) theory
- Operation of the serial port
- Operation of the GPIB bus
- Developing an instrument driver
- Developing a complete application for any student in the class; refer to the NI Example Finder, available by selecting **Help»Find Examples**, for example VIs you can use and incorporate into VIs you create

E. Course Conventions

The following conventions appear in this course manual:

»

The » symbol leads you through nested menu items and dialog box options to a final action. The sequence **File»Page Setup»Options** directs you to pull down the **File** menu, select the **Page Setup** item, and select **Options** from the last dialog box.



This icon denotes a tip, which alerts you to advisory information.



This icon denotes a note, which alerts you to important information.



This icon denotes a caution, which advises you of precautions to take to avoid injury, data loss, or a system crash.



This icon indicates that an exercise requires a plug-in GPIB interface or DAQ device.

bold	Bold text denotes items that you must select or click in the software, such as menu items and dialog box options. Bold text also denotes parameter names, controls and buttons on the front panel, dialog boxes, sections of dialog boxes, menu names, and palette names.
<i>italic</i>	Italic text denotes variables, emphasis, a cross reference, or an introduction to a key concept. This font also denotes text that is a placeholder for a word or value that you must supply.
monospace	Text in this font denotes text or characters that you should enter from the keyboard, sections of code, programming examples, and syntax examples. This font is also used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, operations, variables, filenames and extensions, and code excerpts.
monospace bold	Text in this font denotes the messages and responses that the computer automatically prints to the screen. This font also emphasizes lines of code that are different from the other examples.
Platform	Text in this font denotes a specific platform and indicates that the text following it applies only to that platform.

Problem Solving

LabVIEW is a programming language you can use to solve various problems. Problem solving skills are essential to creating solutions in LabVIEW. Computer programmers use a software development method to solve problems using software programs. Following a method helps a programmer to develop code that has greater potential to successfully solve a given problem as compared to writing code without a plan. A method also helps to make code more readable, scalable, and modifiable.

The following lesson outlines a software development strategy you can use to solve your measurement and analysis needs.

Topics

- A. Software Development Method
- B. Scenario
- C. Design
- D. Implementation
- E. Testing
- F. Maintenance
- G. Project

A. Software Development Method

Following a set of steps that has been refined over the years by software engineers can simplify solving problems using software. In this course, you will learn a specific set of steps called the software development method. The software development method is a strategy for using LabVIEW to implement a software solution. Use the software development method to create a solution to your problem.

In the software development method, complete the following steps:

1. Define the problem (scenario).
2. Design an algorithm and/or flowchart.
3. Implement the design.
4. Test and verify the implementation.
5. Maintain and update the implementation.

During this course, this software development method serves as a framework for all hands-on development exercises. In most exercises, you receive the scenario and design steps. Then you complete the implementation, testing, and maintenance steps. Most of the instruction in this course helps you create a successful implementation.

Furnace Example—A furnace example in this lesson illustrates each step of the software development method described.

B. Scenario

During this stage of the software development method, you define what your problem is so that you can approach it with all the necessary factors identified. You can remove extraneous factors during this phase and focus on the core problem that you must solve. How you identify the problem initially can save you time while you design and implement a solution.

Furnace Example—Assume that you must cure a material at a certain temperature for a set amount of time in a furnace. For this problem, it is not necessary to know the material type or the time of day. You must know the cure time, cure temperature, and method for adjusting the furnace temperature.

C. Design

After you determine the scope of the problem, you can design a solution by analyzing the problem. Part of analyzing the solution is identifying the inputs and outputs of the software, as well as any additional requirements.

After you define the inputs and outputs, you can design an algorithm, flowchart and/or state transition diagram to help you arrive at a software solution.

Identify the Inputs

The inputs indicate the raw data that you want to process during the problem solving process.

Furnace Example—Inputs for the furnace software are the cure time (seconds), the necessary cure temperature (degrees Kelvin), and the furnace temperature (degrees Kelvin).

Identify the Outputs

The outputs represent the result of the calculation, processing, or other condition that the problem solving process implements.

Furnace Example—The output of the furnace software is an on/off switch that applies voltage to the furnace coil. Voltage is applied to the coil by changing the state of a switch that controls the voltage supply to the coils. When the voltage is applied or removed, the furnace has an immediate change in temperature.

Identifying Additional Requirements

Consider any other factors that might influence solving the problem. For example, do you need to use specific units such as centimeters or seconds?

Furnace Example—As an additional requirement for this example, assume that the furnace can not start until the interior temperature is the same as the exterior temperature.

Designing an Algorithm to Solve the Problem

After determining the inputs, outputs, and additional requirements, you can create an algorithm. An algorithm is a set of steps that process your inputs and create outputs.

Furnace Example—This algorithm describes the operation of the furnace:

1. Read exterior temperature.
2. Read interior temperature.
3. If interior temperature is not equal to exterior temperature, repeat step 1.
4. Read interior temperature.
5. If interior temperature is greater than desired temperature, turn off voltage to coil.

6. If current temperature is less than or equal to desired temperature, turn on voltage to coil.
7. If time is less than cure time, repeat step 4.
8. Turn off voltage to coil.

Designing a Flowchart

A flowchart displays the steps for solving the problem. Flowcharts are useful because you can follow more complex processes of an algorithm in a visual way. For example, you can see if a specific step has two different paths to the end solution and you can plan your code accordingly.

Furnace Example—You can design this example using either an algorithm or a flowchart. Figure 1-1 shows a flowchart following the algorithm designed in the previous subsection.

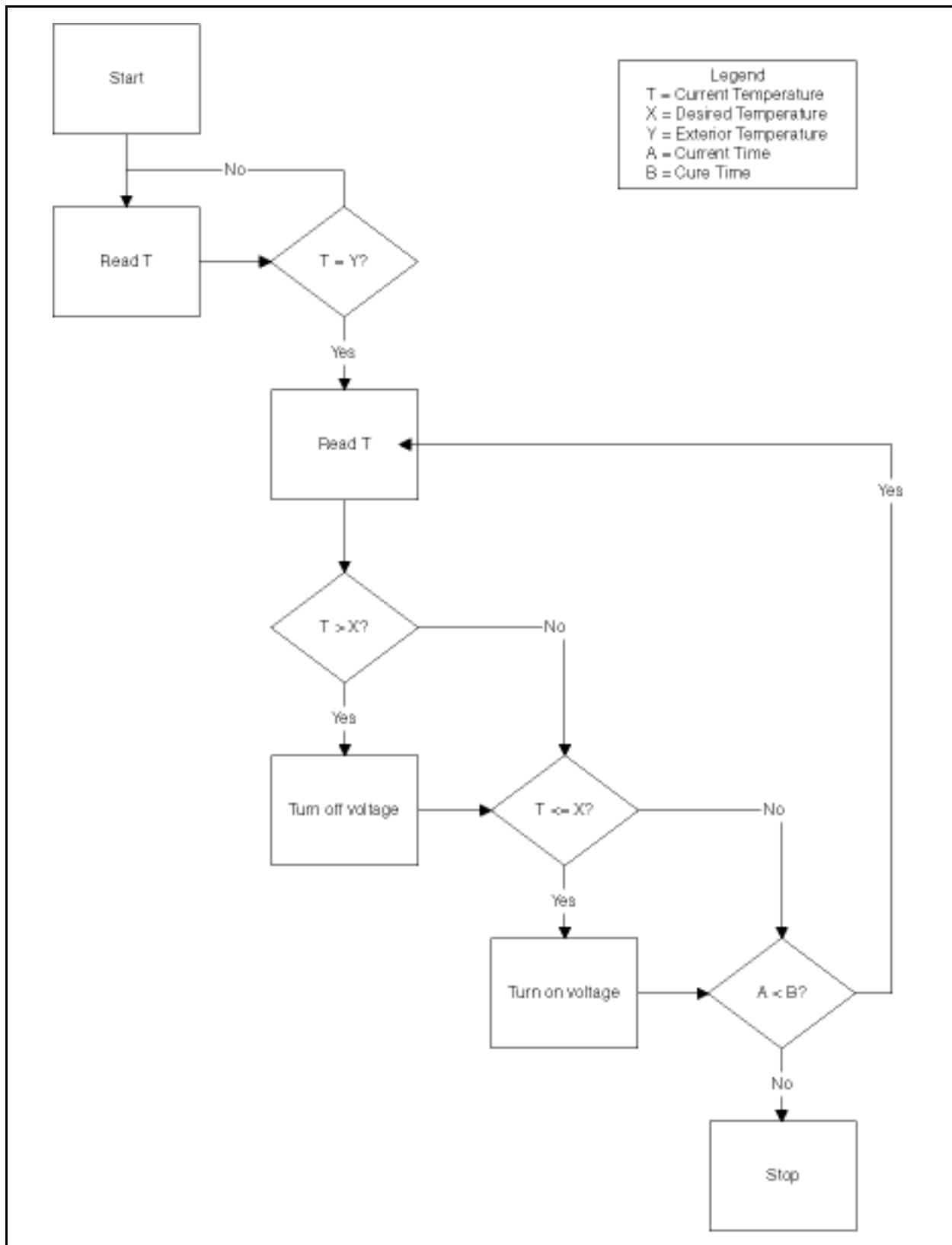


Figure 1-1. Flowchart for Furnace Example

Designing a State Transition Diagram

State transition diagrams are a specific type of flowchart that are commonly used when creating LabVIEW state machines. State transition diagrams allow you to clearly indicate the states of a program and what causes the program to transition from one state to the next. A state transition diagram uses a labeled circle to signify a steady state and a labeled arrow to indicate a transition from a state.

A state is a part of a program that satisfies a condition, performs an action, or waits for an event. A transition is the condition, action or event that causes the program to move to the next state.



The start of the program is signified with a solid circle, as shown at left.



The end of the program is signified with a targeted circle, as shown at left.

Furnace Example—You can also use a state transition diagram for this example. Figure 1-2 shows the furnace example redesigned as a state transition diagram. Both the flowchart and the state transition diagram are valid ways to design a VI, but may lead to a different programming solution.

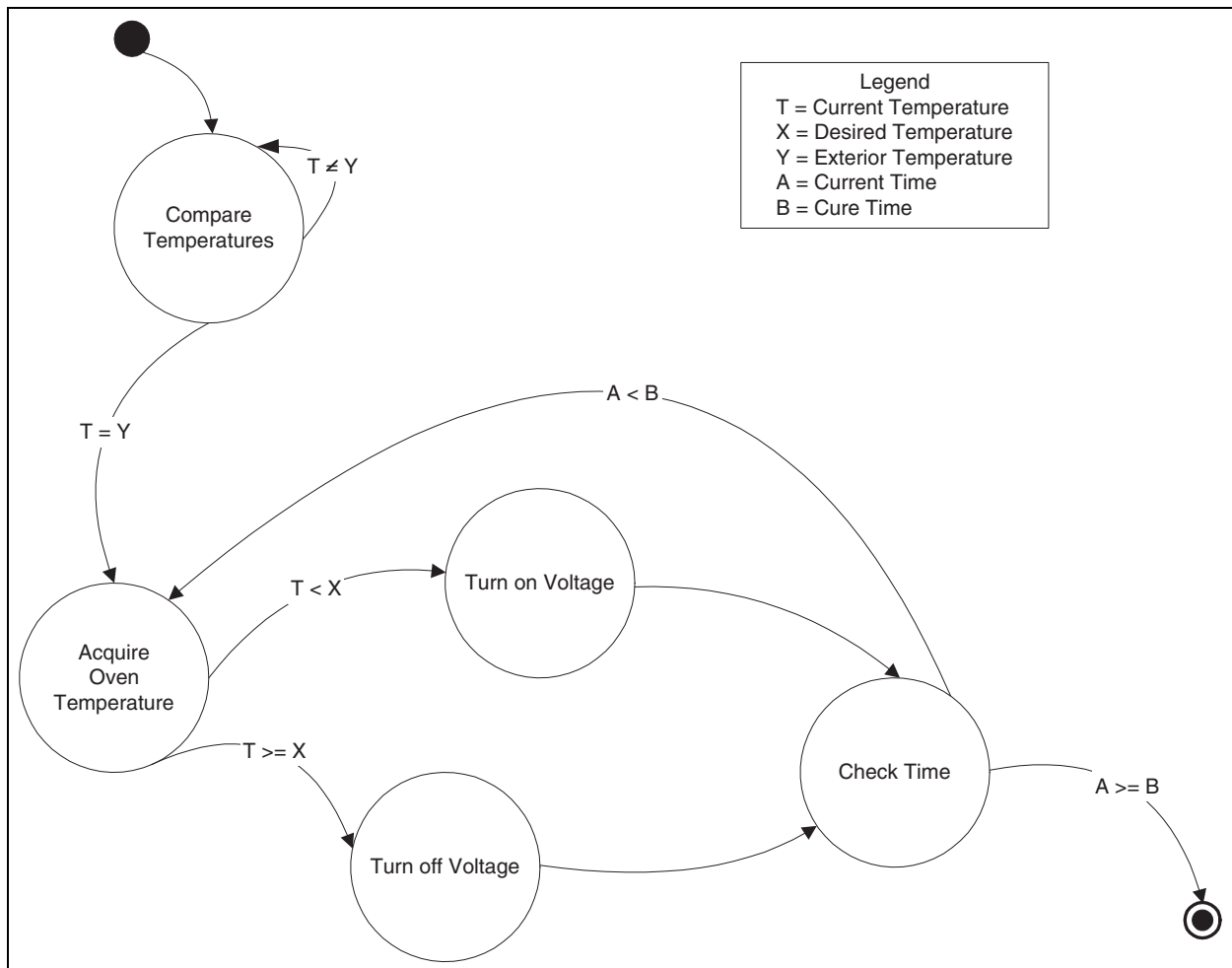


Figure 1-2. State Transition Diagram for Furnace Example

D. Implementation

In the implementation stage, you create code for your algorithm or flowchart. When writing code in a text-based language, the algorithm elegantly translates into each line of code, depending on the level of detail shown in the algorithm. Because LabVIEW is a graphical programming language, the flowchart works much the same way. Refer to Lesson 11, *Common Design Techniques and Patterns* for more information about implementing LabVIEW VIs from a flowchart or state transition diagram.

E. Testing

Testing and verifying is an important part of the software development method. Make sure to test your implementation with data that is both logical and illogical for the solution you created. Testing logical data verifies that the inputs produce the expected result. By testing illogical data, you can test to see if the code has effective error handling.

Furnace Example—To test the error handling strategy of the furnace example, you could input a cure temperature that is less than the ambient temperature. An effective error handling strategy could alert the user that the furnace can only increase temperature, not decrease it.

F. Maintenance

Maintenance is the ongoing process of resolving programming errors adding parallel construction changes to the original solution for a problem.

Furnace Example—After writing this code, you may discover that the customer wants to add a temperature sensor to another area of the oven to add redundancy to the system. Adding features to the program is easier if you plan for scalability in your software from the beginning.

Exercise 1-1 Software Development Method

Goal

Solve a problem using the software development method without using software.

Scenario

You are responsible for displaying the time until arrival for airplanes at an airport. You receive this information in seconds, but must display it as a combination of hours/minutes/seconds.

Design

What inputs are you given?

What outputs are you expected to produce?

What is the relationship/conversion between the inputs and outputs? Use the Windows calculator to help you determine the relationship.

Create an algorithm or flowchart that demonstrates the relationship between the inputs and outputs.

Implementation

During this stage, you implement the program from the algorithm or flowchart. For this exercise, skip this stage. Refer to Exercise 2-1 to see an implementation of a solution to this problem.

Testing

Use a set of known values to test the algorithm or flowchart you designed.

Example inputs with corresponding outputs:

Input	Output
0 seconds	0 hours, 0 minutes, 0 seconds
60 seconds	0 hours, 1 minute, 0 seconds
3600 seconds	1 hour, 0 minutes, 0 seconds
3665 seconds	1 hour, 1 minute, 5 seconds

Maintenance

If a test value set has failed, return to the design phase and check for errors.

End of Exercise 1-1

Exercise 1-2 Project: State Transition Diagram

Goal

Create a state transition diagram.

Scenario

Design a state transition diagram for a program that does the following:

1. Acquires a temperature every half a second
2. Analyzes each temperature to determine if the temperature is too high or too low
3. Alerts the user if there is a danger of heat stroke or freeze
4. Displays the data to the user
5. Logs the data if a warning occurs
6. If the user does not stop the program, the entire process repeats

Design

In the design stage, you determine the inputs and outputs. Design a state transition diagram utilizing the inputs to achieve the required outputs.

Inputs

- Current Temperature (T)
- High Temperature Limit (X)
- Low Temperature Limit (Y)
- Stop

Outputs

- Warning Levels: Heatstroke Warning, No Warning, Freeze Warning
- Current Temperature Display
- Data Log File

State Transition Diagram

Design a state transition diagram that produces the outputs based on the given inputs. Remember to take into account transitions such as whether the time has elapsed before taking the next temperature reading, and whether to log data based on the warning that has occurred.



Note You do not complete the Implementation, Testing and Maintenance stages in this exercise because the goal of the exercise is to complete only the design.

End of Exercise 1-2

G. Project

Throughout this course, the course project illustrates concepts. In Exercise 1-2, you designed the state transition diagram for the course project. The diagram you designed is just one potential design among many.

For this course, one state transition diagram, shown in Figure 1-3, is chosen so that all students may follow the same instruction set. This state transition diagram is chosen because it successfully solves the problem and it has parts that can be effectively used to demonstrate course concepts. However, it may not be the best solution to the problem.

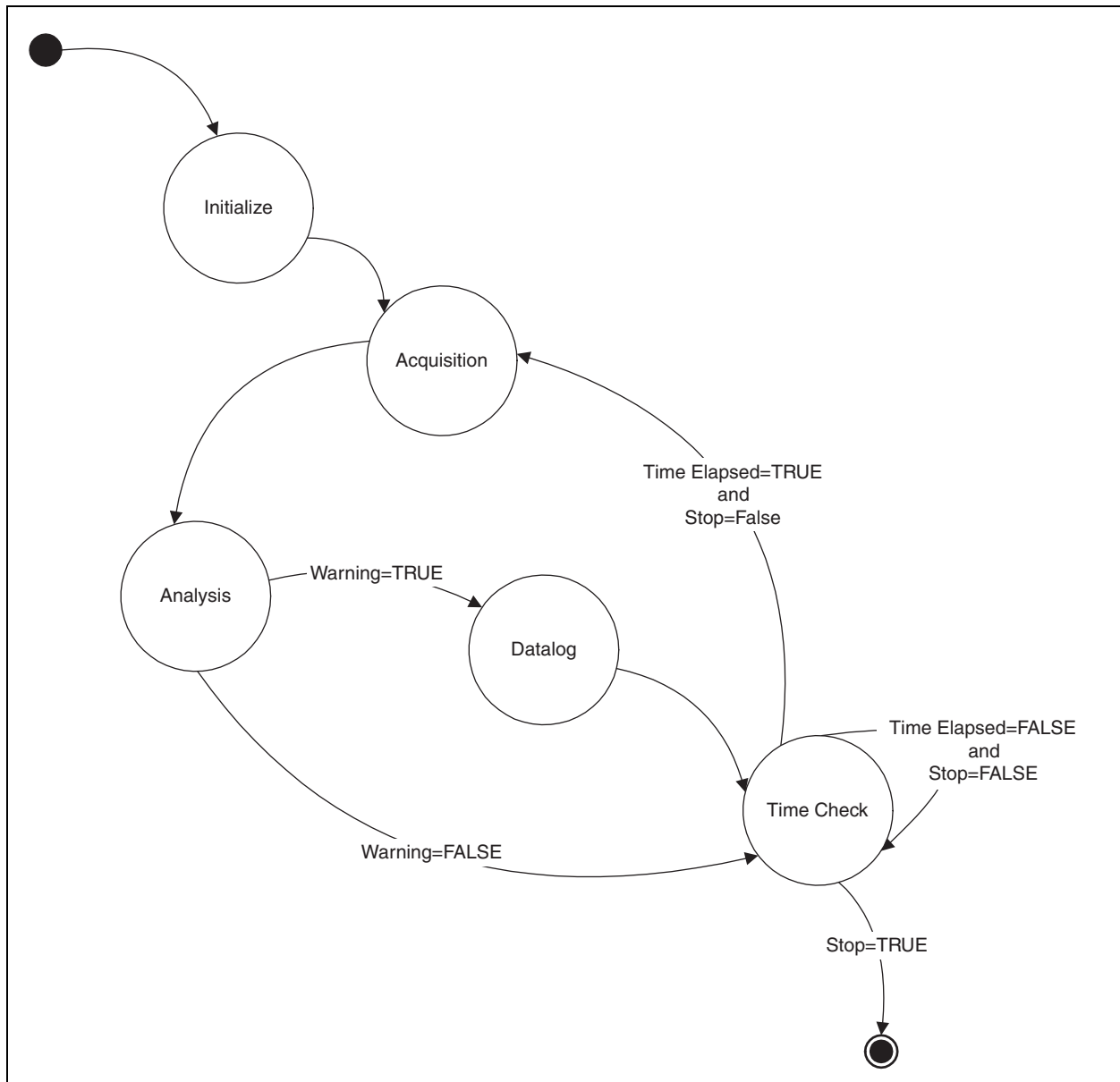


Figure 1-3. Project State Transition Diagram

Figure 1-4 shows an example of an alternate state transition diagram. This state transition diagram also solves the problem very effectively. One of the major differences between these two diagrams is how they can be expanded for future functionality. In the state transition diagram in Figure 1-3, you can modify the diagram to include warning states for other physical phenomena, such as wind, pressure, and humidity. In the state transition diagram in Figure 1-4, you can add other layers of temperature warnings. The possible future changes you expect to your program affect which diagram you choose.

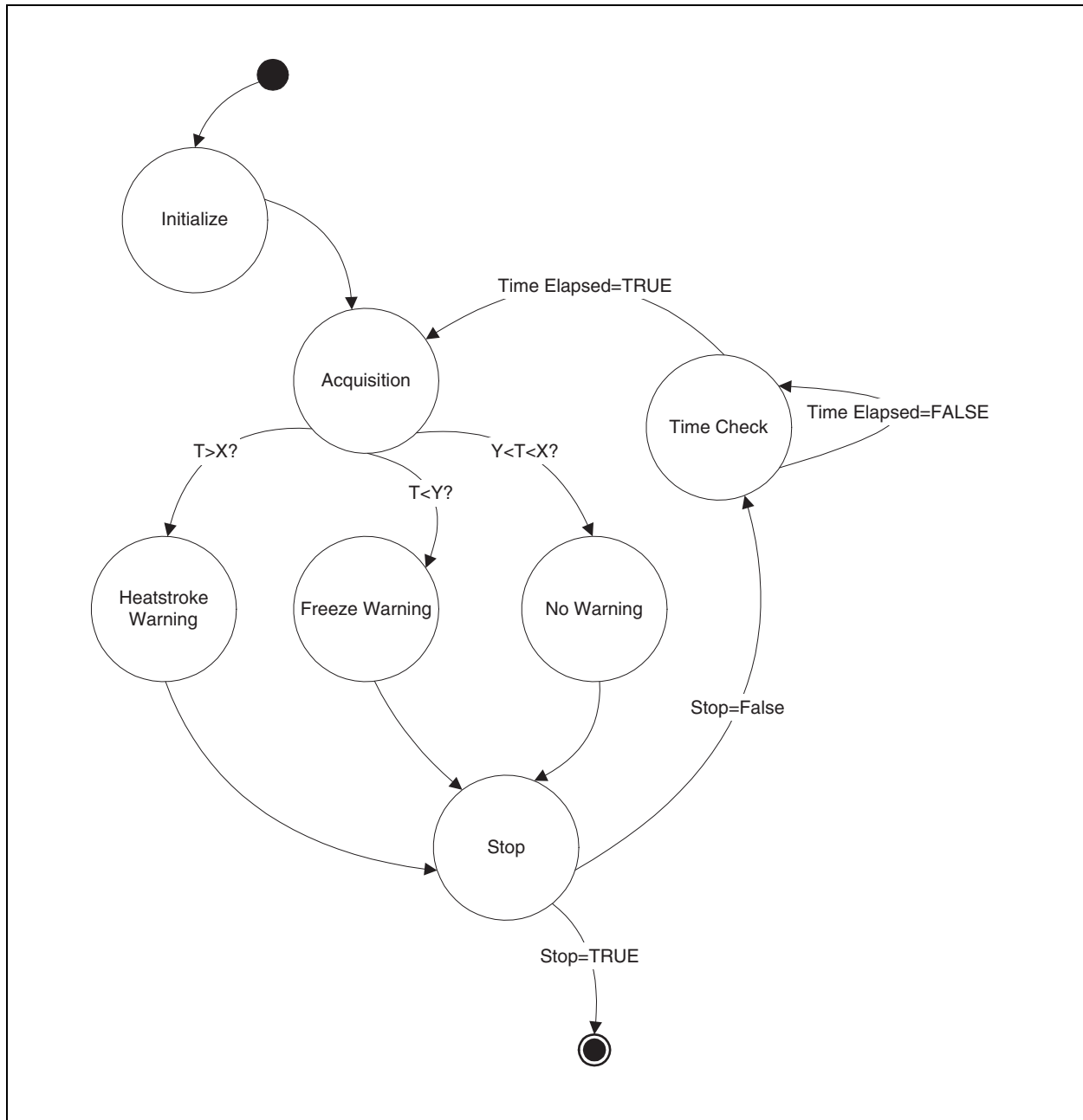


Figure 1-4. Project State Transition Diagram Alternate

Self-Review: Quiz

Match each step of the described software development method to the correct description of the step.

- | | |
|-------------------|------------------------------------|
| 1. Scenario | A. Apply an algorithm or flowchart |
| 2. Design | B. Verify the VI |
| 3. Implementation | C. Define the problem |
| 4. Testing | D. Update the VI |
| 5. Maintenance | E. Identify the inputs and outputs |

Self-Review: Quiz Answers

Match each step of the described software development method to the correct description of the step.

- | | |
|---|--|
| 1 | C. Scenario: Define the problem |
| 2 | E. Design: Identify the inputs and outputs |
| 3 | A. Implementation: Apply an algorithm or flowchart |
| 4 | B. Testing: Verify the VI |
| 5 | D. Maintenance: Update the VI |

Notes

Navigating LabVIEW

This lesson introduces how to navigate the LabVIEW environment. This includes using the menus, toolbars, palettes, tools, help, and common dialog boxes of LabVIEW. You also learn how to run a VI and gain a general understanding of a front panel and block diagram. At the end of this lesson, you build a simple VI that acquires, analyzes, and presents data.

Topics

- A. Virtual Instruments (VIs)
- B. Starting a VI
- C. Parts of a VI
- D. Project Explorer
- E. Front Panel
- F. Block Diagram
- G. Searching for Controls, VIs, and Functions
- H. Selecting a Tool
- I. Dataflow
- J. Building a Simple VI
- K. LabVIEW Help Utilities

A. Virtual Instruments (VIs)

LabVIEW programs are called virtual instruments, or VIs, because their appearance and operation imitate physical instruments, such as oscilloscopes and multimeters. LabVIEW contains a comprehensive set of VIs and functions for acquiring, analyzing, displaying, and storing data, as well as tools to help you troubleshoot your code.

B. Parts of a VI

LabVIEW VIs contain three main components—the front panel, the block diagram, and the icon/connector pane.

Front Panel

The front panel is the user interface for the VI. Figure 2-1 shows an example of a front panel. You build the front panel with controls and indicators, which are the interactive input and output terminals of the VI, respectively.

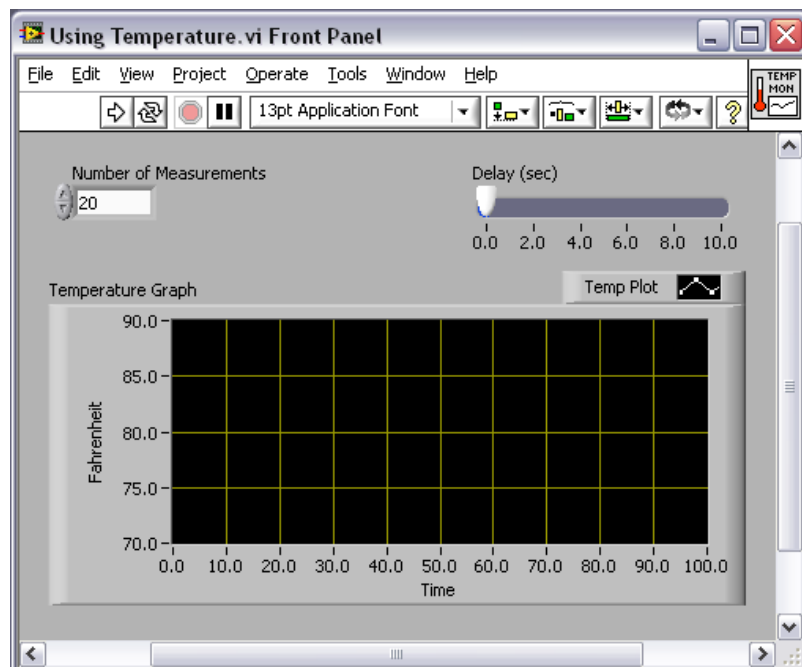


Figure 2-1. VI Front Panel

Block Diagram

After you build the front panel, you add code using graphical representations of functions to control the front panel objects. Figure 2-2 shows an example of a block diagram. The block diagram contains this graphical source code. Front panel objects appear as terminals on the block diagram.

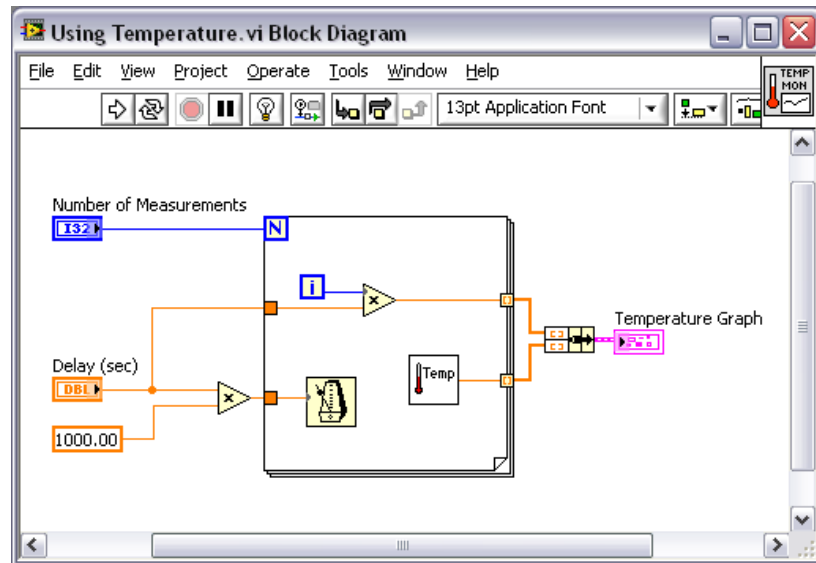


Figure 2-2. Block Diagram

Icon/Connector Pane

You can use a VI as a subVI. A subVI is a VI that is used inside of another VI, similar to a function in a text-based programming language. To use a VI as a subVI, it must have an icon and a connector pane.



Default Icon

Every VI displays an icon, shown at left, in the upper right corner of the front panel and block diagram windows. An icon is a graphical representation of a VI. The icon can contain both text and images. If you use a VI as a subVI, the icon identifies the subVI on the block diagram of the VI. The default icon contains a number that indicates how many new VIs you opened after launching LabVIEW.



To use a VI as a subVI, you need to build a connector pane, shown at left. The connector pane is a set of terminals that corresponds to the controls and indicators of that VI, similar to the parameter list of a function call in text-based programming languages. Access the connector pane by right-clicking the icon in the upper right corner of the front panel window. You cannot access the connector pane from the icon in the block diagram window.

C. Starting a VI

When you launch LabVIEW, the **Getting Started** window appears. Use this window to create new VIs and projects, select among the most recently opened LabVIEW files, find examples, and search the *LabVIEW Help*. You also can access information and resources to help you learn about LabVIEW, such as specific manuals, help topics, and resources at ni.com/manuals.

The **Getting Started** window disappears when you open an existing file or create a new file. You can display the window by selecting **View»Getting Started Window**.

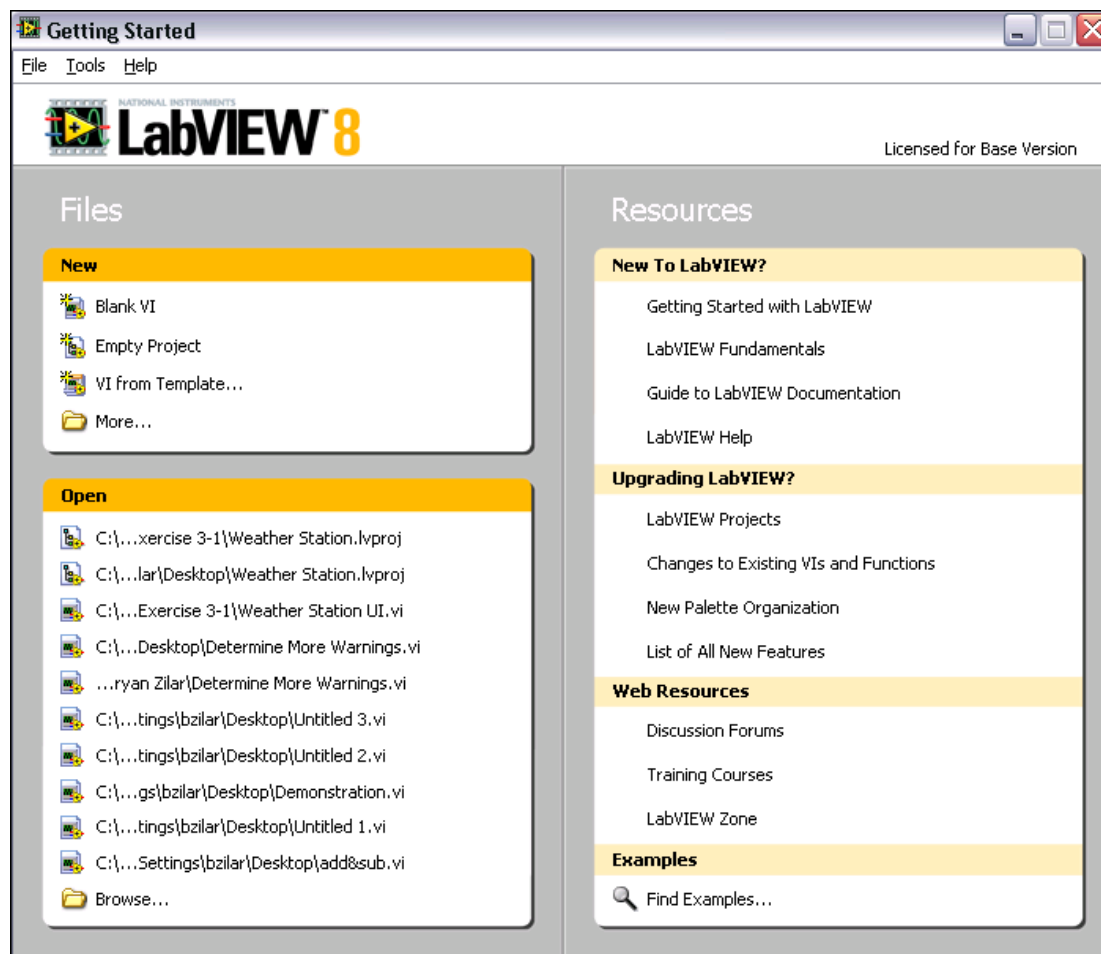


Figure 2-3. The LabVIEW Getting Started Window

You can configure LabVIEW to open a new, blank VI on launch instead of displaying the window. Select **Tools»Options**, select **Environment** from the **Category** list, and place a checkmark in the **Skip Getting Started window on launch** checkbox.

Creating or Opening a VI or Project

You can begin in LabVIEW by starting from a blank VI or project, opening an existing VI or project and modifying it, or opening a template from which to begin your new VI or project.

Starting from Scratch

To open a new project from the **Getting Started** window, select the **Empty Project** option. A new, unnamed project opens, and you can add files to and save the project.

To open a new, blank VI that is not associated with a project, select the **Blank VI** option on the **Getting Started** window.

Create a VI or Project from a Template

Select **File»New** to display the New dialog box, which lists the built-in VI templates. You also can display the New dialog box by clicking the New link in the Getting Started window.

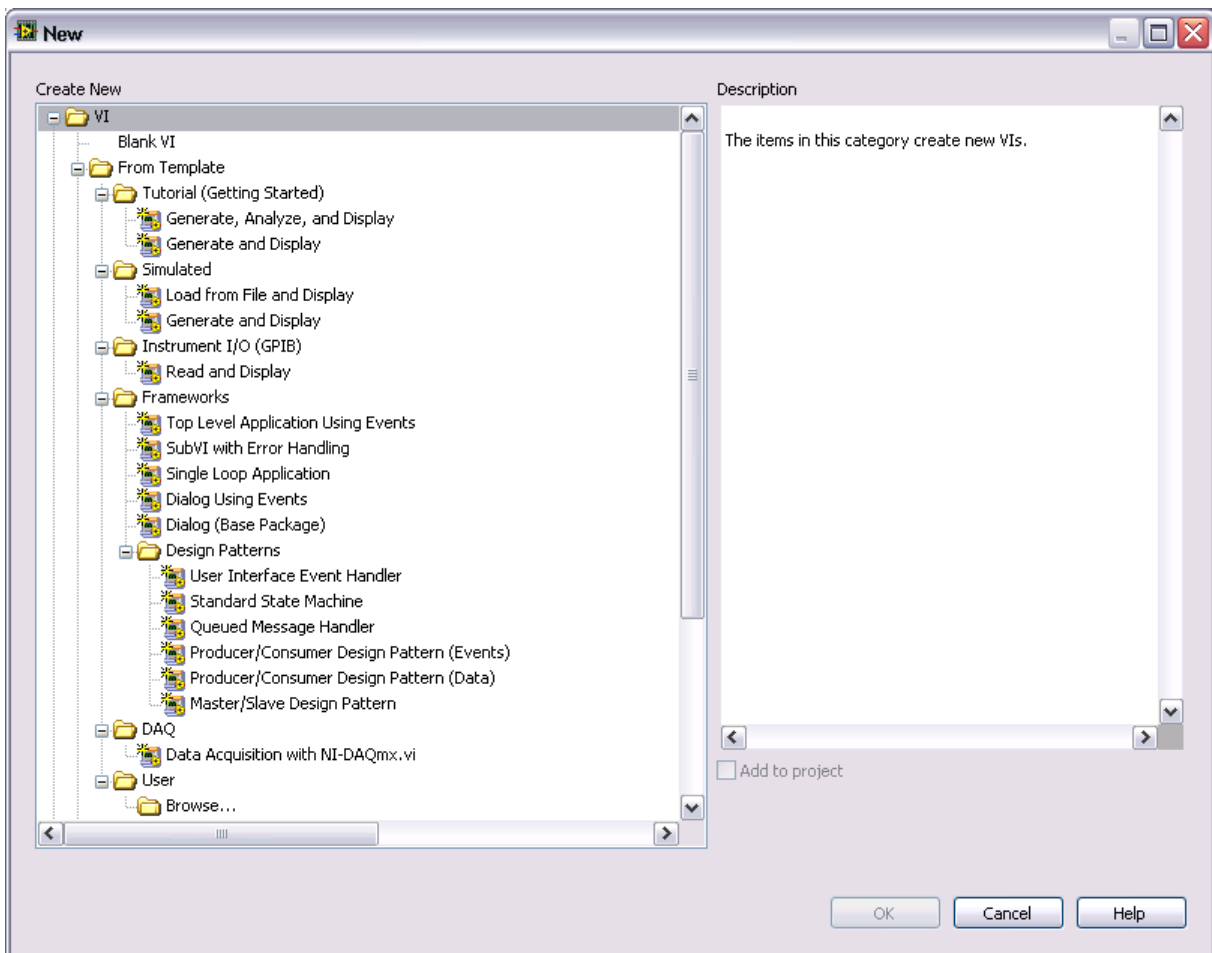


Figure 2-4. New Dialog Box

Opening an Existing VI

Select the **Browse** option in the **Getting Started** window to navigate to and open an existing VI.



Tip The VIs you edit in this course are located in the C:\Exercises\LabVIEW Basics I directory.

As the VI loads, a status dialog box similar to the following example might appear.

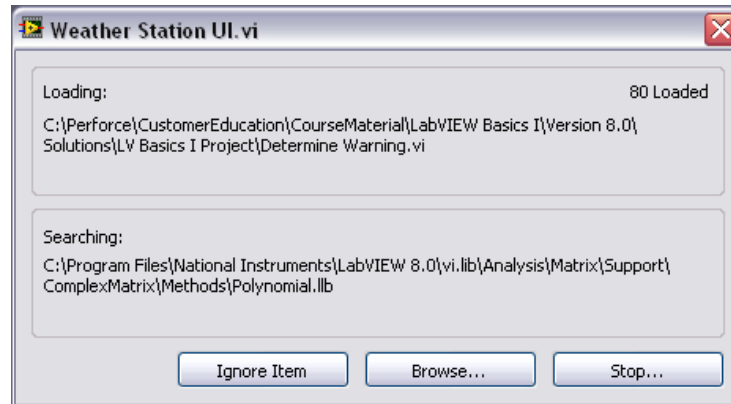


Figure 2-5. Dialog Box Indicating the Status of Loading VIs

The **Loading** section lists the subVIs of the VI as they load into memory and shows the number of subVIs loaded into memory so far. You can cancel the load at any time by clicking the **Stop** button.

If LabVIEW cannot immediately locate a subVI, it begins searching through all directories specified by the VI Search Path. You can edit the VI Search Path by selecting **Tools»Options** and selecting **Paths** from the category list.

You can have LabVIEW ignore a subVI by clicking the **Ignore SubVI** button, or you can click the **Browse** button to search for the missing subVI.

Saving a VI

To save a new VI, select **File»Save**. If you already saved your VI, select **File»Save As** to access the **Save As** dialog box. From the **Save As** dialog box, you can create a copy of the VI, or delete the original VI and replace it with the new one.

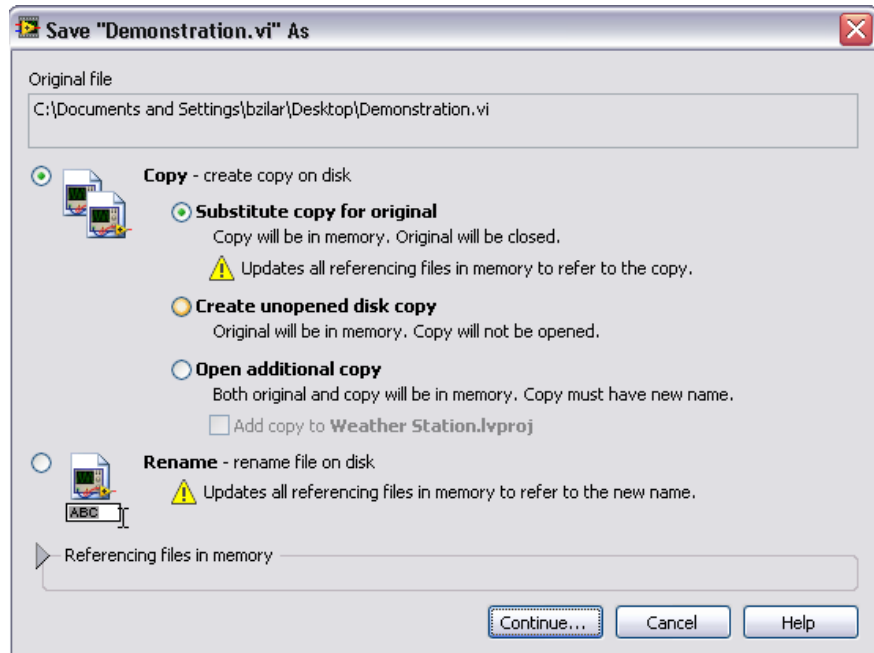


Figure 2-6. Save "<VI Name>" As Dialog Box

D. Project Explorer

Use projects to group together LabVIEW files and non-LabVIEW files, create build specifications, and deploy or download files to targets. When you save a project, LabVIEW creates a project file (.lvproj), which includes references to files in the project, configuration information, build information, deployment information, and so on.

You must use a project to build applications and shared libraries. You also must use a project to work with an RT, FPGA, or PDA target. Refer to the specific module documentation for more information about using projects with the LabVIEW Real-Time, FPGA, and PDA modules.

Project Explorer Window

Use the **Project Explorer** window to create and edit LabVIEW projects. Select **File»New Project** to display the **Project Explorer** window. You also can select **Project»New Project** or select **Empty Project** in the **New** dialog box to display the **Project Explorer** window.

The **Project Explorer** window includes the following items by default:

- **Project root**—Contains all other items in the **Project Explorer** window. This label on the project root includes the filename for the project.
 - **My Computer**—Represents the local computer as a target in the project.
 - **Dependencies**—Includes items that VIs under a target require.
 - **Build Specifications**—Includes build configurations for source distributions and other types of builds available in LabVIEW toolkits and modules. If you have the LabVIEW Professional Development System or Application Builder installed, you can use **Build Specifications** to configure stand-alone applications (EXEs), shared libraries (DLLs), installers, and zip files.



Tip A *target* is any device that can run a VI.

When you add another target to the project, LabVIEW creates an additional item in the **Project Explorer** window to represent the target. Each target also includes Dependencies and Build Specifications. You can add files under each target.

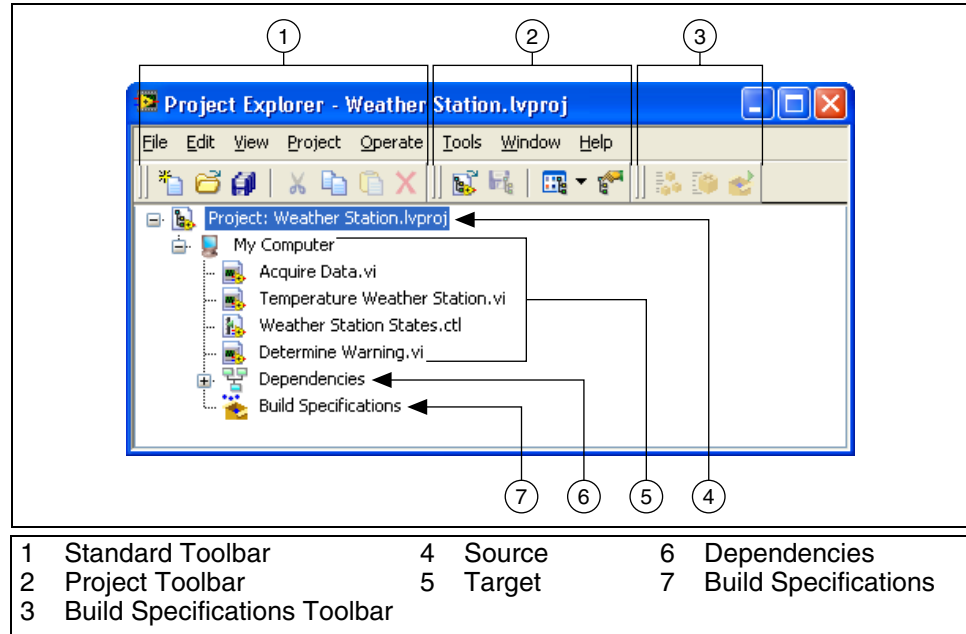


Figure 2-7. Project Explorer Window

Project-Related Toolbars

Use the **Standard**, **Project**, **Build Specifications**, and **Source Control** toolbar buttons to perform operations in a LabVIEW project. The toolbars are available at the top of the **Project Explorer** window, as shown in Figure 2-7. You might need to expand the **Project Explorer** window to view all of the toolbars.



Tip The **Source Control** toolbar is only available if you have Source Control configured in LabVIEW.

You can show or hide toolbars by selecting **View»Toolbars** and selecting the toolbars you want to show or hide. You also can right-click an open area on the toolbar and select the toolbars you want to show or hide.

Creating a LabVIEW Project

Complete the following steps to create a project.

1. Select **File»New Project** to display the **Project Explorer** window. You also can select **Project»New Project** or select **Empty Project** in the **New** dialog box to display the **Project Explorer** window.
2. Add items you want to include in the project under a target.
3. Select **File»Save Project** to save the project.

Adding Existing Files To A Project

You can add existing files to a project. Use the **Source** item in the **Project Explorer** window to add files such as VIs or text files, to a LabVIEW project. An item only can appear once in **Source**. For example, if you add a file from a folder on disk to **Source** and then add the entire folder on disk to **Source**, LabVIEW does not include the file again. Similarly, if an item already appears in **Dependencies**, you cannot add the item to **Source** also.

You can add items to the **Source** of a project in the following ways:

1. Right-click **Source** and select **Add»File** from the shortcut menu to add a file. You also can select **Source** and then select **Project»Add To Project»Add File** to add a file.
2. Right-click **Source** and select **Add»Folder** from the shortcut menu to add a folder. You also can select **Source** and then select **Project»Add To Project»Add Folder** to add a folder. Selecting a folder on disk adds contents of the entire folder, including files and contents of subfolders.



Note After you add a folder on disk to a project, LabVIEW does not automatically update the folder in the project if you make changes to the folder on disk.

3. Right-click **Source** and select **New»VI** from the shortcut menu to add a new, blank VI. You also can select **File»New VI** or **Project»Add To Project»New VI** to add a new, blank VI
4. Select the VI icon in the upper right corner of a front panel or block diagram window and drag the icon to **Source**.
5. **Windows** Select an item or folder from the file system on your computer and drag it to **Source**.

You also can add new LabVIEW files to a project from the **New** dialog box. Select **File»New** or **Project»Add To Project»New** to display the **New** dialog box. In the **New** dialog box, select the item you want to add and place a checkmark in the **Add to project** checkbox.

Removing Items from a Project

You can remove items from the **Project Explorer** window in the following ways:

- Right-click the item you want to remove and select **Remove** from the shortcut menu.
- Select the item you want to remove and press <Delete>.
- Select the item you want to remove and click the **Delete** button on the Standard toolbar.



Note Removing an item from a project does not delete the item on disk.

Organizing Items in a Project

Use folders to organize items in the Source and System Definition in the Project Explorer window. Right-click the Source or System Definition and select **New»Folder** from the shortcut menu to add a new folder. You also can create a new subfolder by right-clicking an existing folder and selecting **New»Folder** from the shortcut menu.

You can arrange items in a folder. Right-click a folder and select **Arrange By»Name** from the shortcut menu to arrange items in alphabetical order. Right-click a folder and select **Arrange By»Type** from the shortcut menu to arrange items by file type.

Viewing Files in a Project

When you add a file to a LabVIEW project, LabVIEW includes a reference to the file on disk. Right-click a file in the **Project Explorer** window and select **Open** from the shortcut menu to open the file in its default editor.

Right-click the project and select **View»Full Paths** from the shortcut menu to view where files that a project references are saved on disk.

Use the **Project File Information** dialog box to view where files that a project references are located on disk and in the **Project Explorer** window. Select **Project»File Information** to display the **Project File Information** dialog box. You also can right-click the project and select **View»File Information** from the shortcut menu to display the **Project File Information** dialog box.

Saving a Project

You can save a LabVIEW project in the following ways:

- Select **File»Save Project**.
- Select **Project»Save Project**.
- Right-click the project and select **Save** from the shortcut menu.
- Click the **Save Project** button on the Project toolbar.

You must save new, unsaved files in a project before you can save the project. When you save a project, LabVIEW does not save dependencies as part of the project file.



Note Make a backup copy of a project when you prepare to make major revisions to the project.

E. Front Panel

When you open a new or existing VI, the front panel of the VI appears. The front panel is the user interface for the VI. Figure 2-8 shows an example of a front panel.

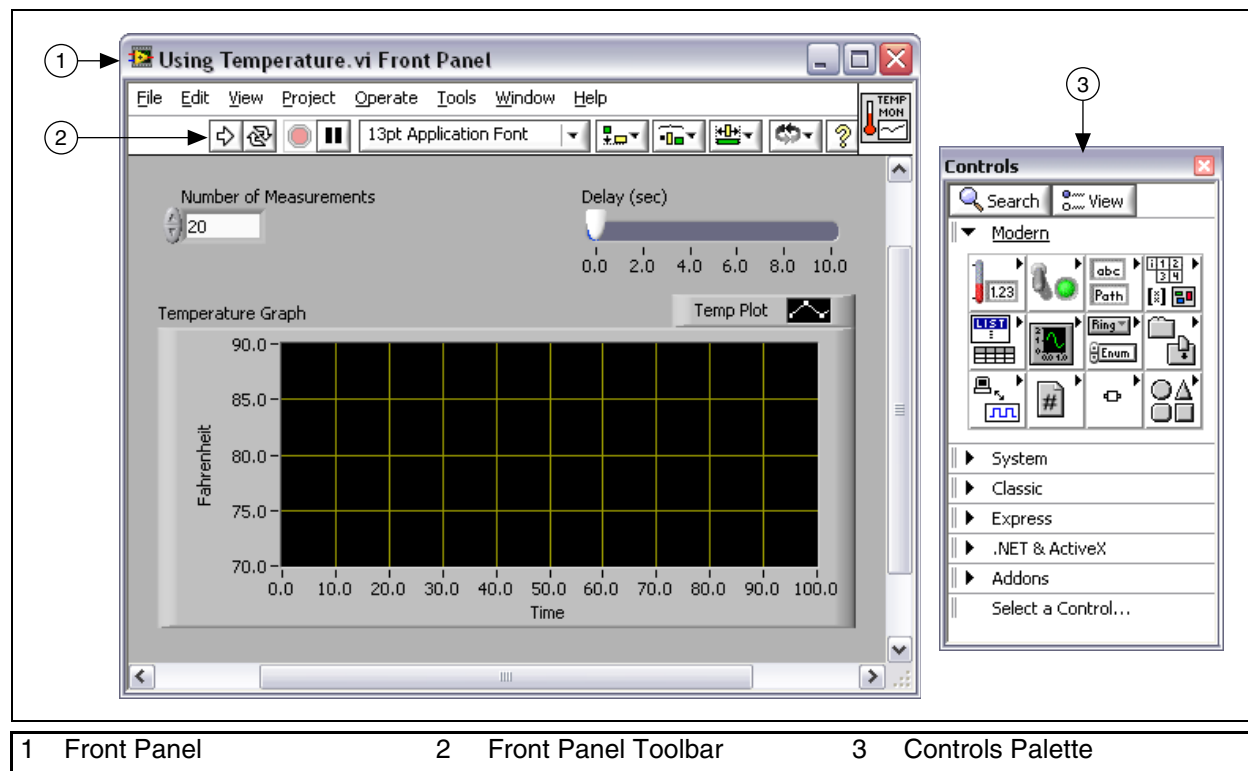


Figure 2-8. Example of a Front Panel

Controls and Indicators

You build the front panel with controls and indicators, which are the interactive input and output terminals of the VI, respectively. Controls are knobs, push buttons, dials, and other input devices. Indicators are graphs, LEDs and other displays. Controls simulate instrument input devices and supply data to the block diagram of the VI. Indicators simulate instrument output devices and display data the block diagram acquires or generates.

Figure 2-8 has the following controls: two controls: **Number of Measurements** and **Delay(sec)**. It has one indicator: an XY graph named **Temperature Graph**.

The user can change the input value for the Number of Measurements and Delay(sec). The user can see the value generated by the VI on the Temperature Graph. The VI generates the values for the indicators based on the code created on the block diagram. You learn about the block diagram in the next section.

Every control or indicator has a data type associated with it. For example, the Delay (sec) horizontal slide is a numeric data type. The most commonly used data types are numeric, Boolean value and string. You learn about other data types in Lesson 3.

Numeric Controls and Indicators

The numeric data type can represent numbers of various types, such as integer or real. The two common numeric objects are the numeric control and the numeric indicator, as shown in Figure 2-9. Objects such as the meter and the dial also represent numeric data.

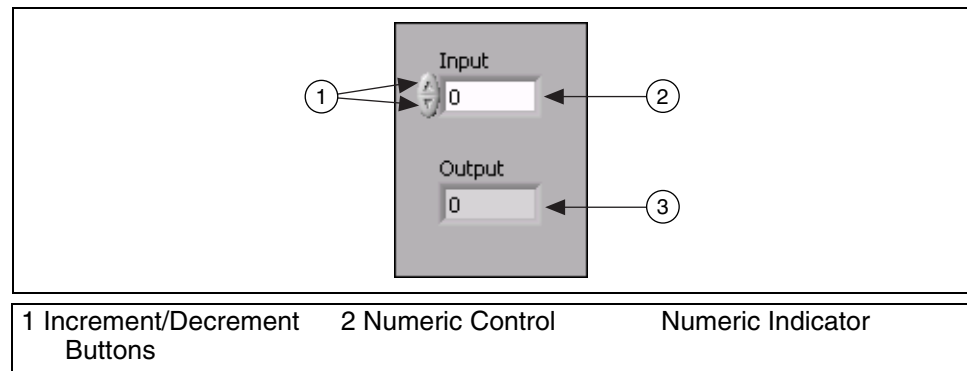


Figure 2-9. Numeric Controls and Indicators

To enter or change values in a numeric control, click the increment and decrement buttons with the Operating tool or double-click the number with either the Labeling tool or the Operating tool, enter a new number, and press the <Enter> key.

Boolean Controls and Indicators

The Boolean data type represents data that only has two parts, such as TRUE and FALSE or ON and OFF. Use Boolean controls and indicators to enter and display Boolean (True or False) values. Boolean objects simulate switches, push buttons, and LEDs. The vertical toggle switch and the round LED Boolean objects are shown in Figure 2-10.

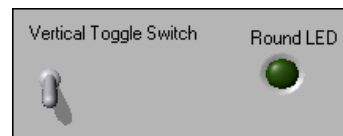


Figure 2-10. Boolean Controls and Indicators

String Controls and Indicators

The string data type is a sequence of ASCII characters. Use string controls to receive text from the user such as a password or user name. Use string

indicators to display text to the user. The most common string objects are tables and text entry boxes as shown in Figure 2-11.

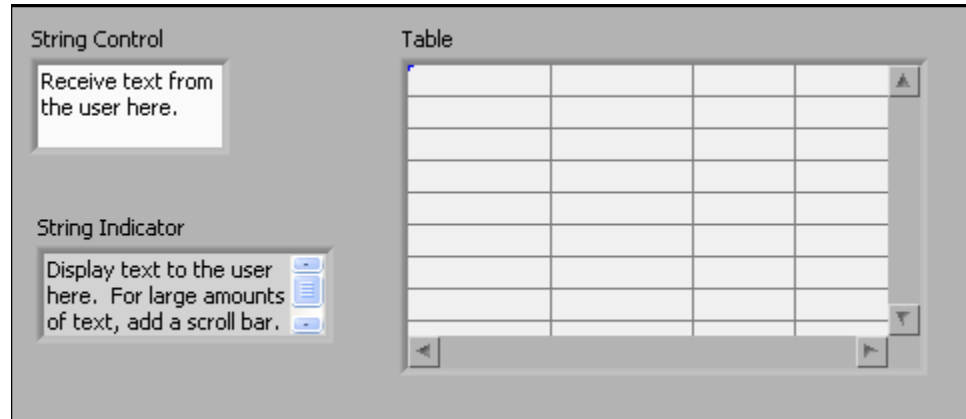


Figure 2-11. String Controls and Indicators

Controls Palette

The **Controls** palette contains the controls and indicators you use to create the front panel. You access the **Controls** palette from the front panel by selecting **View»Controls Palette**. The **Controls** palette is broken into various categories; you can expose some or all of these categories to suit your needs. Figure 2-12 shows a **Controls** palette with all of the categories exposed and the **Modern** category expanded. During this course, you work exclusively in the **Modern** category.

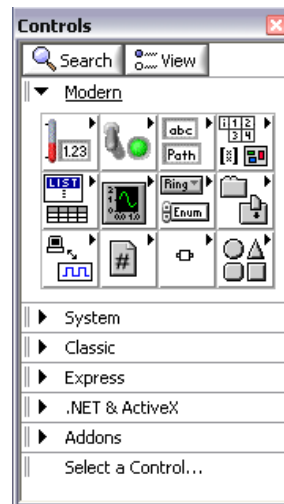


Figure 2-12. Controls Palette

To view or hide categories (subpalettes), select the **View** button on the palette, and select or deselect in the **Always Visible Categories** option. You learn more about using the **Controls** palette in Exercise 2-2.

Shortcut Menus

All LabVIEW objects have associated shortcut menus. As you create a VI, use the shortcut menu items to change the appearance or behavior of front panel and block diagram objects. To access the shortcut menu, right-click the object.

Figure 2-13 shows a shortcut menu for a meter.

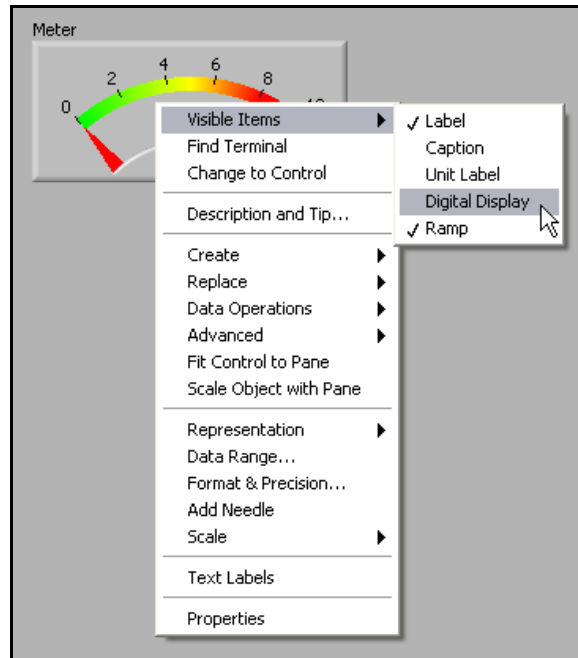


Figure 2-13. Shortcut Menu for a Meter

Property Dialog Boxes

Front panel objects also have property dialog boxes that you can use to change the look or behavior of front panel objects. Right-click a front panel object and select **Properties** from the shortcut menu to access the property dialog box for an object. The following figure shows the property dialog box for the meter shown in the previous figure. The options available on the property dialog box for an object are similar to the options available on the shortcut menu for that object.

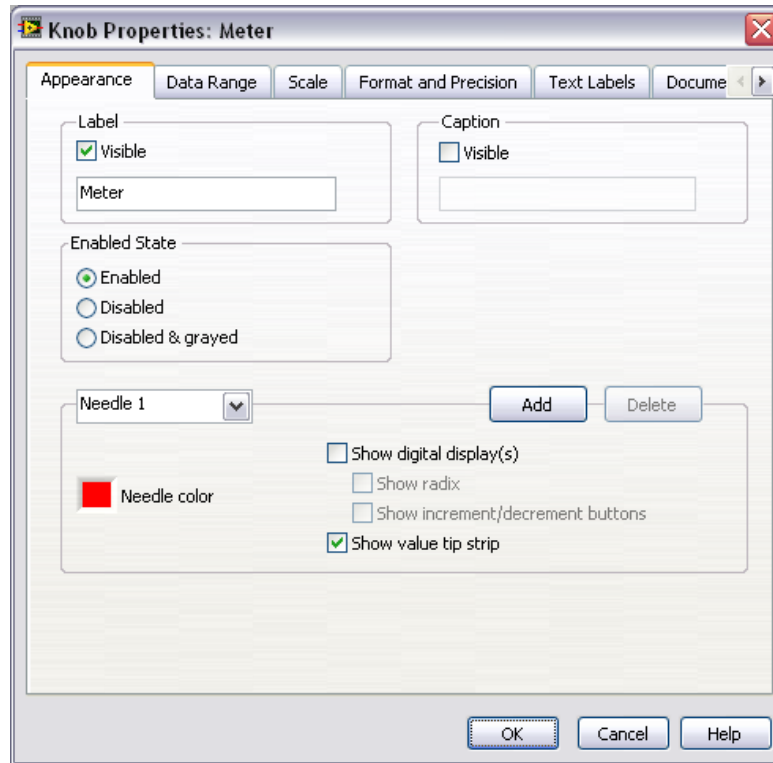
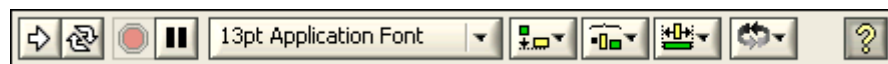


Figure 2-14. Property Dialog Box for a Meter

Front Panel Toolbar

Each window has a toolbar associated with it. Use the front panel toolbar buttons to run and edit the VI.

The following toolbar appears on the front panel.



Click the **Run** button to run a VI. LabVIEW compiles the VI, if necessary. You can run a VI if the Run button appears as a solid white arrow, shown at left. The solid white arrow also indicates you can use the VI as a subVI if you create a connector pane for the VI.



While the VI runs, the **Run** button appears as shown at left if the VI is a top-level VI, meaning it has no callers and therefore is not a subVI.



If the VI that is running is a subVI, the **Run** button appears as shown at left.



The **Run** button appears broken, shown at left, when the VI you are creating or editing contains errors. If the **Run** button still appears broken after you finish wiring the block diagram, the VI is broken and cannot run. Click this button to display the **Error list** window, which lists all errors and warnings.



Click the **Run Continuously** button, shown at left, to run the VI until you abort or pause execution. You also can click the button again to disable continuous running.



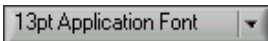
While the VI runs, the **Abort Execution** button, shown at left, appears. Click this button to stop the VI immediately if there is no other way to stop the VI. If more than one running top-level VI uses the VI, the button is dimmed.



Caution The Abort Execution button stops the VI immediately, before the VI finishes the current iteration. Aborting a VI that uses external resources, such as external hardware, might leave the resources in an unknown state by not resetting or releasing them properly. Design VIs with a stop button to avoid this problem.



Click the **Pause** button, shown at left, to pause a running VI. When you click the **Pause** button, LabVIEW highlights on the block diagram the location where you paused execution, and the **Pause** button appears red. Click the **Pause** button again to continue running the VI.



Select the **Text Settings** pull-down menu, shown at left, to change the font settings for the selected portions of the VI, including size, style, and color.



Select the **Align Objects** pull-down menu, shown at left, to align objects along axes, including vertical, top edge, left, and so on.



Select the **Distribute Objects** pull-down menu, shown at left, to space objects evenly, including gaps, compression, and so on.



Select the **Resize Objects** pull-down menu, shown at left, to resize multiple front panel objects to the same size.



Select the **Reorder** pull-down menu, shown at left, when you have objects that overlap each other and you want to define which one is in front or back of another. Select one of the objects with the Positioning tool and then select from **Move Forward**, **Move Backward**, **Move To Front**, and **Move To Back**.



Select the **Show Context Help Window** button, shown at left, to toggle the display of the **Context Help** window.



Type appears to remind you that a new value is available to replace an old value. The Enter button disappears when you click it, press the <Enter> key or click the front panel or block diagram workspace.

F. Block Diagram

Block diagram objects include terminals, subVIs, functions, constants, structures, and wires, which transfer data among other block diagram objects.

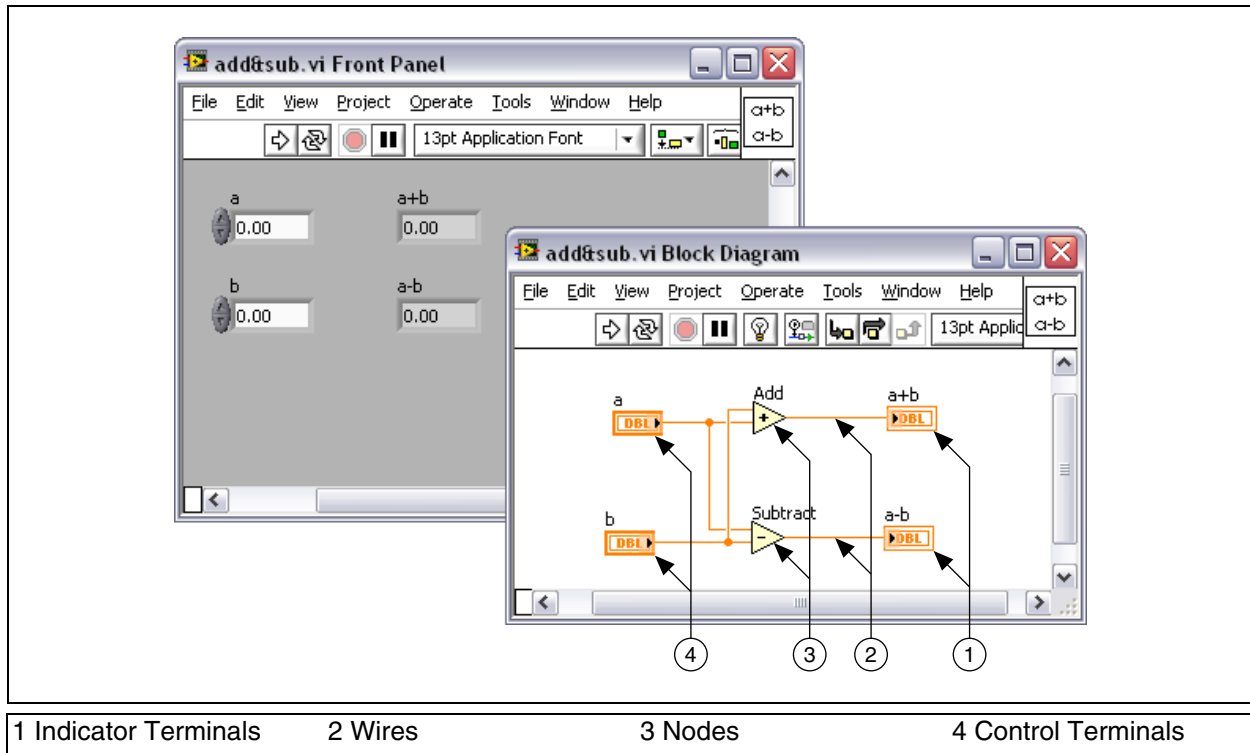


Figure 2-15. Example of a Block Diagram and Corresponding Front Panel

Terminals

Front panel objects appear as terminals on the block diagram. Terminals are entry and exit ports that exchange information between the front panel and block diagram. Terminals are analogous to parameters and constants in text-based programming languages. Types of terminals include control or indicator terminals and node terminals. Control and indicator terminals belong to front panel controls and indicators. Data you enter into the front panel controls (a and b in the previous figure) enter the block diagram through the control terminals. The data then enter the Add and Subtract functions. When the Add and Subtract functions complete their calculations, they produce new data values. The data values flow to the indicator terminals, where they update the front panel indicators (a+b and a-b in the previous figure).



The terminals in Figure 2-15 belong to four front panel controls and indicators. The connector panes of the Add and Subtract functions, shown at left, have three node terminals. To display the terminals of the function

on the block diagram, right-click the function node and select **Visible Items»Terminals** from the shortcut menu.

The terminals represent the data type of the control or indicator. You can configure front panel controls or indicators to appear as icon or data type terminals on the block diagram. By default, front panel objects appear as icon terminals. For example, a knob icon terminal, shown as follows, represents a knob on the front panel.



The DBL at the bottom of the terminal represents a data type of double-precision, floating-point numeric. A DBL terminal, shown as follows, represents a double-precision, floating-point numeric control.



To display a terminal as a data type on the block diagram, right-click the terminal and select **View As Icon** from the shortcut menu to remove the checkmark.

Block Diagram Nodes

Nodes are objects on the block diagram that have inputs and/or outputs and perform operations when a VI runs. They are analogous to statements, operators, functions, and subroutines in text-based programming languages. Nodes can be functions, subVIs, or structures. Structures are process control elements, such as Case structures, For Loops, or While Loops. The Add and Subtract functions in the previous figure are function nodes.

Functions

Functions are the fundamental operating elements of LabVIEW. Functions do not have front panels or block diagrams but do have connector panes. Double-clicking a function only selects the function. A function has a pale yellow background on its icon.

SubVIs

SubVIs are VIs that you build to use inside of another VI or that you access on the Functions palette.

Any VI has the potential to be used as a subVI. When you double-click a subVI on the block diagram, its front panel and block diagram appear. The front panel includes controls and indicators. The block diagram includes

wires, front panel icons, functions, possibly subVIs, and other LabVIEW objects. The upper right corner of the front panel and block diagram displays the icon for the VI. This is the icon that appears when you place the VI on a block diagram as a subVI.

SubVIs also can be Express VIs. Express VIs are nodes that require minimal wiring because you configure them with dialog boxes. Use Express VIs for common measurement tasks. You can save the configuration of an Express VI as a subVI. Refer to the *Express VIs* topic of the *LabVIEW Help* for more information about creating a subVI from an Express VI configuration.

LabVIEW uses colored icons to distinguish between Express VIs, and other VIs on the block diagram. Icons for Express VIs appear on the block diagram as icons surrounded by a blue field whereas subVI icons have a yellow field.

Expandable Nodes versus Icons

You can display VIs and Express VIs as icons or as expandable nodes. Expandable nodes appear as icons surrounded by a colored field. SubVIs appear with a yellow field, and Express VIs appear with a blue field. Use icons if you want to conserve space on the block diagram. Use expandable nodes to make wiring easier and to aid in documenting block diagrams. By default, subVIs appear as icons on the block diagram, and Express VIs appear as expandable nodes. To display a subVI or Express VI as an expandable node, right-click the subVI or Express VI and remove the checkmark next to the **View As Icon** shortcut menu item.

You can resize the expandable node to make wiring even easier, but it also takes a large amount of space on the block diagram. Complete the following steps to resize a node on the block diagram.

1. Move the Positioning tool over the node. Resizing handles appear at the top and bottom of the node.
2. Move the cursor over a resizing handle to change the cursor to the resizing cursor.
3. Use the resizing cursor to drag the border of the node down to display additional terminals.
4. Release the mouse button.

To cancel a resizing operation, drag the node border past the block diagram window before you release the mouse button.

The following figure shows the Basic Function Generator VI as a resized expandable node.

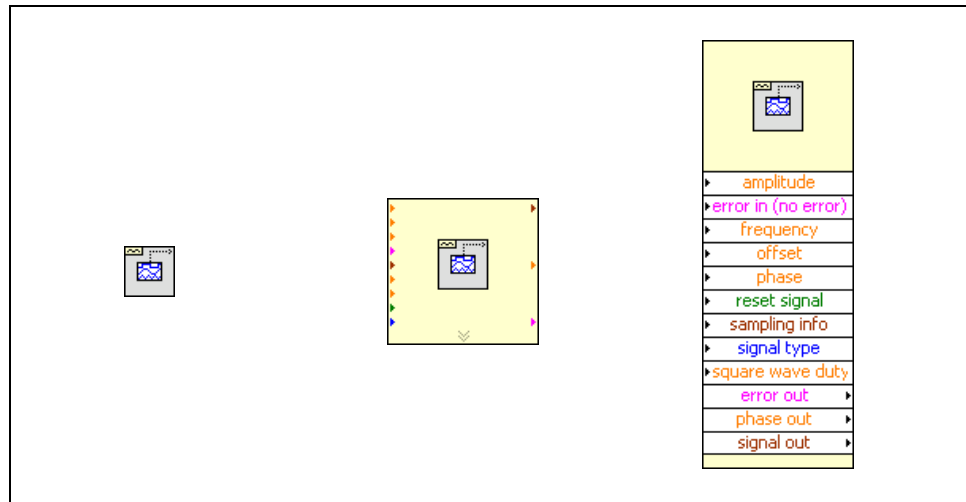


Figure 2-16. Basic Function Generator VI in Different Display Modes



Note If you display a subVI or Express VI as an expandable node, you cannot display the terminals for that node and you cannot enable database access for that node.













Wires

You transfer data among block diagram objects through wires. In Figure, wires connect the control and indicator terminals to the Add and Subtract function. Each wire has a single data source, but you can wire it to many VIs and functions that read the data. Wires are different colors, styles, and thicknesses, depending on their data types.



A broken wire appears as a dashed black line with a red X in the middle, as shown at left. Broken wires occur for a variety of reasons, such as when you try to wire two objects with incompatible data types.

The following examples are the most common wire types.

Wire Type	Scalar	1D Array	2D Array	Color
Numeric	 	 	 	Orange (floating-point), Blue (integer)
Boolean				Green
String				Pink

In LabVIEW, you use wires to connect multiple terminals together to pass data in a VI. You must connect the wires to inputs and outputs that are compatible with the data that is transferred with the wire. For example, you cannot wire an array output to a numeric input. In addition the direction of

the wires must be correct. You must connect the wires to only one input and at least one output. For example, you cannot wire two indicators together. The components that determine wiring compatibility include the data type of the control and/or the indicator and the data type of the terminal.

Data Types

Data types indicate what objects, inputs, and outputs you can wire together. For example, if a switch has a green border, you can wire a switch to any input with a green label on an Express VI. If a knob has an orange border, you can wire a knob to any input with an orange label. However, you cannot wire an orange knob to an input with a green label. Notice the wires are the same color as the terminal.

Automatically Wiring Objects

As you move a selected object close to other objects on the block diagram, LabVIEW draws temporary wires to show you valid connections. When you release the mouse button to place the object on the block diagram, LabVIEW automatically connects the wires. You also can automatically wire objects already on the block diagram. LabVIEW connects the terminals that best match and does not connect the terminals that do not match.

Toggle automatic wiring by pressing the space bar while you move an object using the Positioning tool.

By default, automatic wiring is enabled when you select an object from the **Functions** palette or when you copy an object already on the block diagram by pressing the <Ctrl> key and dragging the object. Automatic wiring is disabled by default when you use the Positioning tool to move an object already on the block diagram.

You can adjust the automatic wiring settings by selecting **Tools»Options** and selecting **Block Diagram** from the top pull-down menu.

Manually Wiring Objects

When you pass the Wiring tool over a terminal, a tip strip appears with the name of the terminal. In addition, the terminal blinks in the **Context Help** window and on the icon to help you verify that you are wiring to the correct terminal. To wire objects together, pass the Wiring tool over the first terminal, click, pass the cursor over the second terminal, and click again. After wiring, you can right-click the wire and select **Clean Up Wire** from the shortcut menu to have LabVIEW automatically choose the a path for the wire. If you have broken wires to remove, press <Ctrl>-B to delete all the broken wires on the block diagram.

Functions Palette

The **Functions** palette contains the VIs, functions and constants you use to create the block diagram. You access the **Functions** palette from the block diagram by selecting **View»Functions Palette**. The **Functions** palette is broken into various categories; you can show and hide categories to suit your needs. Figure 2-17 shows a **Functions** palette with all of the categories exposed and the **Programming** category expanded. During this course, you work mostly in the **Programming** category, but you also use other categories, or subpalettes.

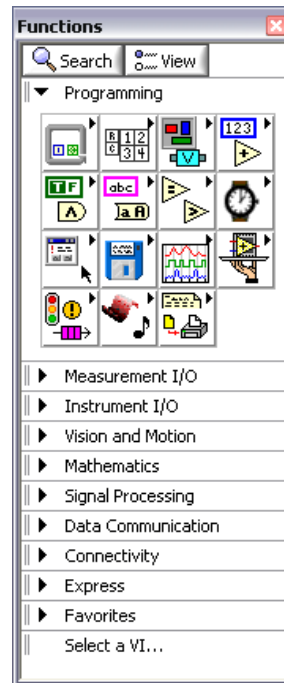


Figure 2-17. Functions Palette

To view or hide categories, select the **View** button on the palette, and select or deselect in the **Always Visible Categories** option. You learn more about using the **Functions** palette in Exercise 2-2.

Block Diagram Toolbar

When you run a VI, buttons appear on the block diagram toolbar that you can use to debug the VI. The following toolbar appears on the block diagram.



Click the **Highlight Execution** button, shown at left, to display an animation of the block diagram execution when you click the **Run** button.

Notice the flow of data through the block diagram. Click the button again to disable execution highlighting.



Click the **Retain Wire Values** button, shown at left, to save the wire values at each point in the flow of execution so that when you place a probe on the wire, you can immediately retain the most recent value of the data that passed through the wire. You must successfully run the VI at least once before you are able to retain the wire values.



Click the **Step Into** button, shown at left, to open a node and pause. When you click the **Step Into** button again, it executes the first action and pauses at the next action of the subVI or structure. You also can press <Ctrl> and down arrow keys. Single-stepping through a VI steps through the VI node by node. Each node blinks to denote when it is ready to execute. By stepping into the node, you are ready to single-step inside the node.



Click the **Step Over** button, shown at left, to execute a node and pause at the next node. You also can press <Ctrl> and right arrow keys. By stepping over the node, you execute the node without single-stepping through the node.



Click the **Step Out** button, shown at left, to finish executing the current node and pause. When the VI finishes executing, the **Step Out** button becomes dimmed. You also can press <Ctrl> and up arrow keys. By stepping out of a node, you complete single-stepping through the node and navigate to the next node.



The **Warning** button, shown at left, appears if a VI includes a warning and you placed a checkmark in the **Show Warnings** checkbox in the **Error List** window. A warning indicates there is a potential problem with the block diagram, but it does not stop the VI from running.

Exercise 2-1 Concept: Exploring a VI

Goal

Identify the parts of an existing VI.

Description

You received a VI from an employee that can convert seconds into a combination of hours/minutes/seconds. You must evaluate this VI to see if it works as expected and can display the remaining time until the plane arrives.

1. Open the `Exploring_a_VI.exe` simulation from the `C:\Exercises\LabVIEW Basics I\Exploring a VI` directory.
2. Follow the instructions given in the simulation.
3. Open `Seconds Breakdown.vi` in the `C:\Exercises\LabVIEW Basics I\Exploring a VI` directory. This is the LabVIEW VI shown in the simulation.
4. Test the VI using the values given in Table 2-1.
 - Enter the input value in the **Total Time in Seconds control**.
 - Click the **Run** button.
 - For each input, compare the given outputs to the outputs listed in Table 2-1. If the VI works correctly, they should match.



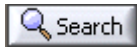
Table 2-1. Testing Values for Seconds Breakdown.vi

Input	Output
0 seconds	0 hours, 0 minutes, 0 seconds
60 seconds	0 hours, 1 minute, 0 seconds
3600 seconds	1 hour, 0 minutes, 0 seconds
3665 seconds	1 hour, 1 minutes, 5 seconds

End of Exercise 2-1

G. Searching for Controls, VIs, and Functions

When you access the **Controls** and **Functions** palettes from the **View** menu, there are two buttons at the top of the palette:



Search—Changes the palette to search mode so you can perform text-based searches to locate controls, VIs, or functions on the palettes. While a palette is in search mode, click the **Return** button to exit search mode and return to the palette.



View—Provides options for selecting a format for the current palette, showing and hiding categories for all palettes, and sorting items in the **Text** and **Tree** formats alphabetically. Select **Options** from the shortcut menu to display the **Controls/Functions Palettes** page of the **Options** dialog box, in which you can select a format for all palettes. This button appears only if you click the thumbtack in the upper left corner of a palette to pin the palette.

Until you are familiar with the location of VIs and functions, search for the function or VI using the **Search** button. For example, if you want to find the Random Number function, click the **Search** button on the **Functions** palette toolbar and start typing `Random Number` in the text box at the top of the palette. LabVIEW lists all matching items that either start with or contain the text you typed. You can click one of the search results and drag it to the block diagram, as shown in Figure 2-18.

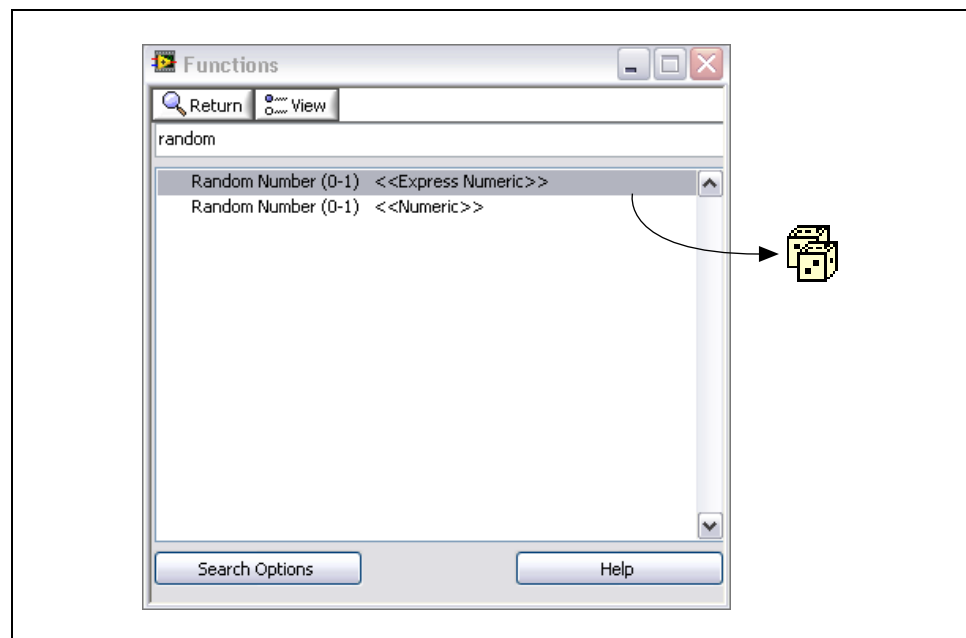


Figure 2-18. Searching for an object in the Functions palette

Double-click the search result to highlight its location on the palette. If the object is one you need to use frequently, you can add it to your Favorites

category. Right-click the object and select **Add Item to Favorites**, as shown in Figure 2-19.

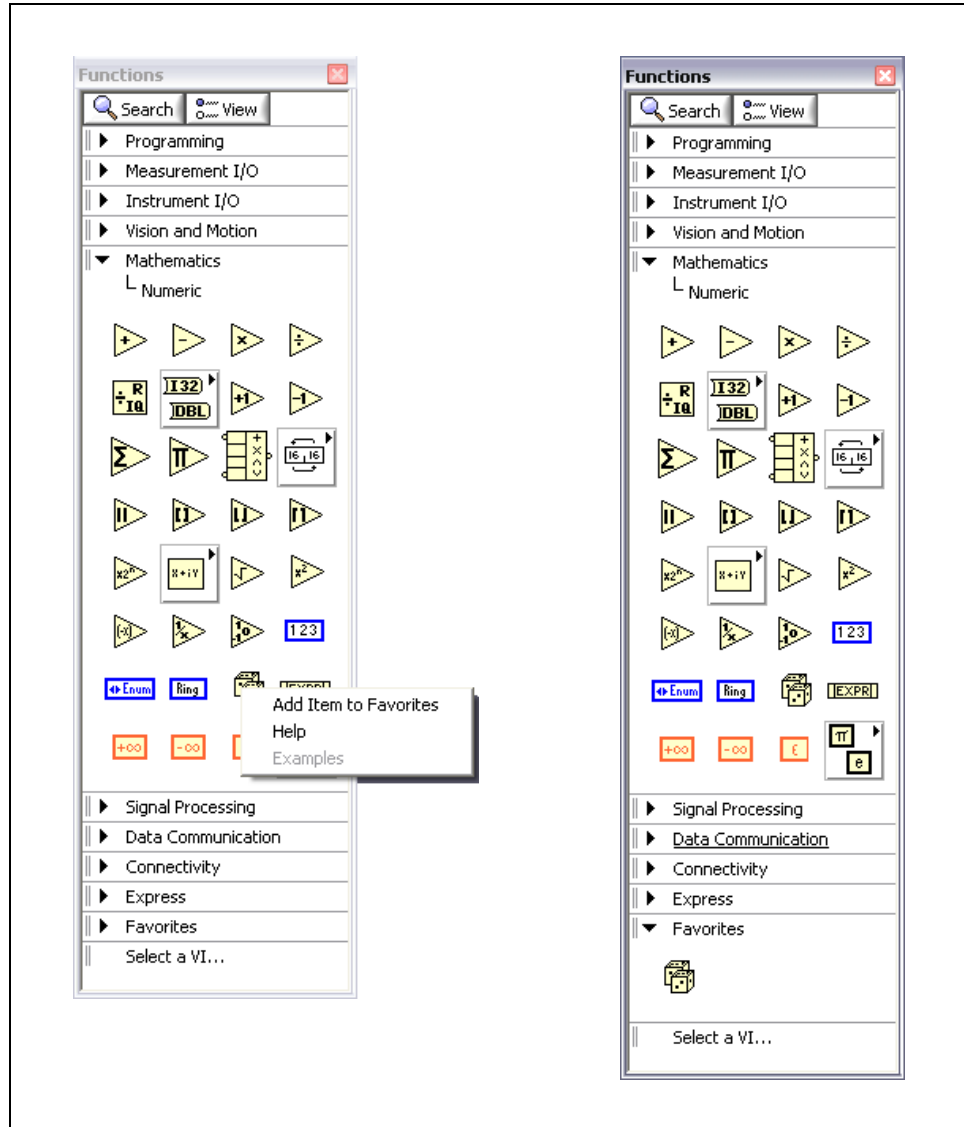


Figure 2-19. Adding an Item to the Favorites Category of a Palette

Exercise 2-2 Concept: Navigating Palettes

Goal

Learn to find controls and functions.

Description

1. Open `Navigating Palettes.exe` in the `C:\Exercises\LabVIEW Basics I\Navigating Palettes` directory.
2. Follow the instructions given. This simulation demonstrates how to find a control or function.
3. Using the instructions given in the simulation, place the DAQ Assistant Express VI in the **Favorites** category of the **Functions** palette.

End of Exercise 2-2

H. Selecting a Tool

You can create, modify and debug VIs using the tools provided by LabVIEW. A tool is a special operating mode of the mouse cursor. The operating mode of the cursor corresponds to the icon of the tool selected. LabVIEW chooses which tool to select based on the current location of the mouse.



Figure 2-20. Tools Palette



Tip You can manually choose the tool you need by selecting it on the **Tools** palette. Select **View»Tools Palette** to display the **Tools** palette.

Operating Tool



When the mouse cursor changes to the icon shown at left, it is using the Operating tool. The Operating tool changes the values of a control. For example, in Figure 2-21 the Operating tool moves the pointer on the Horizontal Pointer Slide. When the mouse hovers over the pointer, the cursor automatically accesses the Operating tool.

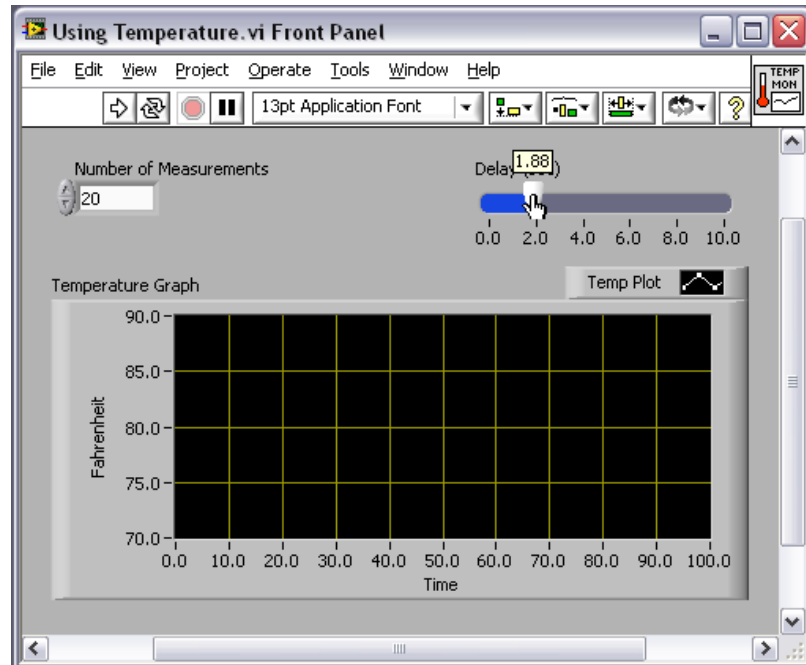


Figure 2-21. Using the Operating Tool

The Operating tool is mostly used on the front panel, but you also can use the Operating tool on the block diagram to operate increment/decrement buttons.

Positioning Tool



When the mouse cursor is an arrow, shown at left, the Positioning tool is functioning. The Positioning tool selects or resizes objects. For example, in Figure 2-22 the Positioning tool selects the `Number of Measurements` numeric control. After selecting an object, you can move, copy, or delete the object. When the mouse hovers over the edge of an object, the cursor automatically accesses the Positioning tool.

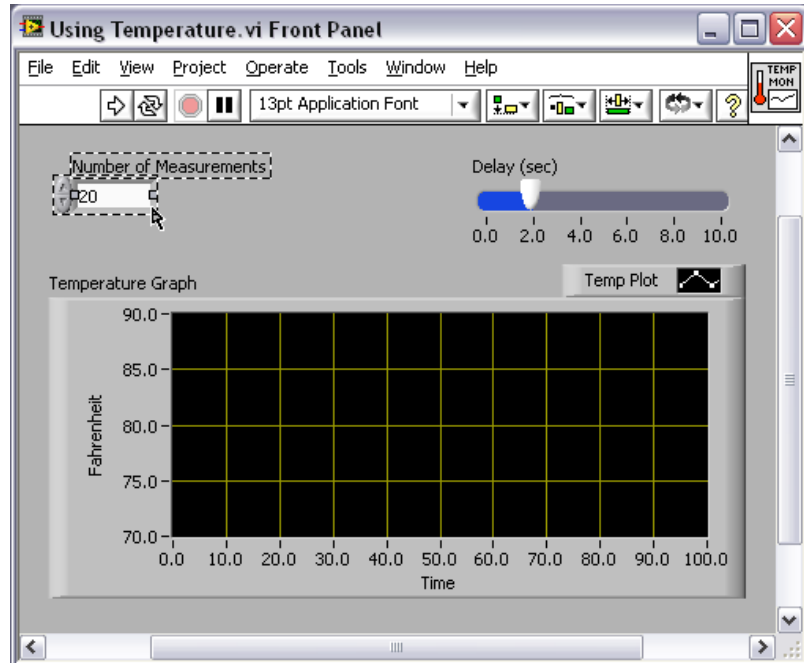


Figure 2-22. Using the Positioning Tool to select an object

If the mouse hovers over a resizing node of an object, the cursor mode changes to show that you can resize the object, as shown in Figure 2-23. Notice that the cursor is hovering over a corner of the XY Graph at a resizing node, and the cursor mode changes to a double-sided arrow.

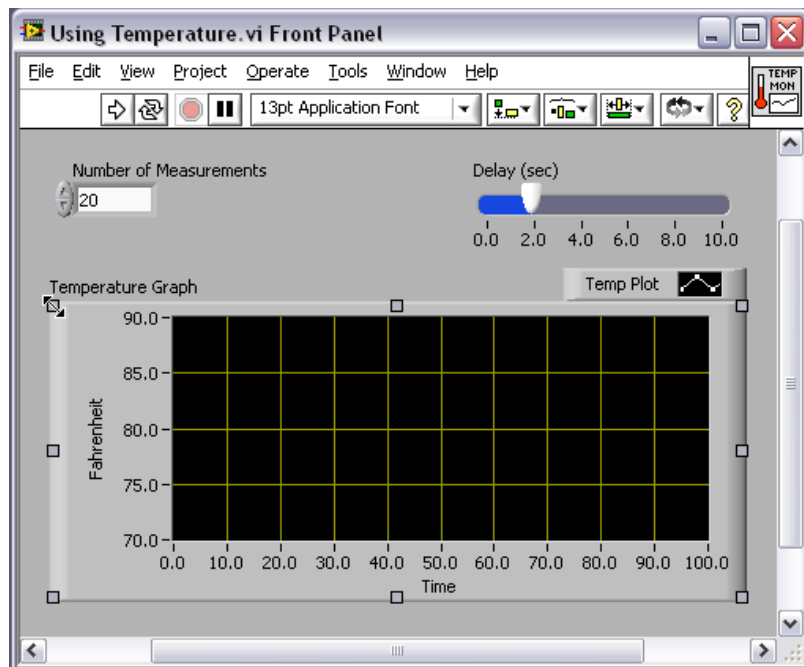


Figure 2-23. Using the Positioning tool to resize an object

You can use the Positioning tool on both the front panel and the block diagram.

Labeling Tool



When the mouse cursor changes to the icon shown at left, the Labeling tool is in operation. Use the Labeling tool to enter text in a control, to edit text, and to create free labels. For example, in Figure 2-24 the Labeling tool enters text in the **Number of Measurements** numeric control. When the mouse hovers over the interior of the control, the cursor automatically accesses the Labeling tool. Click once to place a cursor inside the control. Then, double-click to select the current text.

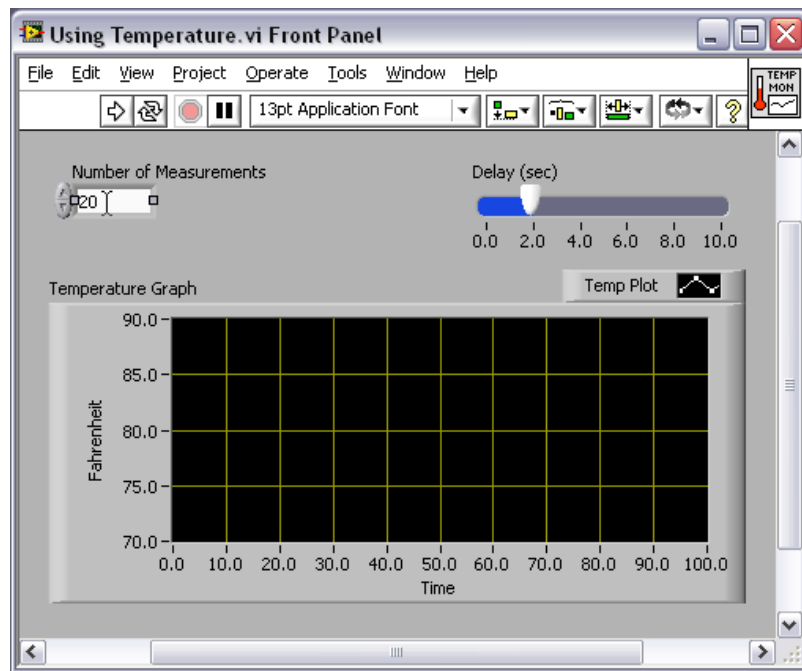


Figure 2-24. Using the Labeling Tool

When you are not in a specific area of a front panel or block diagram that accesses a certain mouse mode, the cursor appears as cross-hairs. When the cross-hairs mode is active, you can double-click to access the Labeling tool and create a free label.

Wiring Tool



When the mouse cursor changes to the icon shown at left, the Wiring tool is in operation. Use the Wiring tool to wire objects together on the block diagram. For example, in Figure the Wiring tool wires the **Number of Measurements** terminal to the count terminal of the For Loop. When the mouse hovers over the exit or entry point of a terminal or over a wire, the cursor automatically accesses the Wiring tool.

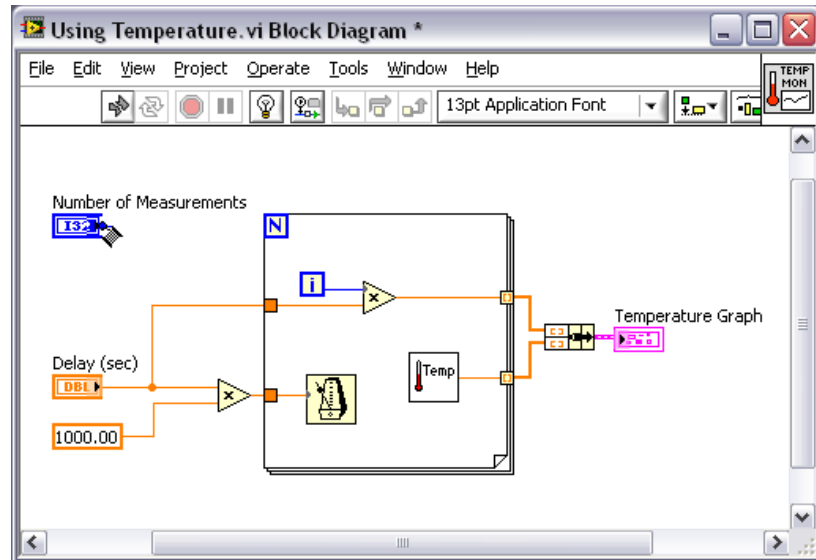


Figure 2-25. Using the Wiring Tool

The Wiring tool works mainly with the block diagram and when you create a connector pane on the front panel.

Other Tools Accessed from the Palette

You can access the Operating, Positioning, Labeling, and Wiring tools directly from the **Tools** palette, rather than using the Auto tool selection mode. Select **View»Tools Palette** to access the **Tools** palette.



Figure 2-26. The Tools Palette



The top item in the **Tools** palette, shown at left, is the Automatic Tool Selection. When this is selected, LabVIEW automatically chooses a tool based on the location of your cursor. You can turn off Auto tool by deselecting the item, or by selecting another item in the palette. There are some additional tools on the palette, as described below:



Use the Object Shortcut Menu tool, shown at left, to access an object shortcut menu with the left mouse button.



Use the Scrolling tool, shown at left, to scroll through windows without using scrollbars.



Use the Breakpoint tool, shown at left, to set breakpoints on VIs, functions, nodes, wires, and structures to pause execution at that location.



Use the Probe tool, shown at left, to create probes on wires on the block diagram. Use the Probe tool to check intermediate values in a VI that produces questionable or unexpected results.



Use the Color Copy tool, shown at left, to copy colors for pasting with the Coloring tool.



Use the Coloring tool, shown at left, to color an object. The Coloring tool also displays the current foreground and background color settings.

Exercise 2-3 Concept: Selecting a Tool

Goal

Use the Automatic Tool Selection to build experience for understanding its operation.

Description

During this exercise, you complete tasks in a partially built front panel and block diagram. These tasks give you experience in using the Auto tool.

1. Open the `Using Temperature.vi`.
 - Open LabVIEW.
 - Select **File»Open**.
 - Navigate to the `C:\Exercises\LabVIEW Basics I\Using Temperature` directory.
 - Select the `Using Temperature.vi`.

Figure 2-27 shows an example of the front panel as it appears after your modifications. You increase the size of the waveform graph, rename the numeric control, change the value of the numeric control, and move the pointer on the horizontal pointer slide.

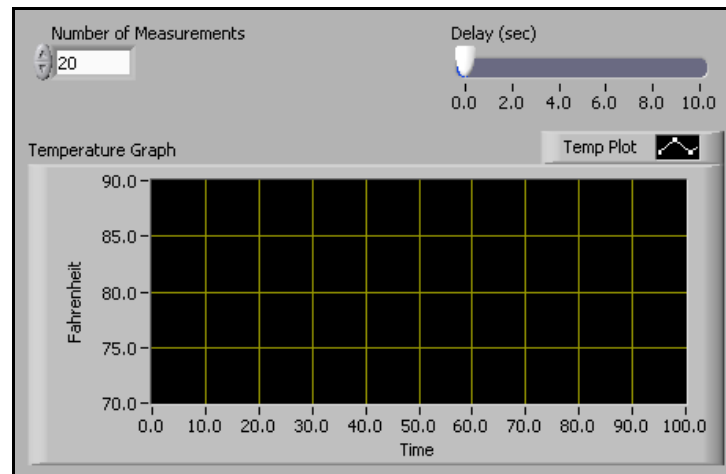
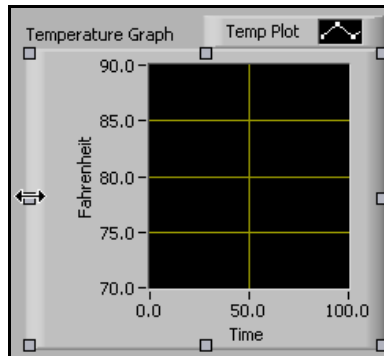


Figure 2-27. Using Temperature VI Front Panel

2. Expand the waveform graph horizontally using the Positioning tool.
 - Move the cursor to the left edge of the Waveform Graph.

- Move the cursor to the middle left resizing node until the cursor changes to a double arrow, as shown in Figure 2-28.



- Drag the repositioning point until the Waveform Graph is the size you want.
3. Rename the numeric control to `Number of Measurements` using the Labeling Tool
 - Move the cursor to the text `Numeric`.
 - Double click the word `Numeric`.
 - Enter the text `Number of Measurements`.
 - Complete the entry by pressing the `<Enter>` key on the numeric keypad, pressing the **Enter Text** button on the toolbar, or clicking the mouse outside of the control.
 4. Change the value of the `Number of Measurements` control to `20` using the Labeling tool.
 - Move the cursor to the interior of the numeric control.
 - When the cursor changes to the Labeling tool icon, as shown at left,

 press the mouse button.
 - Enter the text `20`.
 - Complete the entry by pressing the `<Enter>` key on the numeric keypad, pressing the **Enter Text** button on the toolbar, or clicking the mouse outside of the control.
 5. Change the value of the pointer on the horizontal pointer slide using the Operating tool.
 - Move the cursor to the pointer on the slide.



- When the cursor changes to the Operating tool icon, as shown at left, press the mouse button and drag to the value you want.
 - Leave the value at a value greater than 0.
6. Try changing the value of objects, resizing objects, and renaming objects until you are comfortable with using these tools.

Figure 2-28 shows an example of the block diagram as it appears after your modifications. You move the Number of Measurements terminal and wire the terminal to the count terminal of the For Loop.

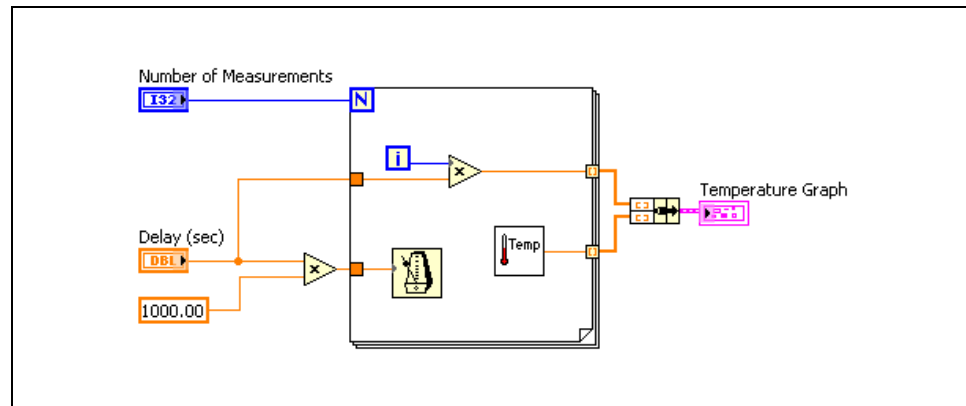


Figure 2-28. Using Temperature VI Block Diagram



7. Open the block diagram.
8. Move the Number of Measurements terminal using the Positioning tool.
- Move the cursor to the Number of Measurements terminal.
 - Move the cursor in the terminal until the cursor changes to an arrow, as shown at left.
 - Click and drag at the terminal to the new location as shown in Figure 2-28.



9. Wire the Number of Measurements terminal to the count terminal of the For Loop using the Wiring tool.
- Move the cursor to the Number of Measurements terminal.
 - Move the cursor to the right of the terminal, stopping when the cursor changes to a wiring spool, as shown at left.
 - Click to start the wire.



- Move the cursor to the count (**N**) terminal, shown at left, of the For Loop.
- Click to end the wire.

10. Try moving other objects, deleting wires and rewiring them, and wiring object and wires together until you are comfortable with using these tools.
11. Change the value of the Delay (sec) control to something greater than zero (0).



12. Click the **Run** button, shown at left, to run the VI.
The time required to execute this VI is equivalent to Number of Measurements times Delay (Sec). Once the VI is finished executing, the data is displayed on the Temperature Graph.
13. Select **File»Close** to close the VI. You do not need to save the VI.
14. Click the **Don't Save All** button to save changes before closing.

End of Exercise 2-3

I. Dataflow

LabVIEW follows a dataflow model for running VIs. A block diagram node executes when it receives all required inputs. When a node executes, it produces output data and passes the data to the next node in the dataflow path. The movement of data through the nodes determines the execution order of the VIs and functions on the block diagram.

Visual Basic, C++, JAVA, and most other text-based programming languages follow a control flow model of program execution. In control flow, the sequential order of program elements determines the execution order of a program.

For a dataflow programming example, consider a block diagram that adds two numbers and then subtracts 50.00 from the result of the addition, as shown in Figure 2-29. In this case, the block diagram executes from left to right, not because the objects are placed in that order, but because the Subtract function cannot execute until the Add function finishes executing and passes the data to the Subtract function. Remember that a node executes only when data are available at all of its input terminals and supplies data to the output terminals only when the node finishes execution.

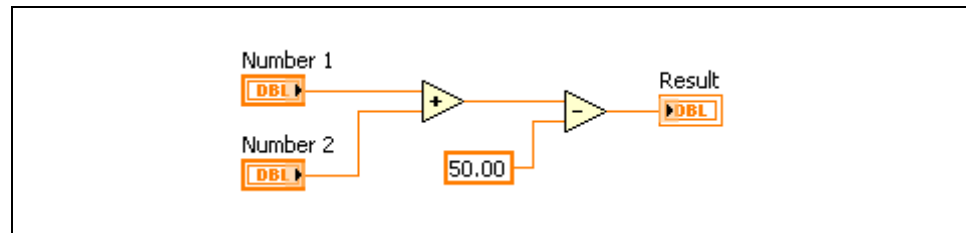


Figure 2-29. Dataflow Programming Example

In Figure 2-30, consider which code segment would execute first—the Add, Random Number, or Divide function. You cannot know because inputs to the Add and Divide functions are available at the same time, and the Random Number function has no inputs. In a situation where one code segment must execute before another, and no data dependency exists between the functions, use other programming methods, such as error clusters, to force the order of execution. Refer to Lesson 4, *Relating Data*, for more information about error clusters.

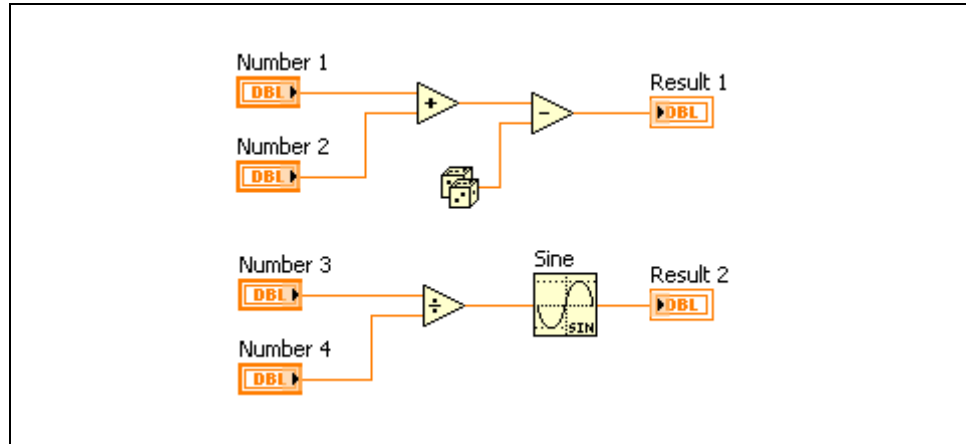


Figure 2-30. Dataflow Example for Multiple Code Segments

Exercise 2-4 Concept: Dataflow

Goal

Understand how dataflow determines the execution order in a VI.

Description

1. Open the `Dataflow.exe` simulation from the `C:\Exercises\LabVIEW Basics I\Dataflow` directory.
2. Follow the instructions given. This simulation demonstrates dataflow.

End of Exercise 2-4

J. Building a Simple VI

Most LabVIEW VIs have three main tasks: acquiring some sort of data, analyzing the acquired data, and presenting the result. When each of these parts are simple, you can complete the entire VI using very few objects on the block diagram. Express VIs are designed specifically for completing common, frequently used operations. In this section, you learn about some Express VIs in each of these categories: acquire, analyze, and present. Then you learn to build a simple VI using these three parts, as shown in Figure 2-31.

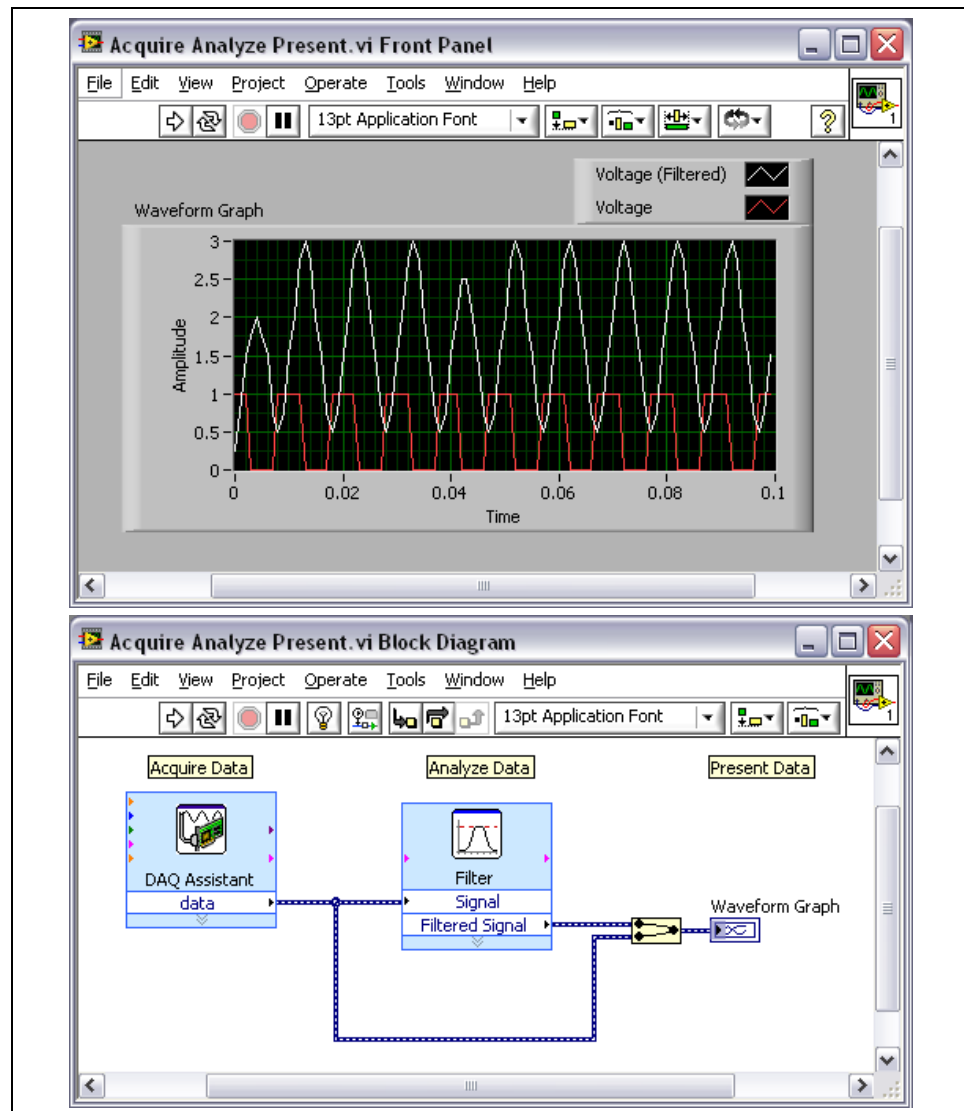


Figure 2-31. Acquire, Analyze and Present Example Front Panel and Block Diagram

On the **Functions** palette, the Express VIs are grouped together in the **Express** category. Express VIs use the dynamic data type to pass data between Express VIs.

Acquire

Express VIs used for the Acquire task include the following: DAQ Assistant, Instrument I/O Assistant, Simulate Signal, and Read from Measurement File.



DAQ Assistant

The DAQ Assistant acquires data through a data acquisition board. You must use this Express VI frequently throughout this course. Refer to Lesson 8, *Navigating LabVIEW*, for more information about the DAQ Assistant. Until you learn more about data acquisition, you only use one channel of the data acquisition board, CH0. This channel is connected to a temperature sensor on the DAQ Signal Accessory. You can touch the temperature sensor to change the temperature the sensor reads.



Instrument I/O Assistant

The Instrument I/O Assistant acquires instrument control data, usually from a GPIB or serial interface. Refer to Lesson 9, *Instrument Control* for more information about the Instrument I/O Assistant.



Simulate Signal

The Simulate Signal Express VI generates simulated data such as a sine wave.



Read From Measurement File

The Read From Measurement File Express VI reads a file that was created using the Write From Measurement File Express VI. It specifically reads LVM or TDM file formats. This Express VI does not read ASCII files. Refer to Lesson 10, *Analyzing and Storing Measurement Data* for more information on reading data from a file.

Analyze

Express VIs used for the Analyze task include the following: amplitude and level measurements, statistics, tone measurements, and so on.



Amplitude and Level Measurements

The Amplitude and Level Measurements Express VI performs voltage measurements on a signal. These include DC, rms, maximum peak, minimum peak, peak to peak, cycle average and cycle rms measurements.



Statistics

The Statistics Express VI calculates statistical data from a waveform. This includes mean, sum, standard deviation, and extreme values.



Spectral Measurements

The Spectral Measurements Express VI performs spectral measurement on a waveform, such as magnitude and power spectral density.



Tone Measurements

The Tone Measurements Express VI searches for a single tone with the highest frequency or highest amplitude. It also finds the frequency and amplitude of a single tone.



Filter

The Filter Express VI processes a signal through filters and windows. Filters used include the following: Highpass, Lowpass, Bandpass, Bandstop, and Smoothing. Windows used include Butterworth, Chebyshev, Inverse Chebyshev, Elliptical, and Bessel.

Present

Present tasks are Express VI that perform a function, such as the Write to Measurement File Express VI or are indicators that present data on the front panel of the VI. The most commonly used indicators for this task include the Waveform Chart, the Waveform Graph, and the XY Graph. Common Express VIs include the Write to Measurement File, the Build Text, DAQ Assistant, and the Instrument I/O Assistant. In this case, the DAQ Assistant and the Instrument I/O Assistant provide output data from the computer to the DAQ board or an external instrument.



Write to Measurement File

The Write to Measurement File Express VI writes a file in LVM or TDM file format. Refer to Lesson 10, *Analyzing and Storing Measurement Data*, for more information on writing to measurement files.



Build Text

The Build Text Express VI creates text, usually for displaying on the front panel or exporting to a file or instrument. Refer to Lesson 10, *Analyzing and Storing Measurement Data*, for more information on creating strings.

Running a VI



After you configure the Express VIs using the dialog boxes, and wire the Express VIs together, you can run the VI. When you finish building your VI, click the **Run** button on the toolbar, shown at left, to execute the VI.



While the VI is running, the **Run** button icon changes to the one shown at left. After the execution completes, the **Run** button icon changes back to its original state, and the front panel indicators contain data.

Run Button Errors

If a VI does not run, it is a broken, or nonexecutable, VI. The **Run** button appears broken, shown as follows, when the VI you are creating or editing contains errors.



If the button still appears broken when you finish wiring the block diagram, the VI is broken and cannot run.

Generally, this means that a required input is not wired, or a wire is broken. Press the broken run button to access the **Error List** dialog box. The **Error List** dialog box lists each error and describes the problem. You can double-click an error to go directly to the error. Refer to Lesson 5, *Debugging VIs*, for more information on debugging.

Exercise 2-5 Simple AAP VI

Goal

Create a simple VI that accomplishes the acquire, analyze, and present tasks.

Scenario

You need to acquire a sine wave for 2 seconds, determine and display the average value, log the data, and display the sine wave on a graph.

Design

The input for this problem is an analog channel of sine wave data. The outputs include a graph of the sine data and a file logging the data.

Flow Chart

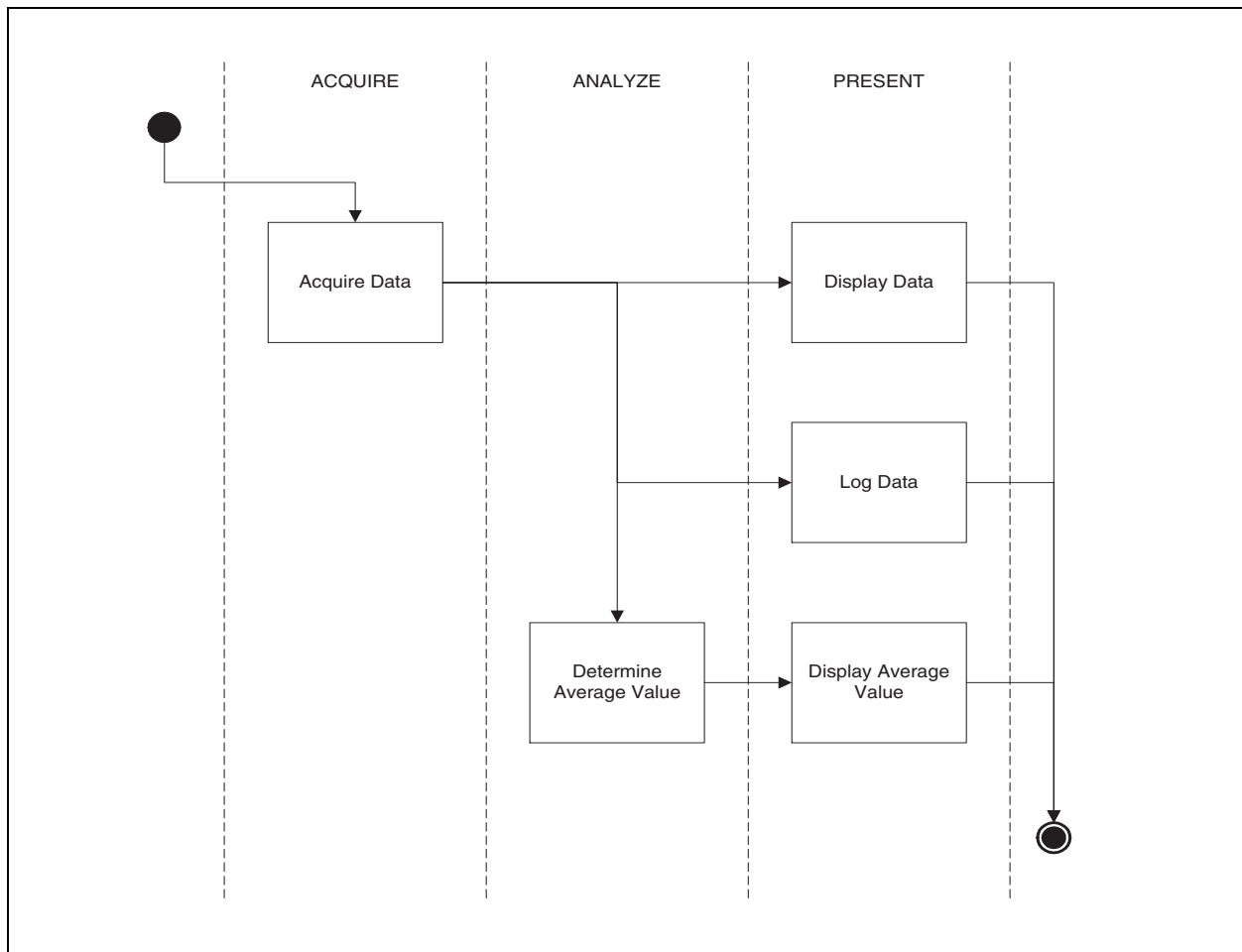









Figure 2-32. Simple AAP VI Flow Chart

Program Architecture—Quiz



1. Acquire: Circle the Express VI that is best suited to acquiring a sine wave from a data acquisition board.



	DAQ Assistant	The DAQ Assistant acquires data through a data acquisition board.
	Instrument I/O Assistant	The Instrument I/O Assistant acquires instrument control data, usually from a GPIB or serial interface.
	Simulate Signal	The Simulate Signal Express VI generates simulated data, such as a sine wave.

2. Analyze: Circle the Express VI that is best suited to determining the average value of the acquired data.

	Tone Measurements	The Tone Measurements Express VI finds the frequency and amplitude of a single tone.
	Statistics	The Statistics Express VI calculates statistical data from a waveform.
	Amplitude and Level Measurements	The Amplitude and Level Measurements Express VI performs voltage measurements on a signal.
	Filter	The Filter Express VI processes a signal through filters and windows.

3. Present: Circle the Express VIs and/or indicators that are best suited to displaying the data on a graph and logging the data to file.

	DAQ Assistant	The DAQ Assistant acquires data through a data acquisition board.
	Write to Measurement File	The Write to Measurement File Express VI writes a file in LVM or TDM file format.

 The icon for the Build Text Express VI, showing a blue square with a white border. Inside, the text "I.3 LEC" is displayed twice, once above and once below a red downward-pointing arrow.	Build Text	The Build Text Express VI creates text, usually for displaying on the front panel or exporting to a file or instrument.
 The icon for the Waveform Graph, showing a small grid with a black waveform. The y-axis is labeled with 0, 1, and 2, and the x-axis is labeled with 0.00 and 1.00.	Waveform Graph	The waveform graph displays one or more plots of evenly sampled measurements.

Refer to the next page for answers to this quiz.

Program Architecture—Solutions to Quiz

1. **Acquire:** Use the DAQ Assistant to acquire the sine wave from the data acquisition board.
2. **Analyze:** Use the Statistics Express VI to determine the average value of the sine wave. Because this signal is cyclical, you could also use the Cycle Average option in the Amplitude and Level Measurements to determine the average value of the sine wave.
3. **Present:** Use the Write to Measurement File Express VI to log the data and use the Waveform Graph to display the data on the front panel.

Implementation

1. Prepare your hardware to generate a sine wave.

Hardware

- Find the DAQ Signal Accessory and visually confirm that it is connected to the DAQ board in your computer.
- Using a wire, connect the Analog In Channel 1 to the Sine Function Generator, as shown in Figure 2-33.
- Set the **Frequency Range** switch and the **Frequency Adjust** knob to their lowest levels.
- Skip to step 2.

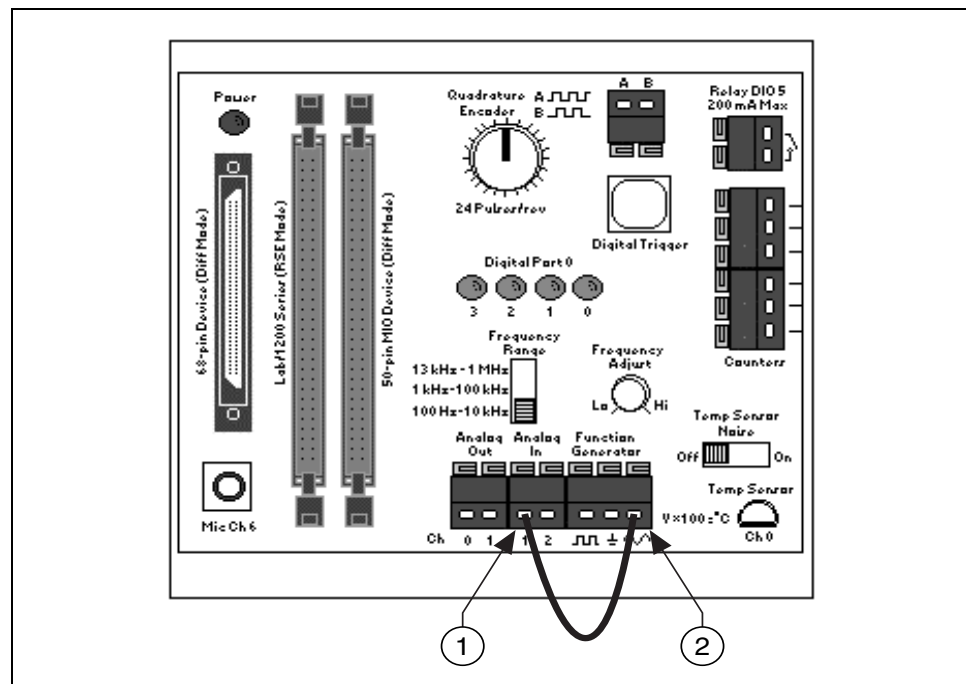


Figure 2-33. Connection for the DAQ Signal Accessory

No Hardware

- Skip this step if you are not using hardware.
2. Open LabVIEW.
 3. Open a blank VI.
 4. Save the VI as Simple AAP.vi.

- Select **File»Save**.
- Navigate to the C:\Exercises\LabVIEW Basics I\ Simple AAP directory.
- Name the VI Simple AAP.vi.
- Click **OK**.

Build a front panel similar to the one in Figure 2-34.

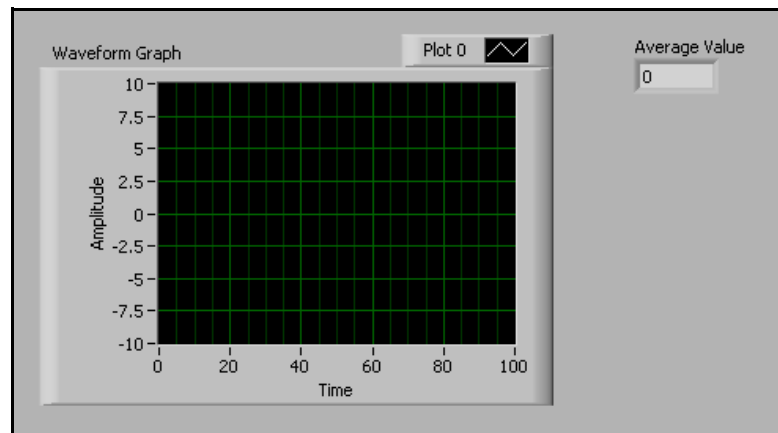


Figure 2-34. Acquire, Analyze and Present Front Panel

5. Place a waveform graph on the front panel to display the acquired data.
 - If the **Controls** palette is not already open, select **View»Controls Palette** from the LabVIEW menu.
 - On the **Controls** palette, select the **Express** category.
 - Select the **Graph Indicators** category from within the **Express** category.
 - Select the waveform graph.
 - Place the graph on the front panel.
6. Place a numeric indicator on the front panel to display the average value.
 - Collapse the **Graph Indicators** category by selecting **Express** on the **Controls** palette.
 - Select the **Numeric Indicators** category from within the **Express** category.
 - Select the numeric indicator.

- Place the indicator on the front panel.
- Enter `Average Value` in the label of the numeric indicator.

Build a block diagram similar to the one in Figure 2-35.

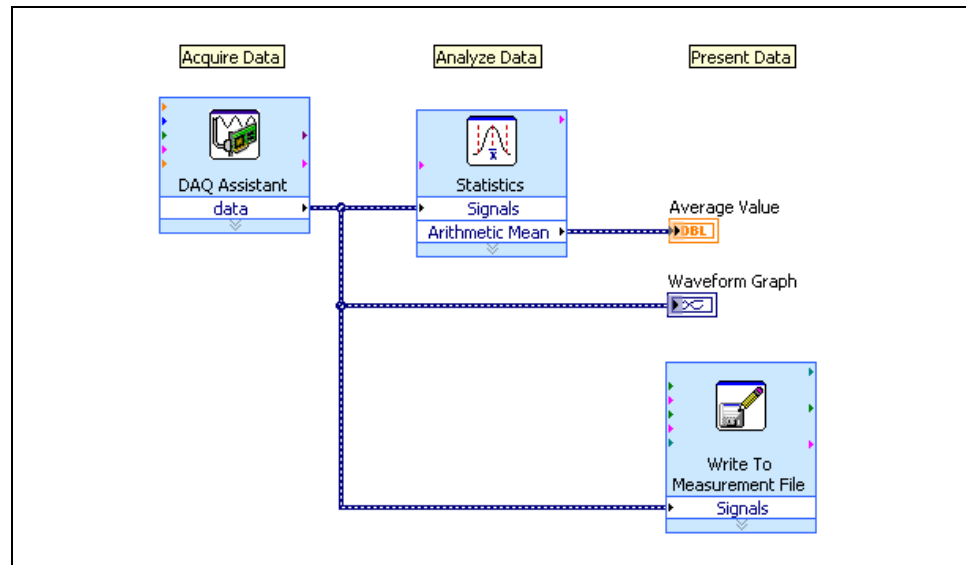


Figure 2-35. Acquire, Analyze, and Present Block Diagram

7. Open the block diagram of the VI.
 - Select **Window»Show Block Diagram**.



Note The terminals corresponding to the new front panel objects appear on the block diagram.

8. Acquire a sine wave for 2 seconds.

Hardware

The sine wave is generated on Analog In Channel 1. Use the DAQ Assistant Express VI to acquire 2 seconds of data.

- On the **Functions** palette, select the **Express** category.
- Select **Input** from the **Express** category.
- Select the **DAQ Assistant** from the **Input** category.
- Place the DAQ Assistant on the block diagram.
- Wait for the DAQ Assistant dialog box to open.

- Select **Analog Input»Voltage** for the measurement type.
- Select **ai1** (analog input channel 1) for the physical channel.
- Click the **Finish** button.
- On the Task Timing tab select **N Samples** as the **Acquisition Mode**.
- In the Clock Settings section enter 2,000 in **Samples To Read**.
- Enter 1,000 in **Rate (Hz)**.



Tip Reading 2,000 samples at a rate of 1,000 Hertz retrieves 2 seconds worth of data.

- Click the **OK** button.
- Skip to step 9.

No Hardware

Simulate the acquisition of a 100 Hz sine wave sampled at 1kHz for 2 seconds. Use the Simulate Signal Express VI.

- On the **Functions** palette, select the **Express** category.
- Select **Input** from the **Express** category.
- Select **Simulate Signal** from the **Input** category.
- Place the Simulate Signal Express VI on the block diagram.
- Wait for the Simulate Signal dialog box to open.
- Select **Sine** for the signal type.
- Set the signal frequency to 100.
- In the **Timing** section, set the **Samples per second (Hz)** to 1,000.
- In the **Timing** section, deselect **Automatic** for the Number of samples.
- In the **Timing** section, set the **Number of samples** to 2,000.
- Select the **Simulate acquisition timing** selection.
- Click the **OK** button.

9. Determine the average value of the data acquired by using the Statistics Express VI.
 - Collapse the **Express** subpalette by selecting **Express** on the **Functions** palette.
 - Select the **Signal Analysis** subpalette.
 - Select the **Statistics** Express VI and drag-and-drop the Statistics Express VI to the block diagram to the right of the DAQ Assistant.
 - Wait for the Statistics Express VI dialog box to open.
 - Place a checkmark in the **Arithmetic mean** checkbox.
 - Click **OK**.

10. Log the generated sine data to a LabVIEW Measurement File.
 - Select **Express** on the **Functions** palette.
 - Select the **Output** category.
 - Select **Write to Measurement File**.
 - Place the Write to Measurement File Express VI on the block diagram below the Statistics Express VI.
 - Wait for the Write to Measurement File Express VI dialog box to open.
 - Leave all settings as default.
 - Click **OK**.



Note Future exercises do not detail the directions for finding specific functions or controls in the palettes. Use the palette search feature to locate functions and controls.

11. Wire the data from the DAQ Assistant (or Simulate Signal Express VI) to the Statistics Express VI.
 - Place the mouse cursor over the **data** output of the DAQ Assistant (or Simulate Signal Express VI) at the location where the cursor changes to the Wiring tool.
 - Click the mouse button to start the wire.

- Place the mouse cursor over the **Signals** input of the Statistics Express VI and click the mouse button to end the wire.
12. Wire the data to the graph indicator.
- Place the mouse cursor over the **data** output wire of the DAQ Assistant (or Simulate Signal Express VI) at the location where the cursor changes to the Wiring tool.
 - Click the mouse button to start the wire.
 - Place the mouse cursor over the **graph** indicator and click the mouse button to end the wire.
13. Wire the **Arithmetic Mean** output of the Statistics Express VI to the **Average Value** numeric indicator.
- Place the mouse cursor over the **Arithmetic Mean** output of the Statistics Express VI at the location where the cursor changes to the Wiring tool.
 - Click the mouse button to start the wire.
 - Place the mouse cursor over the **Average Value** numeric indicator and click the mouse button to end the wire.
14. Wire the **data** output to the **Signals** input of the Write Measurement File Express VI.
- Place the mouse cursor over the **data** output wire of the DAQ Assistant (or Simulate Signal Express VI) at the location where the cursor changes to the Wiring tool.
 - Click the mouse button to start the wire.
 - Place the mouse cursor over the **Signals** input of the Write Measurement File Express VI and click the mouse button to end the wire.



Note Future exercises do not detail the directions for wiring between objects.

15. Save the VI.

Testing

1. Switch to the front panel of the VI.
2. Set the graph properties to be able to view the sine wave.
 - Right-click the waveform graph and select **X Scale»Autoscale X** to deselect autoscaling.
 - Right-click the waveform graph and select **Visible Items»X Scrollbar**.
 - Use the labeling tool to change the last number on the X Scale of the waveform graph to .1.
3. Save the VI.
4. Run the VI.
5. Open the front panel of the VI by selecting **Window»Show Front Panel**.
 - Click the **Run** icon on the front panel toolbar.

The graph indicator should display a sine wave and the **Average Value** indicator should display a number around zero. If the VI does not run as expected, review the implementation steps.

6. Close the VI.

End of Exercise 2-5

K. LabVIEW Help Utilities

Use the **Context Help** window, the *LabVIEW Help*, and the NI Example Finder to help you create and edit VIs. Refer to the *LabVIEW Help* and manuals for more information about LabVIEW.

Context Help Window



The **Context Help** window displays basic information about LabVIEW objects when you move the cursor over each object. To toggle display of the **Context Help** window, select **Help»Show Context Help**, press the <Ctrl-H> keys, or click the **Show Context Help Window** button, shown at left, on the toolbar.

When you move the cursor over front panel and block diagram objects, the **Context Help** window displays the icon for subVIs, functions, constants, controls, and indicators, with wires attached to each terminal. When you move the cursor over dialog box options, the **Context Help** window displays descriptions of those options. In the **Context Help** window, the labels of required terminals appear bold, recommended terminals appear as plain text, and optional terminals appear dimmed. The labels of optional terminals do not appear if you click the **Hide Optional Terminals and Full Path** button, shown as follows, in the **Context Help** window.

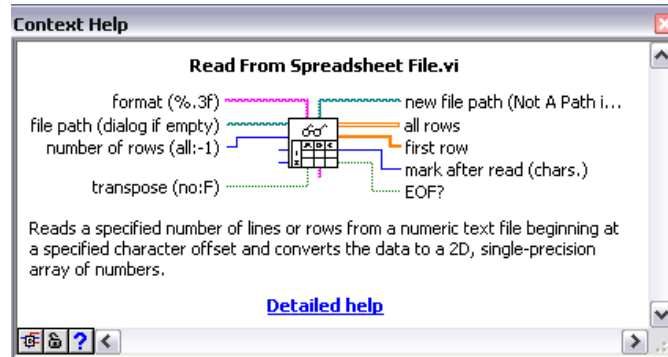


Figure 2-36. Context Help Window



Click the **Hide Optional Terminals and Full Path** button located on the lower left corner of the **Context Help** window to display the optional terminals of a connector pane and to display the full path to a VI. Optional terminals are shown by wire stubs, informing you that other connections exist. The detailed mode displays all terminals, as shown in Figure 2-37.

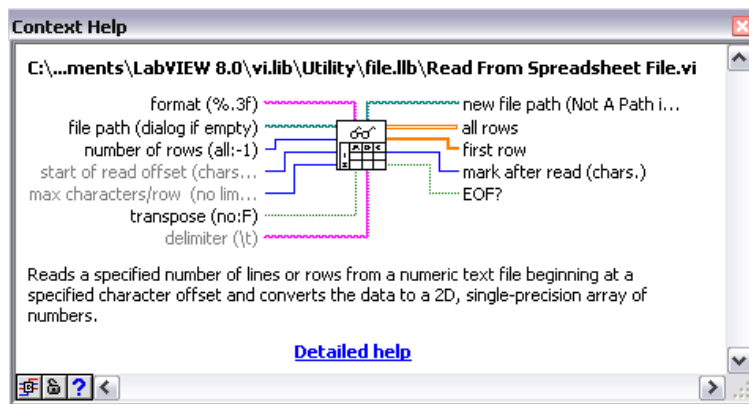


Figure 2-37. Detailed Context Help Window



Click the **Lock Context Help** button to lock the current contents of the **Context Help** window. When the contents are locked, moving the cursor over another object does not change the contents of the window. To unlock the window, click the button again. You also can access this option from the **Help** menu.



If a corresponding *LabVIEW Help* topic exists for an object the **Context Help** window describes, a blue **Click here for more help** link appears in the **Context Help** window. Also, the **More Help** button, shown at left, is enabled. Click the link or the button to display the *LabVIEW Help* for more information about the object.

LabVIEW Help

You can access the *LabVIEW Help* either by clicking the **More Help** button in the **Context Help** window, selecting **Help»Search the LabVIEW Help**, or clicking the blue **Click here for more help** link in the **Context Help** window. You also can right-click an object and select **Help** from the shortcut menu.

You can access the *LabVIEW Help* either by clicking the **More Help** button in the **Context Help** window, selecting **Help»Search the LabVIEW Help**, or clicking the blue

Click here for more help.

link in the **Context Help** window. You also can right-click an object and select **Help**.

The *LabVIEW Help* contains detailed descriptions of most palettes, menus, tools, VIs, and functions. The *LabVIEW Help* also includes step-by-step instructions for using LabVIEW features. The *LabVIEW Help* includes links to the following resources:

1. *Related Documentation*, which includes PDF versions of all LabVIEW manuals.
2. Technical support resources on the National Instruments Web site, such as the NI Developer Zone, the KnowledgeBase, and the Product Manuals Library.

NI Example Finder

The **New** dialog box contains many LabVIEW template VIs that you can use to start building VIs. However, these template VIs are only a subset of the hundreds of example VIs included with LabVIEW. You can modify any example VI to fit an application, or you can copy and paste from an example into a VI that you create.

In addition to the example VIs that ship with LabVIEW, you also can access hundreds of example VIs on the NI Developer Zone at ni.com/zone. To search all examples using LabVIEW VIs, use the NI Example Finder. The NI Example Finder is the gateway to all installed examples and the examples located on the NI Developer Zone.

To launch the NI Example Finder, select **Help»Find Examples** from the front panel or block diagram menu bar. You also can launch the NI Example Finder by selecting **Find Examples** on the **Getting Started** dialog box.

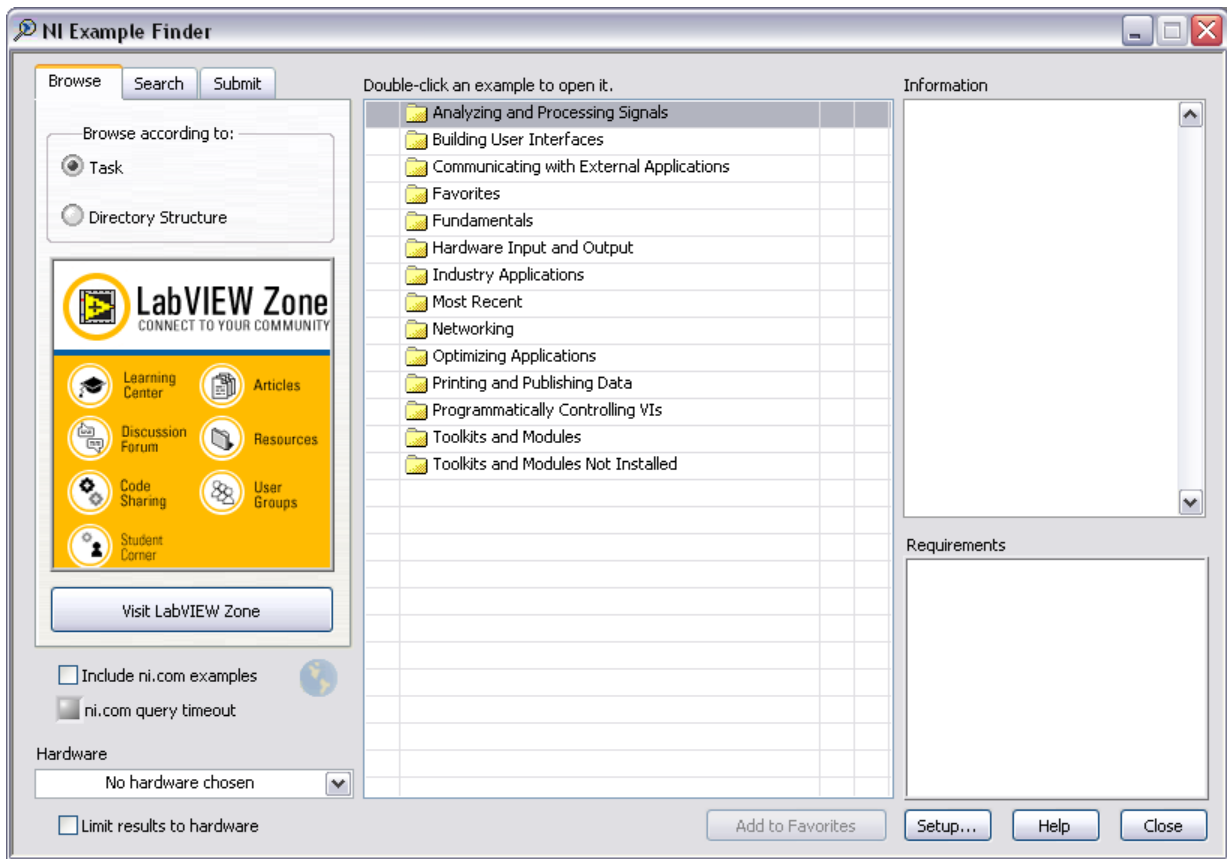


Figure 2-38. NI Example Finder

Exercise 2-6 Concept: Using Help

Goal

Become familiar with using the **Context Help** window, the *LabVIEW Help*, and the NI Example Finder.

Description

This exercise consists of a series of tasks designed to help you become familiar with the *LabVIEW Help* tools. Complete the follow steps.

NI Example Finder

1. You have a GPIB board in your computer, and you want to learn how to communicate with it using LabVIEW. Use the NI Example Finder to find a VI that communicates with a GPIB board.
 - Open LabVIEW.
 - Select **Help»Find Examples** to open the NI Example Finder.
 - Confirm that the **Task** option is selected on the **Browse** tab.
 - Double-click the **Hardware Input and Output** task to find examples related to hardware input and output.
 - Double-click the **GPIB** task.
 - Select the VI shown in this directory.
 - Notice that a description of the VI is provided in the **Information** window so that you can verify that this VI meets your needs.
 - Double-click the VI name to open the VI.
 - Close the VI after you finish exploring it.
2. You want to learn more about Express VIs, especially their use in filtering signals. Use the NI Example Finder to find an appropriate VI.
 - The NI Example Finder should still be open from the previous step. If not, open the NI Example Finder.
 - Click the **Search** tab in the NI Example Finder.
 - Enter *express* in the **Enter keyword(s)** field to find VIs that contain Express VIs.

- Double-click the `Express` result that appears in the **Double-click keyword(s)** field.
- This keyword is associated with many example VIs, as demonstrated by the number of VIs returned. You can select any one of these VIs and read the description in the **Information** window.
- Double-click the **Express Filter.vi** to open it.

Context Help Window

3. Use the **Context Help** window to learn about the Express VIs used in the Express Filter VI.
 - Open the block diagram by selecting **Window»Show Block Diagram**.
 - Open the **Context Help** window by selecting **Help»Show Context Help**.
 - Move the **Context Help** window to a convenient area, where the window does not hide part of the block diagram.
 - Place your mouse cursor over the Simulate Signal Express VI. The **Context Help** window content changes to show information about the object that your mouse is over.
 - Move your mouse over another Express VI. Notice the **Context Help** window content changes in reference to the location of the mouse cursor.
 - Move your mouse over one of the Tone Measurements Express VIs.
 - Examine the configuration details in the **Context Help** window. This gives you the information about how the Express VI is configured.
 - Double-click the Express VI to open the configuration dialog box. Notice that the selections in the configuration dialog box match the information in the **Context Help** window.
 - Click the **OK** button to close the configuration dialog box.
4. Anchor the **Context Help** window so that you can move your mouse without the contents of the window changing. The **Context Help** window should show information about the Simulate Signal Express VI.
 - Move your mouse over the Simulate Signal Express VI.



- To anchor the context help window, select the **Lock** button in the lower left corner of the window, shown at left.



Tip If the contents of the window change before you locked the window, avoid passing your mouse over other objects on the way to the **Context Help** window. Move the window closer to the object of interest to view context help for that item.

- Move your mouse over another object. Notice the contents of the window do not change while the lock button is selected.
 - Deselect the **Lock** button to resume normal operation of the window.
5. Modify the **Description and Tip** associated with the Simulated frequency control to change the content shown in the **Context Help** window.
- Select **Window»Show Front Panel** to open the front panel of the VI.
 - Move your mouse over the Simulated frequency control.
 - Read the contents of the **Context Help** window.
 - Right-click the **Simulated frequency** control.
 - Select **Description and Tip** from the shortcut menu.
 - Replace the text in the "**Simulated frequency**" **Description** box with the text `This is the description of the control.`
 - Replace the text in the "**Simulated frequency**" **Tip** box with the text `This is the tip for the control.`
 - Click **OK**.
 - Move your mouse over the **Simulated frequency** control.
 - Notice that the contents of the **Context Help** window changed to match the text you typed in the **Description** field of the **Description and Tip** dialog box.
 - Run the VI.
 - Place your mouse cursor over the Simulated frequency control.
 - Notice that the tool tip that appears matches the text you typed in the **Tip** field of the **Description and Tip** dialog box.

- Click the **Stop** button.

LabVIEW Help

6. Use the *LabVIEW Help* to learn more information about the Filter Express VI.
 - Select **Window»Show Block Diagram** to open the block diagram of the VI.
 - Right-click the Filter Express VI and select **Help** from the shortcut menu. This opens the *LabVIEW Help* topic for the Filter Express VI.



Note To access the *LabVIEW Help* for this topic, you can also select the **Detailed Help** hyperlink in the **Context Help** window while the Filter Express VI is selected, or click the question mark icon in the **Context Help** window.

- Explore the topic. For example, what is the purpose of the **Phase Response** dialog box option?
 - Click the **Find on the Functions Palette** button at the top of the Help window.
 - Close the **LabVIEW Help** window.
7. Close the Express Filter VI when you finish. Do not save changes.

End of Exercise 2-6

Self-Review: Quiz

Refer to Figure 2-39 to answer the following quiz questions.

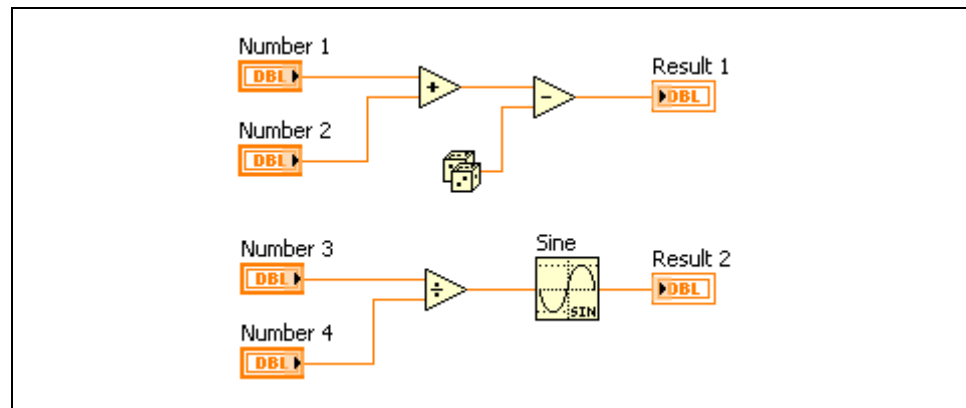


Figure 2-39. Dataflow Questions

1. Which of the following functions execute first?
 - a. Add
 - b. Subtract
 - c. Unknown
2. Which of the following functions execute first?
 - a. Sine
 - b. Divide
3. Which following functions execute first?
 - a. Random Number
 - b. Divide
 - c. Add
 - d. Unknown
4. Which following functions execute last?
 - a. Random Number
 - b. Subtract
 - c. Add
 - d. Unknown
5. What are the three parts of a VI?
 - a. Front Panel
 - b. Block Diagram
 - c. Project
 - d. Icon/Connector Pane

Self-Review: Quiz Answers

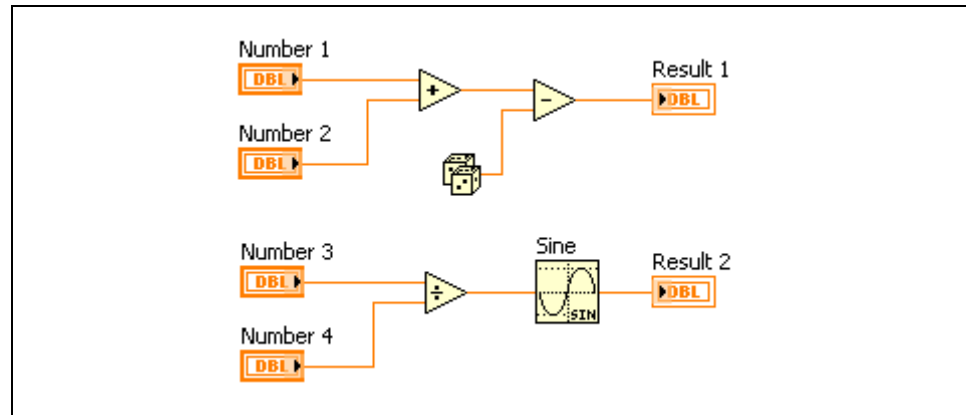


Figure 2-40. Dataflow Questions

- a. **Add**
 - b. Subtract
 - c. Unknown
6. Which of the following functions execute first?
 - a. Sine
 - b. **Divide**
 - c. Unknown
 7. Which following functions execute first?
 - a. Random Number
 - b. Divide
 - c. Add
 - d. **Unknown**
 8. Which following functions execute last?
 - a. Random Number
 - b. **Subtract**
 - c. Add
 - d. Unknown
 9. What are the three parts of a VI?
 - a. **Front Panel**
 - b. **Block Diagram**
 - c. Project
 - d. **Icon/Connector Pane**

Notes

Implementing a VI

This lesson teaches you how to implement code in LabVIEW. These skills include designing a user interface, choosing a data type, documenting your code, using looping structures such as While Loops and For Loops, adding software timing to your code, displaying your data as a plot, and making decisions in your code using a case structure.

Topics

- A. Front Panel Design
- B. Block Diagram Terminals
- C. Documenting Code
- D. While Loops
- E. For Loops
- F. Timing a VI
- G. Iterative Data Transfer
- H. Plotting Data
- I. Case Structures
- J. Formula Nodes

A. Front Panel Design

In the design phase of the software development method, you identify the inputs and outputs of the problem. This identification leads directly to the design of the front panel.

Retrieve the inputs of the problem from the following methods:

- acquiring from a device such as a data acquisition board or a multimeter.
- reading directly from a file.
- manipulating controls.

You can display the outputs of the problem with indicators or log the outputs to a file. You also can output data to a device using signal generation. Lessons about data acquisition, signal generation and file logging appear later in this course.

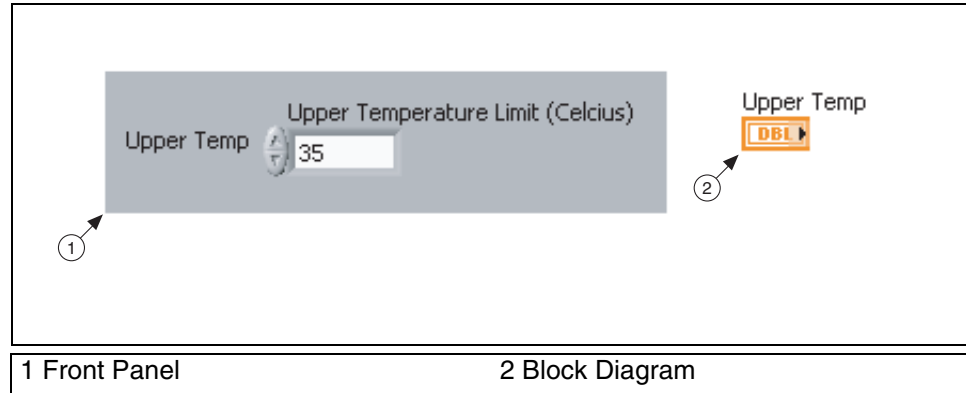
Design of Controls and Indicators

When choosing controls and indicators, make sure that they are appropriate for the task you want to perform. For example, when you want to determine the frequency of a sine wave, choose a dial control, or when you want to display temperature, choose a thermometer indicator.

Labels/Captions

When creating labels for controls and indicators, make sure to label them clearly. These labels help users identify the function for each control and indicator. Also, clear labelling helps you document your code in the block diagram. Control and indicator labels correspond to the names of terminals on the block diagram.

Captions help you describe a control on the front panel. Captions do not appear on the block diagram. Using captions allows you to document the user interface without cluttering the block diagram with long names. For example, in the Weather Station, you must provide an upper boundary for the temperature level. If the temperature rises above this level, the Weather Station indicates a heatstroke warning. You could call this control `Upper Temperature Limit (Celsius)`. However, this label would occupy unnecessary space on the block diagram. Instead use a caption for the control `Upper Temperature Limit (Celsius)` and use the label to create a shorter description for the block diagram, such as `Upper Temp`.



Control/Indicator Options

You can set default values for controls. Figure 3-1 shows a default value of 35 degrees Celsius. By setting a default value, you can assume a reasonable value for a VI if the user does not set another value during run-time. To set the default value complete the following steps:

1. Enter the desired value
2. Right-click the control and select **Data Operations»Make Current Value Default** from the shortcut menu.

You also can hide and view items on controls and indicators. For example, in Figure 3-1, you can see both the caption and the label. However, you only need to see the caption. To hide the label, right-click the control and select **Visible Items»Label** as shown in Figure 3-2.

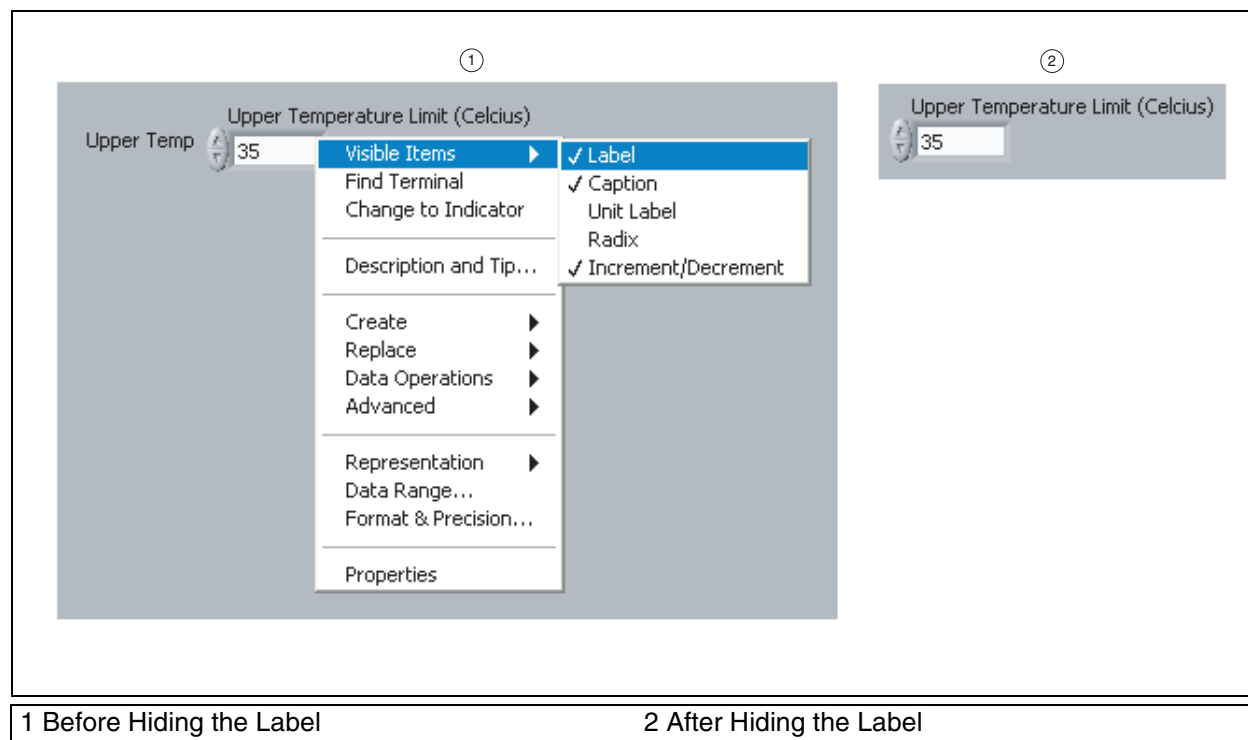


Figure 3-1. Hiding a Label on the Front Panel

Using Color

Proper use of color can improve the appearance and functionality of your front panel. Using too many colors, however, can result in color clashes that cause the front panels to look too busy and distracting.

LabVIEW provides a color picker that can aid in selecting appropriate colors. Select the Coloring tool and right-click an object or workspace to display the color picker. The top of the color picker contains a grayscale spectrum and a box you can use to create transparent objects. The second spectrum contains muted colors that are well suited to backgrounds and front panel objects. The third spectrum contains colors that are well suited for highlights. Moving your cursor vertically from the background colors to the highlight colors helps you select appropriate highlight colors for a specific background color.

The following tips are helpful for color matching:

- Use the default LabVIEW colors. If a color is not available on a computer, LabVIEW replaces it with the closest match. You also can use system colors to adapt the appearance of a front panel to the system colors of any computer that runs the VI.
- Start with a gray scheme. Select one or two shades of gray and choose highlight colors that contrast well against the background.

- Add highlight colors sparingly—on plots, abort buttons, and perhaps the slider thumbs—for important settings. Small objects need brighter colors and more contrast than larger objects.
- Use differences in contrast more often than differences in color. Color-blind users find it difficult to discern objects when differences are in color rather than contrast.
- Use spacing and alignment to group objects instead of grouping by matching colors.
- Good places to learn about color are stand-alone instrument panels, maps, and magazines.
- Choose objects from the **System Controls** category of the Controls palette if you want your front panel controls to use the system colors.

Spacing and Alignment

White space and alignment are probably the most important techniques for grouping and separation. The more items that your eye can find on a line, the cleaner and more cohesive the organization seems. When items are on a line, the eye follows the line from left to right or top to bottom. This is related to the script direction. Although some cultures view items right to left, almost all follow top to bottom.

When you design the front panel, consider how users interact with the VI and group controls and indicators logically. If several controls are related, add a decorative border around them or put them in a cluster.

Centered items are better than random but much less orderly than either left or right alignment. A band of white space acts as a very strong means of alignment. Centered items typically have ragged edges and the order is not as easily noticed.

Do not place front panel objects too closely together. Try to leave some blank space to make the front panel easier to read. Blank space also prevents users from accidentally clicking the wrong control or button.

Left-justify menus and right-justify related shortcuts as shown in Figure 3-3 on the left side: the LabVIEW **File** menu. Locating items in the center-justified menu as shown in the same example on the right is more difficult. Notice how the dividing lines between menu sections in the left example help you find the items quickly and strengthen the relationship between the items in the sections.

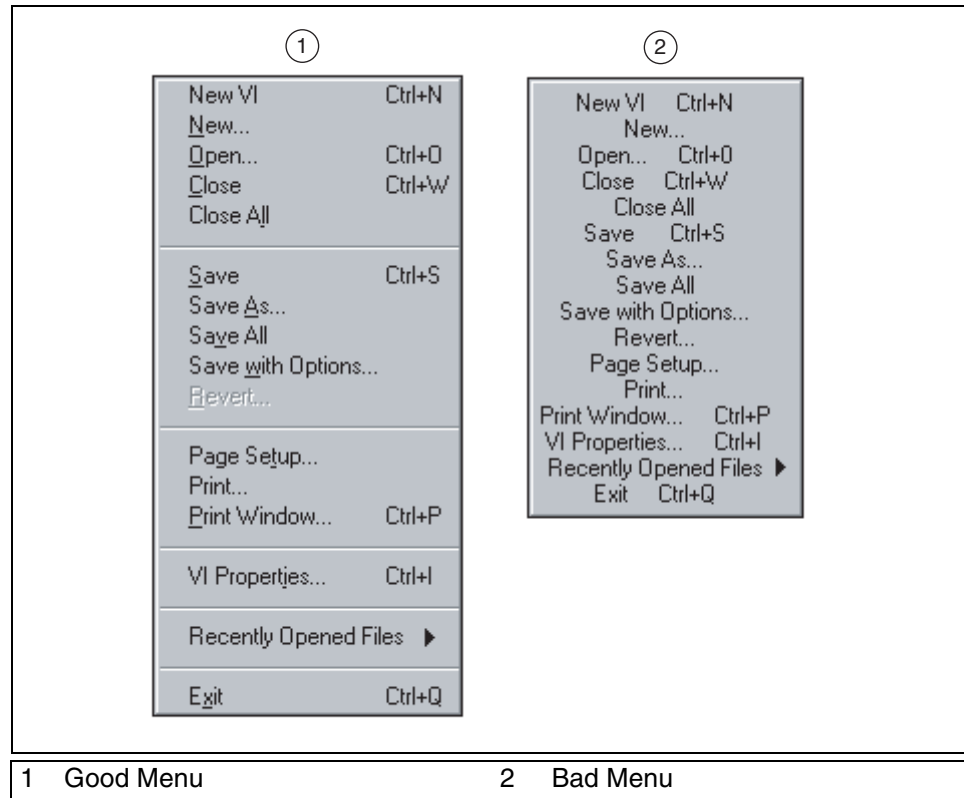


Figure 3-2. Good and Bad Menu Examples

Avoid placing objects on top of other objects. Placing a label or any other object over or partially covering a control or indicator slows down screen updates and can make the control or indicator flicker.

Text and Fonts

Text is easier to read and information is more easily understood when displayed in an orderly way. Use the default LabVIEW fonts. LabVIEW replaces the built-in fonts with comparable font families on different platforms. If you select a different font, LabVIEW substitutes the closest match if the font is unavailable on a computer.

Using too many font styles can make your front panel look busy and disorganized. Instead, use two or three different sizes of the same font. Serifs help people to recognize whole words from a distance. If you are using more than one size of a font, make sure the sizes are noticeably different. If not, it may look like a mistake. Similarly, if you use two different fonts, make sure they are distinct.

Design your front panels with larger fonts and more contrast for industrial operator stations. Glare from lighting or the need to read information from a distance can make normal fonts difficult to read. Also, remember that

touch screens generally require larger fonts and more spacing between selection items.



Note If fonts do not exist on a target machine, substituted fonts can cause the user interface to appear skewed.

User Interface Tips and Tools

Some of the built-in LabVIEW tools for making user-friendly front panels include system controls, tab controls, decorations, menus, and automatic resizing of front panel objects.

System Controls

A common user interface technique is to display dialog boxes at appropriate times to interact with the user. You can make a VI behave like a dialog box by selecting **File»VI Properties**, selecting the **Window Appearance** category, and selecting the **Dialog** option.

Use the system controls and indicators located on the **System** palette in dialog boxes you create. Because the system controls change appearance depending on which platform you run the VI, the appearance of controls in VIs you create is compatible on all LabVIEW platforms. When you run the VI on a different platform, the system controls adapt their color and appearance to match the standard dialog box controls for that platform.

System controls typically ignore all colors except transparent. If you are integrating a graph or non-system control into the front panel, try to match them match by hiding some borders or selecting colors similar to the system colors.

Tab Controls

Physical instruments usually have good user interfaces. Borrow heavily from their design principles, but use smaller or more efficient controls, such as ring controls or tab controls, where appropriate. Use tab controls to overlap front panel controls and indicators in a smaller area.

To add another page to a tab control, right-click a tab and select **Add Page Before** or **Add Page After** from the shortcut menu. Relabel the tabs with the Labeling tool, and place front panel objects on the appropriate pages. The terminals for these objects are available on the block diagram, as are terminals for any other front panel object (except Decorations).

You can wire the enumerated control terminal of the tab control to the selector of a Case structure to produce cleaner block diagrams. With this method you associate each page of the tab control with a subdiagram, or case, in the Case structure. You place the control and indicator terminals

from each page of the tab control—as well as the block diagram nodes and wires associated with those terminals—into the subdiagrams of the Case structure.

Decorations

Use the decorations located on the **Decorations** palette to group or separate objects on a front panel with boxes, lines, or arrows. These objects are for decoration only and do not display data.

Menus

Use custom menus to present front panel functionality in an orderly way and in a relatively small space. Using small amounts of space leaves room on the front panel for critical controls and indicators, items for beginners, items needed for productivity, and items that do not fit well into menus. You also can create keyboard shortcuts for menu items.

To create a run-time shortcut menu for front panel objects, right-click the front panel object and select **Advanced»Run-Time Shortcut Menu»Edit**. To create a custom run-time menu for your VI, select **Edit»Run-Time Menu**.

Automatic Resizing of Front Panel Objects

Use the **VI Properties»Window Size** options to set the minimum size of a window, maintain the window proportion during screen changes, and set front panel objects to resize in two different modes. When you design a VI, consider whether the front panel can display on computers with different screen resolutions. Select **File»VI Properties**, select **Window Size** in the **Category** pull-down menu, and place a checkmark in the **Maintain Proportions of Window for Different Monitor Resolutions** checkbox to maintain front panel window proportions relative to the screen resolution.

Most professional applications do not enlarge every control when the window changes size, but you can scale a table, graph, or list with the window, leaving other objects near the window edge. To scale one front panel object with the front panel, select that object and select **Edit»Scale Object with Panel**.

Exercise 3-1 Project: Weather Station UI VI

Goal

Create a front panel based on a given specification.

Scenario

You are given the design information for a weather station project that you must implement. Using the information, create a front panel that will be the user interface for the Weather Station VI.

Design

User Interface Inputs and Outputs

Type	Name	Properties
Numeric Control	Upper Limit	Double-Precision
Numeric Control	Lower Limit	Double-Precision
Waveform Chart	Temperature History	Digital Display visible
String Indicator	Warning	Three possible values: Heatstroke Warning, No Warning, and Freeze Warning
Boolean Control	Stop	

External Inputs and Outputs

- Current temperature—retrieved from a sensor attached to AI0 of the data acquisition board. Conversion formula: Voltage x 100 = degrees Celsius
- ASCII File—time, current temperature, temperature limits and warning string for each warning that occurred; tab-delimited.

State Transition Diagram

You developed a state transition diagram for the weather station in Exercise 1-2. However, to ensure consistency with the instructions given in this manual, a state transition diagram is shown in Figure 3-3 for you to use on the course project from this exercise forward.

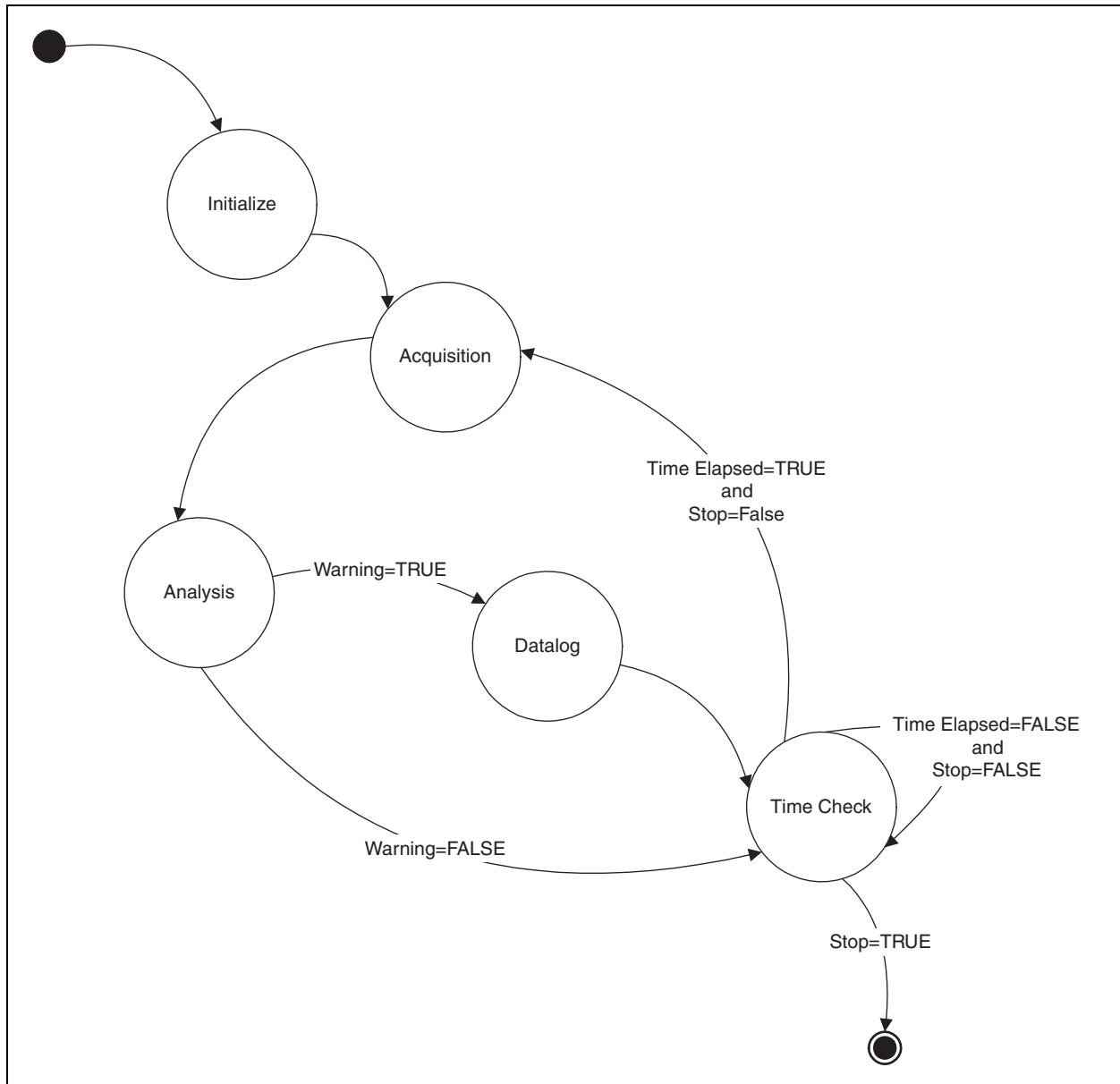


Figure 3-3. State Transition Diagram for the Course Project

Implementation

Each input entered and each output displayed must appear on the front panel. Complete the following instructions to create a front panel similar to that shown in Figure 3-4. This front panel retrieves the upper and lower temperature limits and a stop Boolean and displays the warning string and the temperature history.

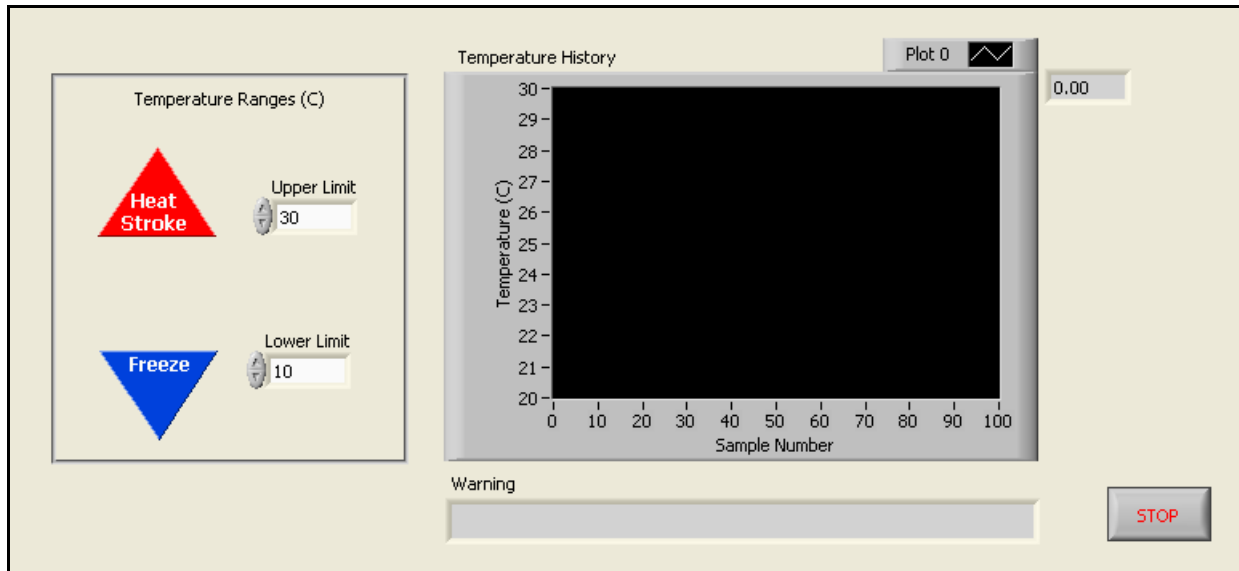


Figure 3-4. Front Panel for the Temperature Weather Station Project

1. Select **File»New Project** to create a new project in LabVIEW to contain the VIs associated with the temperature weather station.
2. Create a new VI for the user interface of the project.
 - Select **File»New VI**.
 - Save the VI as `Weather Station UI.vi` in the `C:\Exercises\LabVIEW_Basics_I\Course Project` directory.
3. Save the project.
 - Select **File»Save Project**.
 - Save the project as `Weather Station.lvproj` in the `C:\Exercises\LabVIEW_Basics_I\Course Project` directory.
4. Create a numeric control for the upper temperature limit.
 - Place a numeric control on the front panel.



- Change the label text of the numeric control to `Upper Limit`.
- Right-click the control, select **Representation**, and confirm that the representation type is set to double precision.

5. Create a numeric control for the lower temperature limit.



- Place a numeric control on the front panel.
- Change the label of the numeric control to `Lower Limit`.
- Right-click the control, select **Representation**, and confirm that the representation type is set to double precision.

6. Create a waveform chart for displaying the current temperature.



- Place a waveform chart on the front panel.
- Change the label text to `Temperature History`.
- Right-click the waveform chart and select **Visible Items»Digital Display**. This display shows the numeric value of the last item charted.



Note You modify the behavior and appearance of the waveform chart in another exercise.

7. Create a string indicator for displaying the current warning state.



- Place a string indicator on the front panel.
- Change the label text to `Warning`.
- Place the mouse cursor inside the indicator area.
- Use the font tools on the front panel toolbar to change the size, color and font of the indicator.



Tip You can enter text into the Warning indicator to test the fonts that you choose.

8. Create a Stop button.



- Place a Stop button on the front panel.
- Right-click the Stop button and select **Visible Items»Label**, to hide the label.

- Increase the size of the Stop button by selecting a corner of the Stop button with the selection tool and dragging the area to the desired size and shape.
9. Add labels and decorations and arrange the objects on the front panel to increase the usability. Refer to Figure 3-4 for suggestions, or design the front panel your own way. Tips:
- Temperature Ranges (C) is a free label. You can create a free label by double-clicking in an empty area of the front panel and typing the text.
 - The square around the temperature ranges and the triangles used to illustrate heatstroke and freeze is available in the Decorations category of the Controls palette.
 - Visually group all controls together on the left side of the front panel.
 - Visually group all indicators together on the right side of the panel. The Stop button is the only exception to this rule, as it is generally placed in the lower right side of the panel.
 - Visually group similar objects together.
 - Use colors where appropriate on the front panel. For example, the Stop button is usually red, heat is generally associated with red, and cold is generally associated with blue.
10. Save the VI.
11. Close the project.

You continue to build this project in later exercises.

End of Exercise 3-1

B. Block Diagram Terminals

When you design the front panel, the controls and indicators have corresponding terminals on the block diagram.

Controls, Indicators and Constants

Controls, indicators, and constants behave as inputs and outputs of the block diagram algorithm. Consider the implementation of the algorithm for the area of a triangle:

$$\text{Area} = .5 * \text{Base} * \text{Height}$$

In this algorithm, Base and Height are inputs and Area is an output, as shown in Figure 3-5.

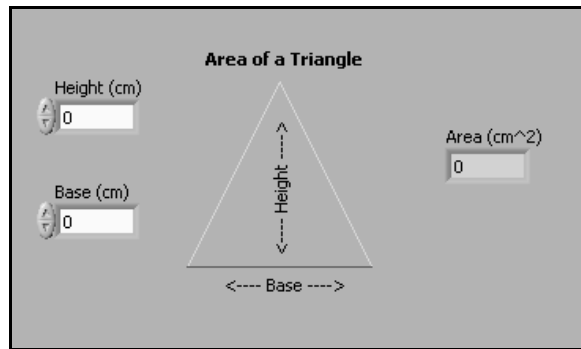


Figure 3-5. Area of a Triangle Front Panel

The constant .5 does not necessarily appear on the front panel, except possibly as documentation of the algorithm.

Figure 3-6 shows a possible implementation of this algorithm on a LabVIEW block diagram. This block diagram has five different terminals created by two controls, one constant, and one indicator.

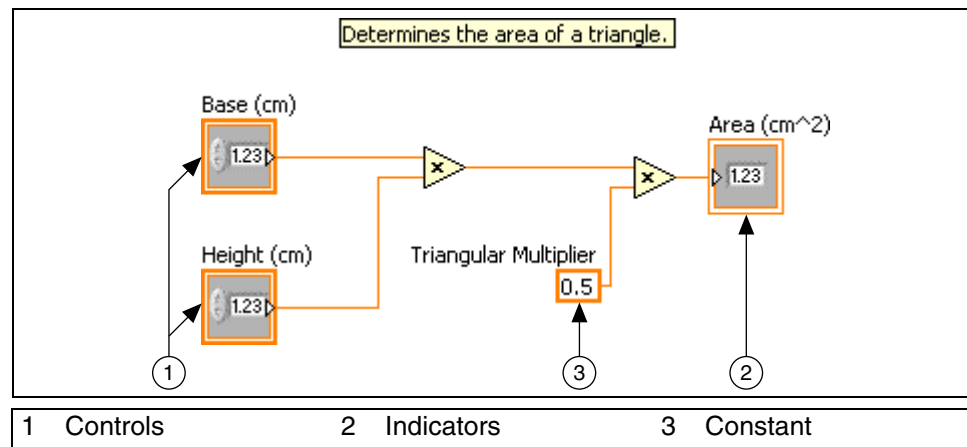


Figure 3-6. Area of a Triangle Block Diagram with Icon Terminal View

Notice that the **Base (cm)** and **Height (cm)** block diagram terminals have a different appearance from **Area (cm²)**. There are two distinguishing characteristics between a control and an indicator on the block diagram. The first is an arrow on the terminal that indicates the direction of data flow. The controls have arrows showing the data leaving the terminal, whereas the indicator has an arrow showing the data entering the terminal. The second distinguishing characteristic is the border around the terminal. Controls have a thick border, and indicators have a thin border.

Figure 3-7 shows the same block diagram without using the icon view of the terminals; however, the same distinguishing characteristics between controls and indicators exist.

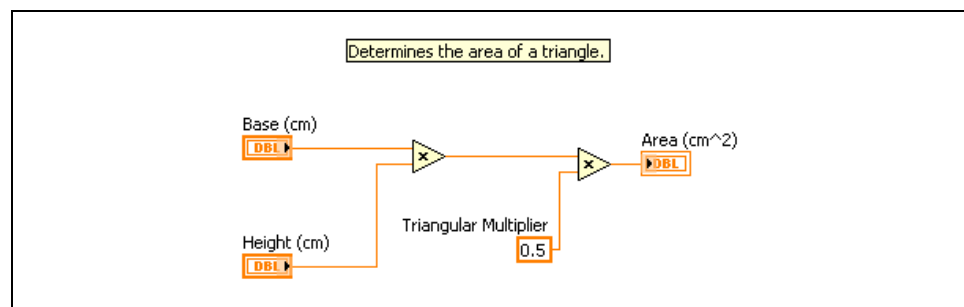


Figure 3-7. Area of a Triangle Block Diagram without Icon Terminal View

LabVIEW Data Types

Many different data types exist for data. You already learned about numeric, Boolean, and string data types in Lesson 2, *Navigating LabVIEW*. Other data types include the enumerated data type, dynamic data, and others. Even within numeric data types, there are different data types, such as whole numbers or fractional numbers.

The block diagram terminals visually communicate to the user some information about the data type they represent. For example, in Figure 3-7, **Height (cm)** is a double-precision, floating-point numeric. This is indicated by the color of the terminal, orange, and by the text shown on the terminal, DBL.

Numeric

The numeric data type represents numbers of various types. To change the representation type of a number, right-clicking the control, indicator, or constant, and select **Representation**, as shown in Figure 3-8.

When you wire two or more numeric inputs of different representations to a function, the function usually returns output in the larger or wider format. The functions coerce the smaller representations to the widest

representation before execution and LabVIEW places a coercion dot on the terminal where the conversion takes place.

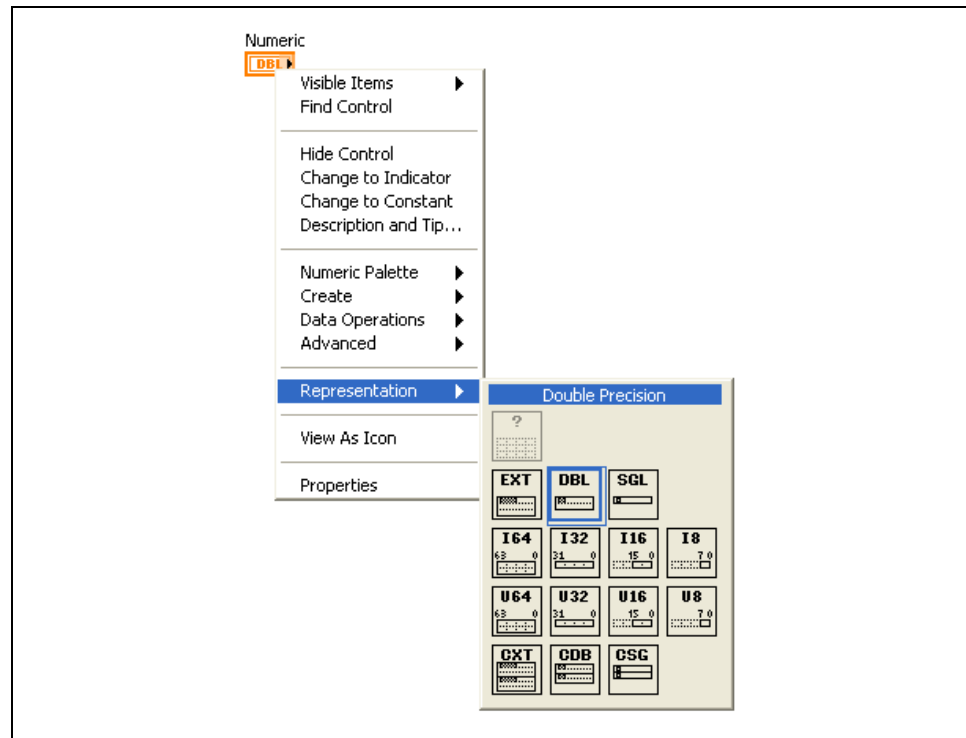


Figure 3-8. Numeric Representation

Within the numeric data type, there are the following subcategories of representation: floating-point numbers, signed integers, unsigned integers, and complex numbers.

Floating-Point Numbers

Floating-point numbers represent fractional numbers. In LabVIEW, floating-point numbers are represented with the color orange.

Single-precision (SGL)—Single-precision, floating-point numbers have 32-bit IEEE single-precision format. Use single-precision, floating-point numbers to save memory and avoid overflowing the range of the numbers.

Double-precision (DBL)—Double-precision, floating-point numbers have 64-bit IEEE double-precision format. Double-precision is the default format for numeric objects. For most situations, use double-precision, floating-point numbers.

Extended-precision (EXT)—In memory, the size and precision of extended-precision numbers vary depending on the platform. In Windows, they have 80-bit IEEE extended-precision format.

Integers

Integers represent whole numbers. Signed integers can be positive or negative. Use the unsigned integer datatypes when you know the integer is always positive. In LabVIEW, integers are represented with the color blue.

When LabVIEW converts floating-point numbers to integers, the VI rounds to the nearest even integer. For example, LabVIEW rounds 2.5 to 2 and rounds 3.5 to 4.

Byte (I8)—Byte integer numbers have 8 bits of storage.

Word (I16)—Word integer numbers have 16 bits of storage.

Long (I32)—Long integer numbers have 32 bits of storage. In most cases, it is best to use a 32-bit integer.

Quad (I64)—Quad integer numbers have 64 bits of storage.

Complex Numbers

Complex numbers are represented by two values linked together in memory: one representing the real part and one representing the imaginary part. In LabVIEW, since complex numbers are a type of floating-point number, complex numbers are also represented with the color orange.

Complex Single—Complex single-precision floating-point numbers consist of real and imaginary values in 32-bit IEEE single-precision format.

Complex Double—Complex double-precision floating-point numbers consist of real and imaginary values in 64-bit IEEE double-precision format.

Complex Extended—Complex extended-precision floating-point numbers consist of real and imaginary values in IEEE extended-precision format. In memory, the size and precision of extended-precision numbers vary depending on the platform. In Windows, they have 80-bit IEEE extended-precision format.

Boolean Values

LabVIEW stores Boolean data as 8-bit values. If the 8-bit value is zero, the Boolean value is FALSE. Any nonzero value represents TRUE. In LabVIEW, the color green represents Boolean data.

Boolean values also have a mechanical action associated with them. The two major actions are latch and switch. Latch action is similar to a doorbell, whereas switch action is similar to a light switch. You can also define when the switch or latch occurs: when pressed, when released or until released. To

learn more about mechanical action, experiment with the Mechanical Action of Booleans VI in the NI Example Finder.

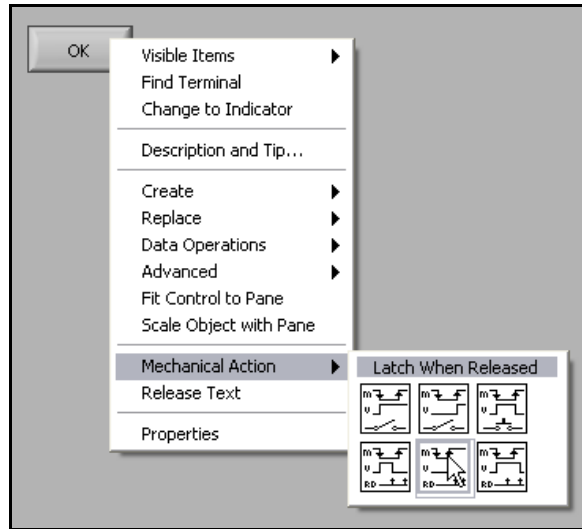


Figure 3-9. Boolean Mechanical Action

String

A string is a sequence of displayable or non-displayable ASCII characters. Strings provide a platform-independent format for information and data. Some of the more common applications of strings include the following:

- Creating simple text messages.
- Passing numeric data as character strings to instruments and then converting the strings to numeric values.
- Storing numeric data to disk. To store numeric data in an ASCII file, you must first convert numeric data to strings before writing the data to a disk file.
- Instructing or prompting the user with dialog boxes.

On the front panel, strings appear as labels, text entry boxes, and tables. LabVIEW includes built-in VIs and functions you can use to manipulate strings, including formatting strings, parsing strings, and other editing.

In LabVIEW, strings are represented with the color pink.

You can display strings in different ways. **Normal Display** is most commonly used. **'/' Codes Display** allows you to view non-displayable ASCII characters. **Password Display** allows you to display an asterisk (*), instead of the actual character typed. **Hex Display** shows the hex value for each character and is useful when debugging instrument communication.

Right-click a string control or indicator on the front panel to select from the display types shown in the following table. The table also shows an example message in each display type.

Display Type	Description	Message
Normal Display	Displays printable characters using the font of the control. Non-displayable characters generally appear as boxes.	There are four display types. \ is a backslash.
'\ ' Codes Display	Displays backslash codes for all non-displayable characters.	There\sare\sfour\sdisplay\stypes.\n\\\sis\sa\sbackslash.
Password Display	Displays an asterisk (*) for each character including spaces.	***** ***** ***
Hex Display	Displays the ASCII value of each character in hex instead of the character itself.	5468 6572 6520 6172 6520 666F 7572 2064 6973 706C 6179 2074 7970 6573 2E0A 5C20 6973 2061 2062 6163 6B73 6C61 7368 2E

LabVIEW stores strings as a pointer to a structure that contains a 4-byte length value followed by a 1D array of byte integers (8-bit characters).

Enumerated Control

An enum (enumerated control, constant or indicator) is a combination of data types. An enum represents a pair of values, a string and a numeric, where the enum can be one of a list of values. For example, if you created an enum type called Month, the possible value pairs for a Month variable are January-0, February-1, and so on through December-11. Figure 3-10 shows an example of these data pairs in the **Properties** dialog box for an enumerated control.

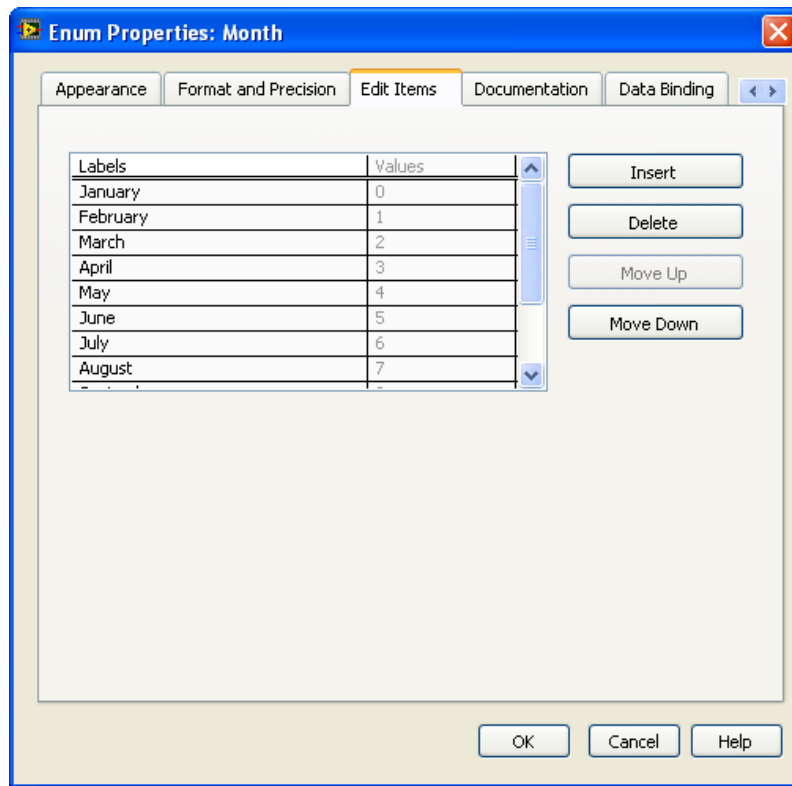


Figure 3-10. Properties for the Month Enumerated Control

Enums are useful because manipulating numbers on the block diagram is easier than strings. Figure 3-11 shows the Month enumerated control on the front panel, the selection of a data pair in the enumerated control, and the corresponding block diagram terminal.

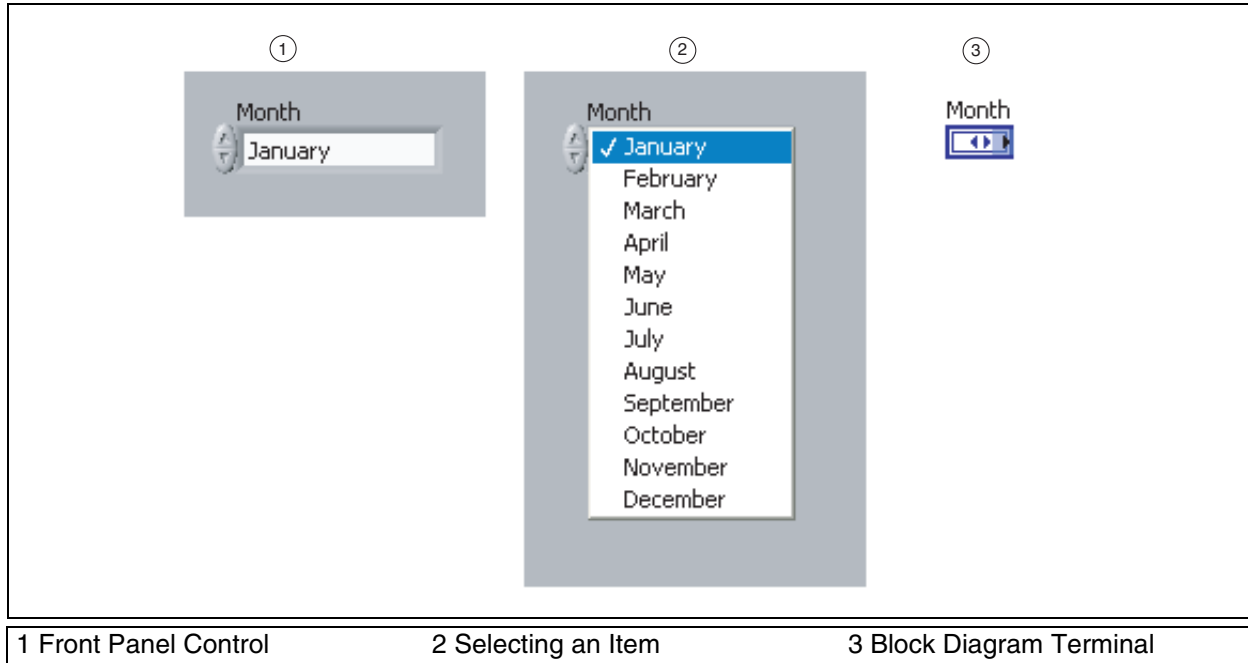


Figure 3-11. Month Enumerated Control

Dynamic



The dynamic data type stores the information generated or acquired by an Express VI. The dynamic data type appears as a dark blue terminal, shown at left. Most Express VIs accept and/or return the dynamic data type. You can wire the dynamic data type to any indicator or input that accepts numeric, waveform, or Boolean data. Wire the dynamic data type to an indicator that can best present the data. Indicators include graphs, charts, or numeric indicators.

Most other VIs and functions in LabVIEW do not accept the dynamic data type. To use a built-in VI or function to analyze or process the data the dynamic data type includes, you must convert the dynamic data type.



Use the Convert from Dynamic Data Express VI, shown at left, to convert the dynamic data type to numeric, Boolean, waveform, and array data types for use with other VIs and functions. When you place the Convert from Dynamic Data Express VI on the block diagram, the **Configure Convert from Dynamic Data** dialog box appears. The **Configure Convert from Dynamic Data** dialog box displays options that let you specify how you want to format the data that the Convert from Dynamic Data Express VI returns.

When you wire a dynamic data type to an array indicator, LabVIEW automatically places the Convert from Dynamic Data Express VI on the block diagram. Double-click the Convert from Dynamic Data Express VI to

open the **Configure Convert from Dynamic Data** dialog box to control how the data appears in the array.

Use the Convert to Dynamic Data Express VI to convert numeric, Boolean, waveform, and array data types to the dynamic data type for use with Express VIs. When you place the Convert to Dynamic Data Express VI on the block diagram, the **Configure Convert to Dynamic Data** dialog box appears. Use this dialog box to select the kind of data to convert to the dynamic data type.

C. Documenting Code

Professional developers who maintain and modify VIs know the value of good documentation. Document the block diagram well to ease future modification of the code. In addition, document the front panel well to explain the purpose of the VI and the front panel objects.

Use tip strips, descriptions, VI Properties, and good design to document front panels.

Tip Strips and Descriptions

Tip strips are explanations that appear when you mouse over a control or indicator. For example, you might add a tip strip saying that a temperature is in degrees Celsius or explain how the input works in an algorithm. Descriptions provide more additional information about specific controls and indicators. To add tip strips and descriptions to controls, right-click the control or indicator and select **Description and Tip** from the shortcut menu.

VI Properties

Use the Documentation component of the **VI Properties** dialog box to create VI descriptions and to link from VIs to HTML files or to compiled help files. To display VI Properties right-click the VI icon on the front panel or block diagram and select **VI Properties** from the shortcut menu or select **File»VI Properties**. Then select **Documentation** from the **Categories** drop-down menu. You cannot access this dialog box while a VI runs.

This page includes the following components:

- **VI description**—Contains the text that appears in the Context Help window if you move the cursor over the VI icon. Use `` and `` tags around any text in the description you want to format as bold. You also can use the VI Description property to edit the VI description programmatically.
- **Help tag**—Contains the HTML filename or index keyword of the topic you want to link to in a compiled help file. You also can use the Help:Document Tag property to set the help tag programmatically.

- **Help path**—Contains the path to the HTML file or to the compiled help file you want to link to from the Context Help window. If this field is empty, the blue Click here for more help. link does not appear in the Context Help window, and the More Help button is dimmed. You also can use the Help:Document Path property to set the help path programmatically.
- **Browse**—Displays a file dialog box to use to navigate to an HTML file or to a compiled help file to use as the Help path.

Naming Controls and Indicators

Giving controls and indicators logical and descriptive names adds usability to front panels. For example, if you name a control Temperature, a user may not know which units to use. However, naming a control Temperature °C adds more information to the front panel. You now know to enter temperatures in metric units.

Graphical Programming

While the graphical nature of LabVIEW aids in self-documentation of block diagrams, extra comments are helpful when modifying your VIs in the future. There are two types of block diagram comments—comments that describe the function or operation of algorithms and comments that explain the purpose of data that passes through wires. Both types of comments are shown in the following block diagram. You can insert standard labels either with the Labeling tool, or by inserting a free label from the **Functions»All Functions»Decorations** subpalette. By default, free labels have a yellow background color.

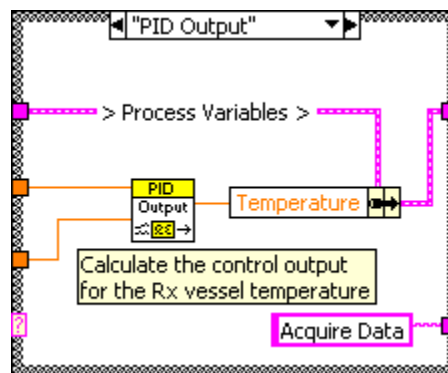


Figure 3-12. Documenting a Block Diagram

Use the following guidelines for commenting your VIs:

- Use comments on the block diagram to explain what the code is doing.
- While LabVIEW code can be self-documenting because it is graphical, use free labels to describe how the block diagram functions.

- Do not show labels on function and subVI calls because they tend to be large and unwieldy. A developer looking at the block diagram can find the name of a function or subVI by using the Context Help window.
- Use small free labels with white backgrounds to label long wires to identify their use. Labeling wires is useful for wires coming from shift registers and for long wires that span the entire block diagram. Refer to the *Case Structures* section of this lesson for more information about shift registers.
- Label structures to specify the main functionality of the structure.
- Label constants to specify the nature of the constant.
- Use free labels to document algorithms that you use on the block diagrams. If you use an algorithm from a book or other reference, provide the reference information.

Exercise 3-2 Project: Determine Warnings VI

Goal

Create and document a simple VI.

Scenario

You must create a portion of a larger project. The lead developer gives you the inputs of the VI, the algorithm and the expected outputs. Build and document a VI based on the design given.

Design

Inputs and Outputs

Type	Name	Properties
Numeric Control	Current Temp	Double-Precision
Numeric Control	Max Temp	Double-Precision
Numeric Control	Min Temp	Double-Precision
String Indicator	Warning Text	Three potential values: Heatstroke Warning, No Warning, and Freeze Warning
Round LED	Warning?	

Flowchart

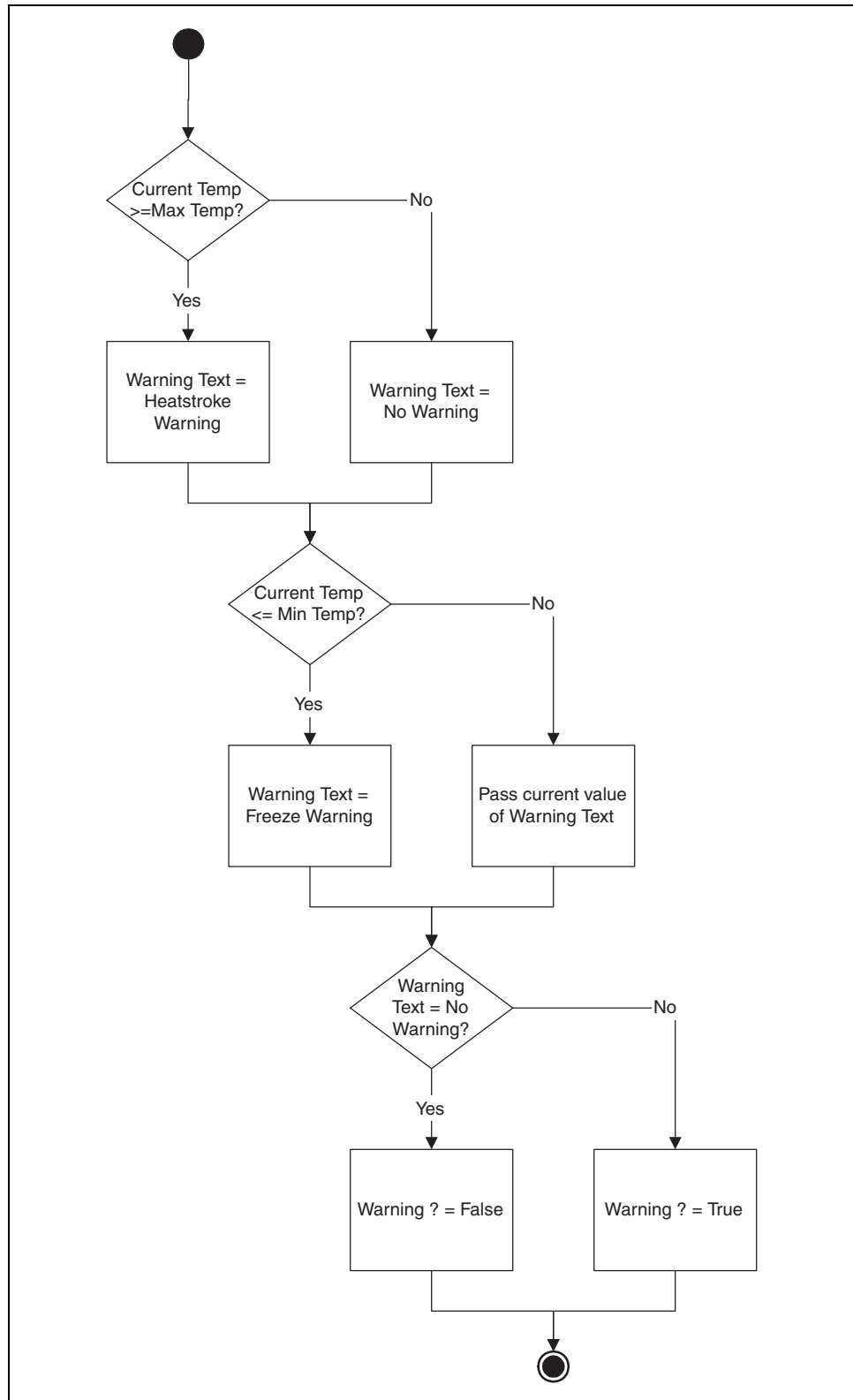


Figure 3-13. Determine Warnings VI Front Panel

Implementation

Follow the instructions given below to create a front panel similar to Figure 3-14. This front panel retrieves from the user the current temperature, the maximum temperature, and the minimum temperature, and displays to the user the warning string and the warning Boolean LED. This VI is part of the temperature weather station project you created in the previous exercise.

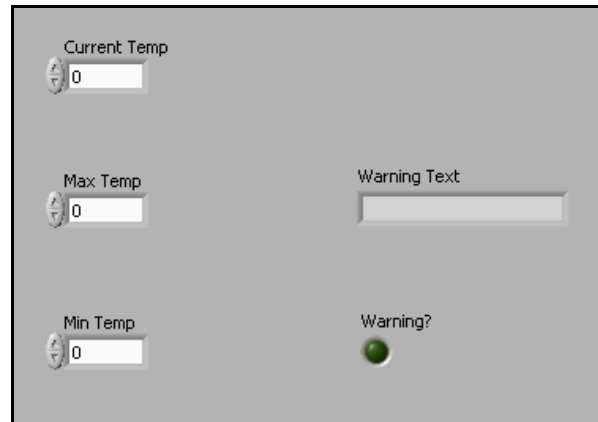


Figure 3-14. Determine Warnings VI Front Panel

1. Add a new VI to the temperature weather station project you created in the previous exercise.
 - Select **File»Open Project**.
 - Open the Weather Station.lvproj in the C:\Exercises\LabVIEW_Basics_I\Course Project directory.
 - Select **File»New VI**.
2. Save the new VI.
 - Select **File»Save**.
 - Save the VI as Determine Warnings.vi in the C:\Exercises\LabVIEW_Basics_I\Course Project directory.
3. Create a numeric control for the current temperature.
 - Place a Numeric Control on the front panel.
 - Change the label of the numeric control to Current Temp.



- Right-click the control, select **Representation**, and confirm that the representation type is set to double precision.



Tip This subVI could be used for Fahrenheit, Kelvin, or any temperature scale, as long as all inputs use the same scale. Therefore, it is not necessary to add scale units to the labels.

4. Create a numeric control for the maximum temperature.
 - Hold down the <Ctrl> key and click and drag the Current Temp numeric control to create a copy of the control.
 - Change the label text of the new numeric control to Max Temp.
5. Create a numeric control for the minimum temperature.
 - Hold down the <Ctrl> key and click and drag the Max Temp numeric control to create a copy of the control.
 - Change the label text of the new numeric control to Min Temp.
6. Create a string indicator for the warning text.
 - Place a string indicator on the front panel.
 - Change the label text of the string indicator to Warning Text.
7. Create a Round LED or other Boolean indicator for the warning Boolean.
 - Place a Round LED on the front panel.
 - Change the label text of the Boolean indicator to Warning?.
8. Switch to the block diagram.



Tip If you do not want to use the Icon Terminal view on the block diagram, select **Tools»Options**, then select **Block Diagram** from the **Category** list. Remove the checkmark from the **Place front panel terminals as icons** item.

Follow the instructions given below to build a block diagram similar to the one shown in Figure 3-15.

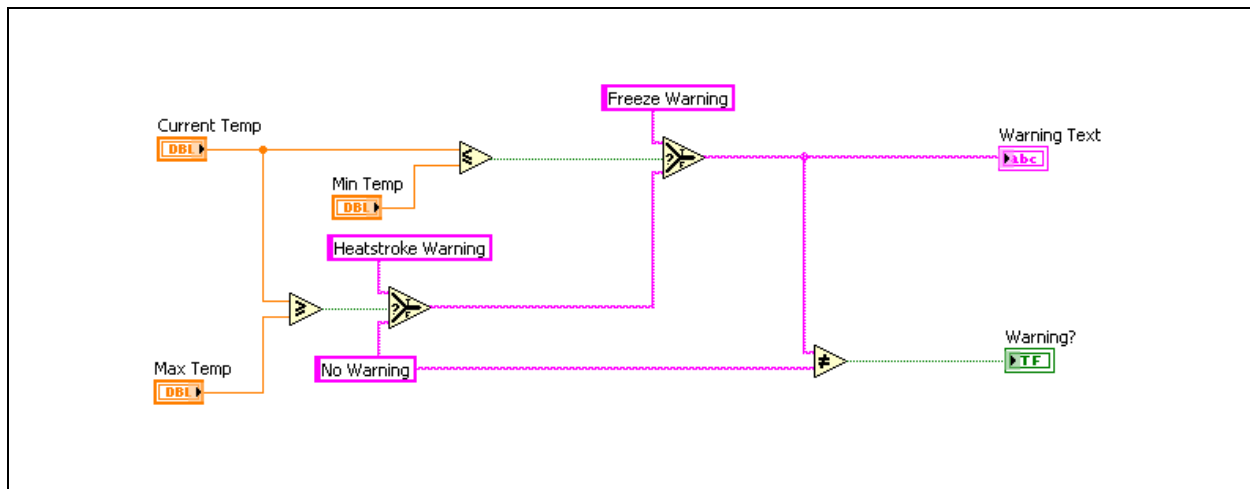


Figure 3-15. Determine Warnings VI Block Diagram

9. Compare `Current Temp` and `Max Temp`.



- Place a Greater Or Equal? function on the block diagram.
- Wire the `Current Temp` terminal to the `x` input terminal of the Greater Or Equal? function.
- Wire the `Max Temp` terminal to the `y` input terminal of the Greater Or Equal? function.

10. Compare `Current Temp` and `Min Temp`.



- Place a Less Or Equal? function on the block diagram.
- Wire the `Current Temp` terminal to the `x` input terminal of the Less Or Equal? function.
- Wire the `Min Temp` terminal to the `y` input terminal of the Less Or Equal? function.

11. If the `Current Temp` is equal to or greater than the `Max Temp`, generate a `Heatstroke Warning` string, otherwise generate a `No Warning` string.



- Place the Select function on the block diagram to the right of the Greater Or Equal? function.
- Wire the output of the Greater Or Equal? function to the `s` input terminal of the Select function.



- Place a string constant on the block diagram to the upper left of the Select function.
- Enter `Heatstroke Warning` in the string constant.
- Wire the `Heatstroke Warning` string to the **t** input of the Select function.
- Hold down the <Ctrl> key and click and drag the `Heatstroke Warning` string constant to the lower left of the Select function to create a copy of the constant.
- Enter `No Warning` in the second string constant.
- Wire the `No Warning` string to the **f** input of the Select function.

12. If the `Current Temp` is equal to or less than the `Min Temp`, generate a `Freeze Warning` string, else use the string generated in Step 11.

- Create a copy of the Select function and place it to the right of the Less Or Equal?.
- Wire the output terminal of the Less Or Equal? function to the **s** input terminal of the Select function.
- Create a copy of the string constant and place it to the upper right of the Select function.
- Enter `Freeze Warning` in the string constant.
- Wire the `Freeze Warning` string to the **t** input terminal of the Select function.
- Wire the output of the previous Select function to the **f** input terminal of the new Select function.

13. Display the generated text.

- Wire the output of the second Select function to the Warning Text indicator.

14. Generate the `Warning? Boolean` by determining if the value of `Warning Text` is equal to `No Warning`.



- Place a Not Equal function to the left of the `Warning? Boolean`.
- Wire the output of the second Select function to the **x** input terminal of the Not Equal function.

- Wire the No Warning string constant to the y input terminal of the Not Equal function.
 - Wire the output of the Not Equal function to the Warning? Boolean.
15. Document the code using the following suggestions on the front panel.
- Create tip strips for each control and indicator stating the purpose of the object and the units used. To access tip strips, right-click a control, and select **Description and Tip**.
 - Document the VI Properties giving a general description of the VI, a list of inputs and outputs, your name and the date the VI was created. To access VI Properties, select **File»VI Properties**.
 - Document the block diagram algorithm with a free label.
16. Save the VI.

Testing

1. Test the VI by inputting a value for Current Temp, Max Temp, and Min Temp, and running for each set.

Table 3-1 shows the expected Warning Text string and Warning? Boolean for each set of input values.

Table 3-1. Testing Values for Determine Warnings.vi

Current Temp	Max Temp	Min Temp	Warning Text	Warning?
30	30	10	Heatstroke Warning	True
25	30	10	No Warning	False
10	30	10	Freeze Warning	True

What happens if you input a Max Temp value that is less than the Min Temp? What would you expect to happen? You learn to handle errors like this one in a Exercise 3-7.

2. Save and close the VI.
3. Save and close the project.

End of Exercise 3-2

D. While Loops

Similar to a Do Loop or a Repeat-Until Loop in text-based programming languages, a While Loop, shown as follows, executes a subdiagram until a condition occurs.

The following illustration shows a While Loop in LabVIEW, a flowchart equivalent of the While Loop functionality, and a pseudo code example of the functionality of the While Loop.

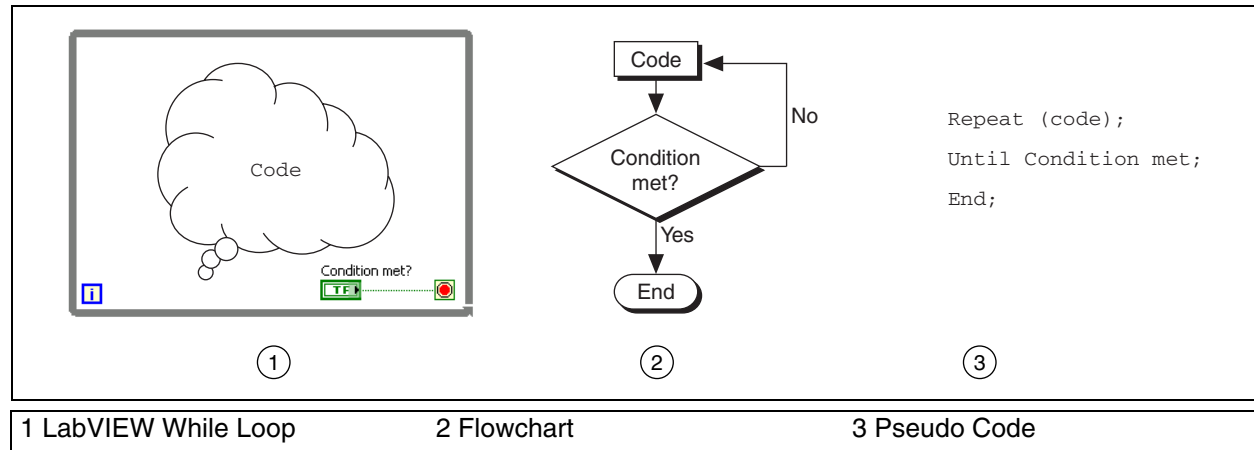


Figure 3-16. While Loop

The While Loop is located on the **Structures** palette. Select the While Loop from the palette then use the cursor to drag a selection rectangle around the section of the block diagram you want to repeat. When you release the mouse button, a While Loop boundary encloses the section you selected.

Add block diagram objects to the While Loop by dragging and dropping them inside the While Loop.



Tip The While Loop always executes at least once.

The While Loop executes the subdiagram until the conditional terminal, an input terminal, receives a specific Boolean value. The default behavior and appearance of the conditional terminal is **Stop if True**, shown as follows.



When a conditional terminal is **Stop if True**, the While Loop executes its subdiagram until the conditional terminal receives a TRUE value. You can change the behavior and appearance of the conditional terminal by right-clicking the terminal or the border of the While Loop and selecting **Continue if True**, shown as follows, from the shortcut menu.



When a conditional terminal is **Continue if True**, the While Loop executes its subdiagram until the conditional terminal receives a FALSE value. You also can use the Operating tool to click the conditional terminal to change the condition.

The iteration terminal (an output terminal), shown as follows, contains the number of completed iterations.



The iteration count always starts at zero. During the first iteration, the iteration terminal returns **0**.

In the following block diagram, the While Loop executes until the subVI output is greater than or equal to 10.00 and the **Enable** control is True. The And function returns True only if both inputs are True. Otherwise, it returns False.

In the following example, there is an increased probability of an infinite loop. Generally, the desired behavior is to have one condition met to stop the loop, rather than requiring both conditions to be met.

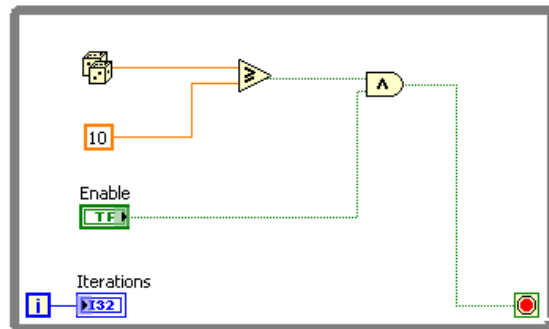


Figure 3-17. Possible Infinite Loop

Structure Tunnels

Tunnels feed data into and out of structures. The tunnel appears as a solid block on the border of the While Loop. The block is the color of the data type wired to the tunnel. Data pass out of a loop after the loop terminates. When a tunnel passes data into a loop, the loop executes only after data arrive at the tunnel.

In the following block diagram, the iteration terminal is connected to a tunnel. The value in the tunnel does not get passed to the **Iterations** indicator until the While Loop finishes executing.

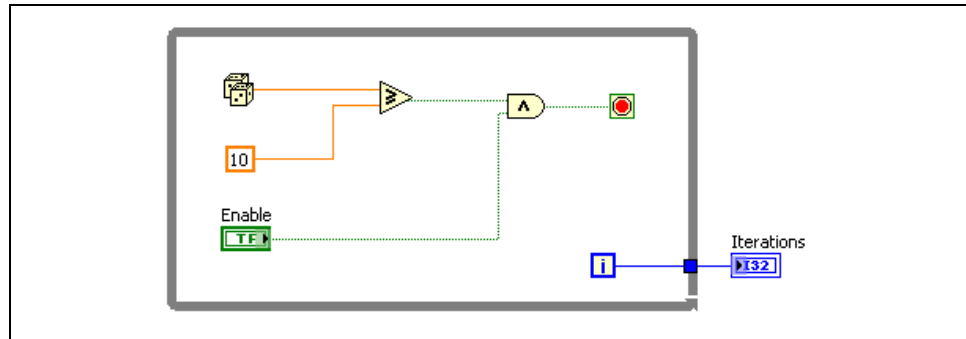


Figure 3-18. While Loop Tunnel

Only the last value of the iteration terminal displays in the **Iterations** indicator.

Exercise 3-3 Auto Match VI

Goal

Use a While Loop and an iteration terminal and pass data through a tunnel.

Scenario

Build a VI that continuously generates random numbers between 0 and 1000 until it generates a number that matches a number selected by the user. Determine how many random numbers the VI generated before the matching number.

Design

Table 3-2. Inputs and Outputs

Type	Name	Properties
Input	Number to Match	Floating Point Double between 0 and 1000 coerce to nearest whole number default value = 50
Output	Current Number	Floating Point Double
Output	Number of Iterations	Integer

Flowchart

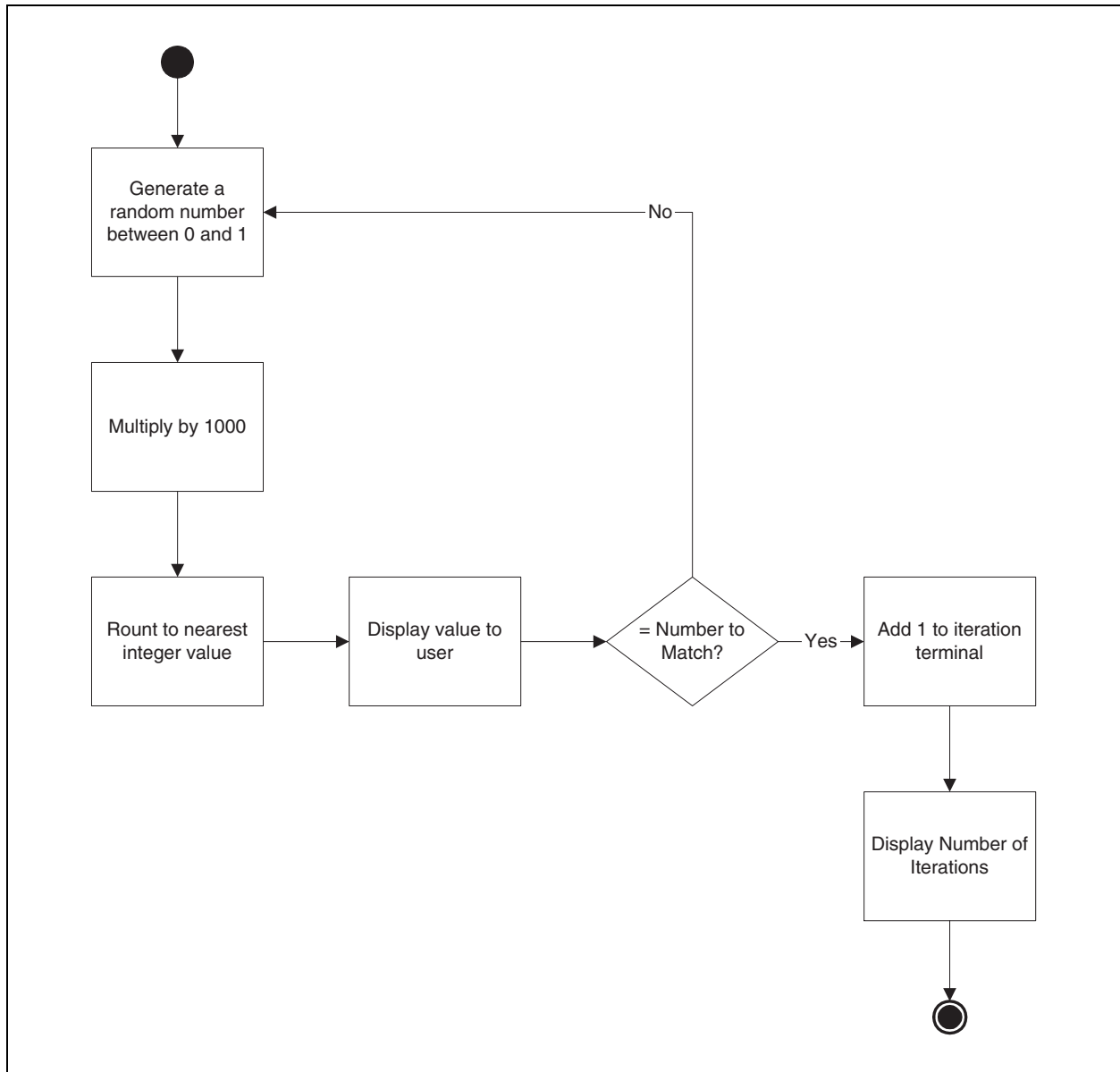
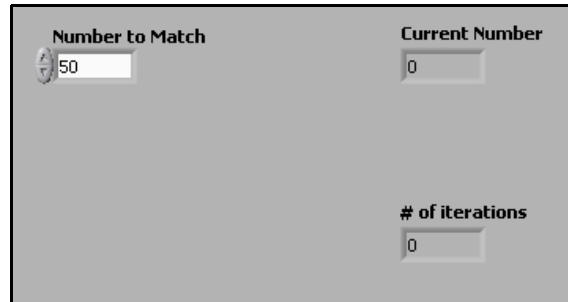


Figure 3-19. Auto Match Flowchart

Implementation

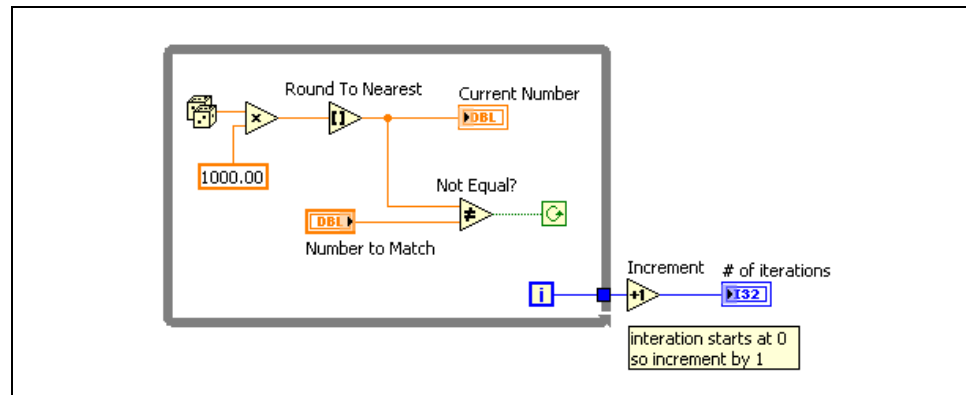
Open a blank VI and build the following front panel. Modify the controls and indicators as shown in the following front panel and as described in the following steps.



1. Create the Number to Match input.
 - Place a numeric control on the front panel.
 - Label the control `Number to Match`.
2. Set the properties for the Number to Match control so that the default value is 50, the data range is from 0 to 1000, the increment value is 1, and the digits of precision is 1.
 - Right-click the **Number to Match** control and select **Data Range** from the shortcut menu. The **Data Range** page of the **Numeric Properties** dialog box appears.
 - Remove the checkmark from the **Use Default Range** checkbox.
 - Set the **Default Value** to 50.
 - Set the **Minimum** value to 0 and select **Coerce** from the **Out of Range Action** pull-down menu.
 - Set the **Maximum** value to 1000 and select **Coerce** from the **Out of Range Action** pull-down menu.
 - Set the **Increment** value to 1 and select **Coerce to Nearest** from the **Out of Range Action** pull-down menu.
 - Select the **Format and Precision** tab.
 - Select **Floating Point** and change Precision Type from **Significant digits** to **Digits of precision**.
 - Enter 0 in the **Digits** text box and click the **OK** button.

3. Create the Current Number output.
 - Place a numeric indicator on the front panel.
 - Label the indicator `Current Number`.
4. Set the digits of precision for the Current Number output to 0.
 - Right-click the Current Number indicator and select **Format & Precision** from the shortcut menu. The **Format & Precision** page of the **Numeric Properties** dialog box appears.
 - Select **Floating Point** and change Precision Type from **Significant digits** to **Digits of precision**.
 - Enter 0 in the **Digits** text box and click the **OK** button.
5. Create the # of iterations output.
 - Place a numeric indicator on the front panel.
 - Label the indicator `# of iterations`.
6. Set the representation for the # of iterations output to a long integer.
 - Right-click the **# of iterations** indicator.
 - Select **Representation»I32** from the shortcut menu.

Build the following block diagram.



7. Generate a random number integer between 0 and 1000.



- Select the Random Number (0-1) function and place it on the block diagram. The Random Number (0-1) generates a random number between 0 and 1.



- Select the Multiply function and place it on the block diagram. The Multiply function multiplies the random number by y to produce a random number between 0 and y .
- Right-click the y input of the Multiply function, select **Create» Constant** from the shortcut menu, enter 1000, and press the <Enter> key to create a numeric constant.



- Place the Round To Nearest function on the block diagram. This function rounds the random number to the nearest integer.

8. Compare the randomly generated number to the value in the **Number to Match** control.



- Place the Not Equal? function on the block diagram. This function compares the random number with **Number to Match** and returns True if the numbers are not equal; otherwise, it returns False.

9. Repeat the algorithm until the Not Equal? function returns True.



- Place a While Loop from the **Structures** palette on the block diagram.
- Right-click the conditional terminal and select **Continue if True** from the shortcut menu.

10. Display the number of random numbers generated to the user by adding one to the iteration terminal value.



- Wire the iteration terminal to the border of the While Loop. A blue tunnel appears on the While Loop border.



Tip Each time the loop executes, the iteration terminal increments by one. Wire the iteration value to the Increment function because the iteration count starts at 0. The iteration count passes out of the loop upon completion.



- Place the Increment function on the block diagram. This function adds 1 to the While Loop count.

11. Save the VI as `Auto Match.vi` in the `C:\Exercises\LabVIEW Basics I\Automatch` directory.

Testing

1. Display the front panel.
2. Change the number in **Number to Match** to a number that is in the data range, which is 0 to 1000 with an increment of 1.
3. Run the VI.
4. Change **Number to Match** and run the VI again. **Current Number** updates at every iteration of the loop because it is inside the loop. **# of iterations** updates upon completion because it is outside the loop.
5. To see how the VI updates the indicators, enable execution highlighting.



- On the block diagram toolbar, click the **Highlight Execution** button, shown at left, to enable execution highlighting. Execution highlighting shows the movement of data on the block diagram from one node to another so you can see each number as the VI generates it.
6. Run the VI and observe the dataflow.
 7. Try to match a number that is outside of the data range.
 8. Change **Number to Match** to a number that is out of the data range.
 - Run the VI. LabVIEW coerces the out-of-range value to the nearest value in the specified data range.
 9. Close the VI.

End of Exercise 3-3

E. For Loops

A For Loop, shown as follows, executes a subdiagram a set number of times. The following illustration shows a For Loop in LabVIEW, a flowchart equivalent of the For Loop functionality, and a pseudo code example of the functionality of the For Loop.

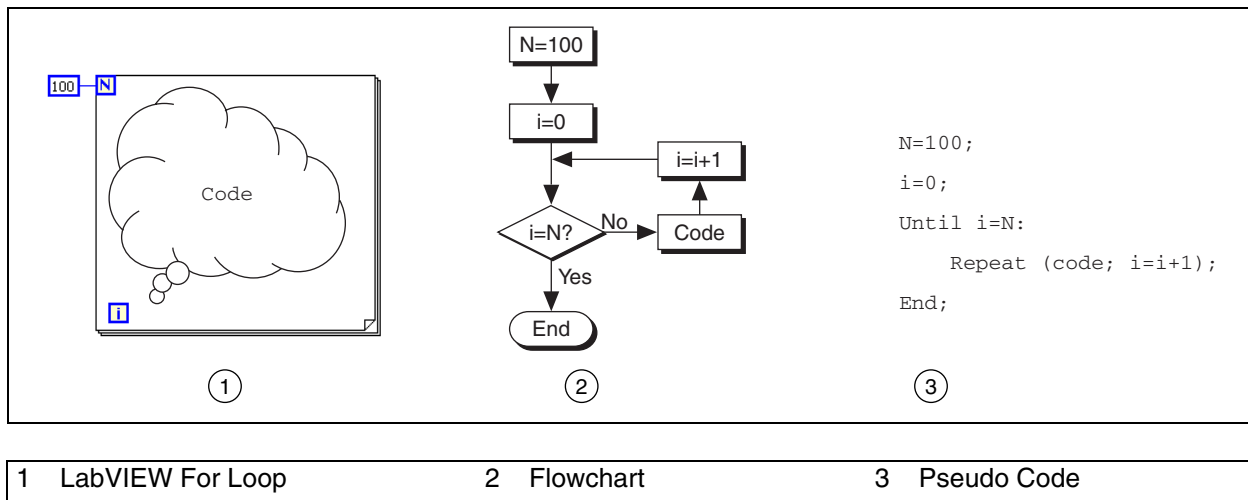


Figure 3-20. For Loop

The For Loop is located on the **Functions»All Functions»Structures** palette. You also can place a While Loop on the block diagram, right-click the border of the While Loop, and select **Replace with For Loop** from the shortcut menu to change a While Loop to a For Loop. The value in the count terminal (an input terminal), shown as follows, indicates how many times to repeat the subdiagram.

The iteration terminal (an output terminal), shown as follows, contains the number of completed iterations.



The iteration count always starts at zero. During the first iteration, the iteration terminal returns **0**.

The For Loop differs from the While Loop in that the For Loop executes a set number of times. A While Loop stops executing the subdiagram only if the value at the conditional terminal exists.

The following For Loop generates a random number every second for 100 seconds and displays the random numbers in a numeric indicator.

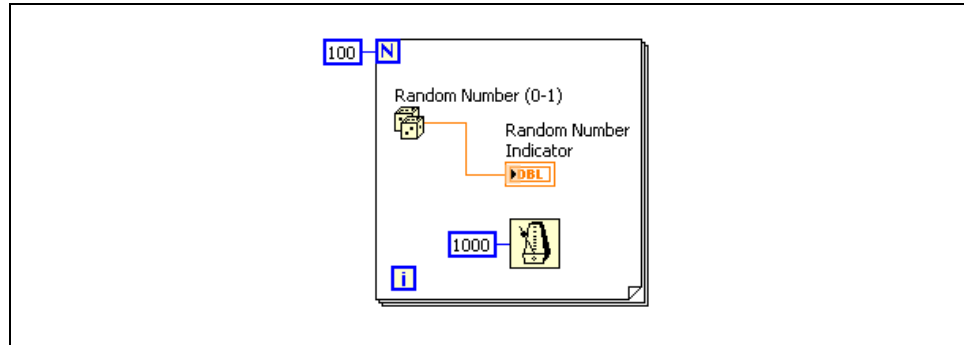
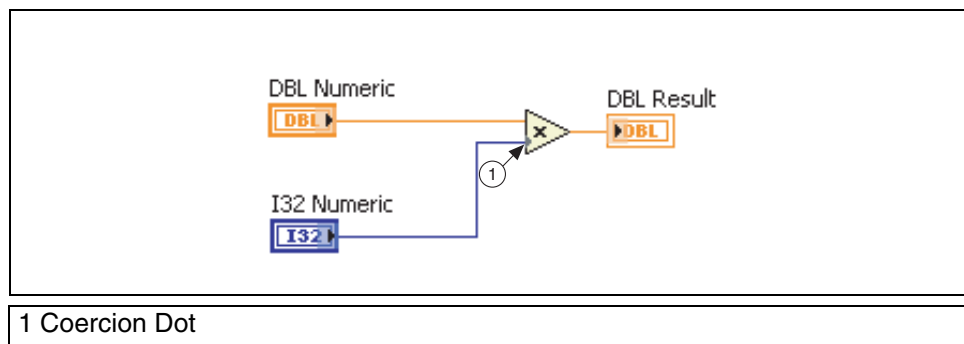


Figure 3-21. For Loop Example

Numeric Conversion

LabVIEW can represent numeric data types as signed or unsigned integers, floating-point numeric values, or complex numeric values, as discussed in the *LabVIEW Data Types* section of this lesson. Normally, when you wire different representation types to the inputs of a function, the function returns an output in the larger or wider format. LabVIEW chooses the representation that uses more bits. If the number of bits is the same, LabVIEW chooses unsigned over signed. For example, if you wire a DBL and an I32 to a Multiply function, the result is a DBL, as shown in Figure 3-22. The 64-bit signed integer is coerced because it uses fewer bits than the double-precision, floating-point numeric value. The lower input of the Multiply function shows a grey dot, called a coercion dot, that indicates a numeric coercion occurred.



1 Coercion Dot

Figure 3-22. Numeric Conversion Example

However, the For Loop count terminal works in the opposite manner. If you wire a double-precision, floating-point numeric value to the 64-bit count terminal, LabVIEW converts the larger numeric value to a 32-bit signed integer. Although the conversion is contrary to normal conversion standards, it is necessary, as a For Loop can only execute an integer number of times.

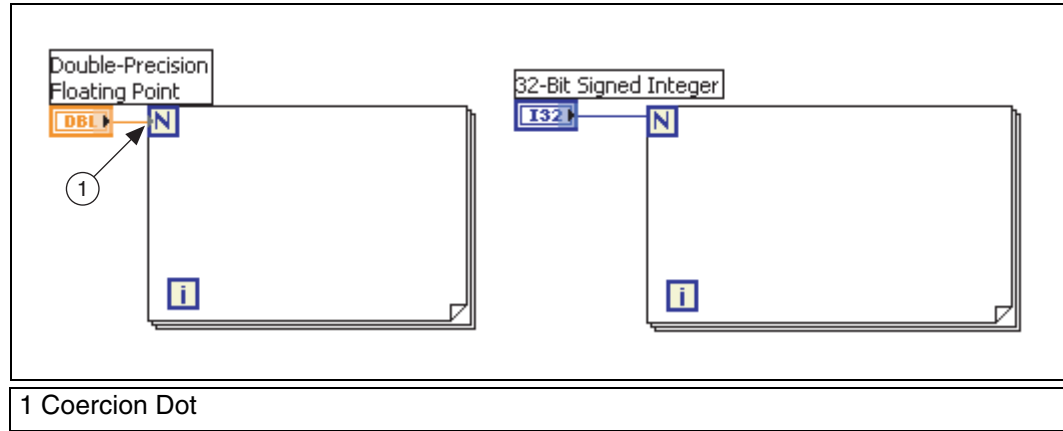


Figure 3-23. Coercion on a For Loop

Exercise 3-4 Concept: While Loops versus For Loops

Goal

Understand when to use a While Loop and when to use a For Loop.

Description

For the following scenarios, decide whether to use a While Loop or a For Loop.

Scenario 1

Acquire a pressure every second for one minute.

1. If you use a While Loop, what is the condition that you need to stop the loop?
2. If you use a For Loop, how many iterations does the loop need to run?
3. Is it easier to implement a For Loop or a While Loop?

Scenario 2

Acquire a pressure until the pressure is 1400 psi.

1. If you use a While Loop, what is the condition that you need to stop the loop?
2. If you use a For Loop, how many iterations does the loop need to run?
3. Is it easier to implement a For Loop or a While Loop?

Scenario 3

Acquire a pressure and a temperature until both values are stable for two minutes.

1. If you use a While Loop, what is the condition that you need to stop the loop?
2. If you use a For Loop, how many iterations does the loop need to run?
3. Is it easier to implement a For Loop or a While Loop?

Scenario 4

Output a voltage ramp starting at zero, increasing incremental by 0.5 V every second, until the output voltage is equal to 5 V.

1. If you use a While Loop, what is the condition that you need to stop the loop?
2. If you use a For Loop, how many iterations does the loop need to run?
3. Is it easier to implement a For Loop or a While Loop?

Answers

Scenario 1

Acquire a pressure every second for one minute.

1. While Loop: Time = 1 minute
2. For Loop: 60 iterations
3. Both are possible.

Scenario 2

Acquire a pressure until the pressure is 1400 psi.

1. While Loop: Pressure = 1400 psi
2. For Loop: unknown
3. A While Loop. Without more information, a For Loop is impossible.

Scenario 3

Acquire a pressure and a temperature until both values are stable for two minutes.

1. While Loop: [(Last Temperature = Previous Temperature) for 2 minutes or more] AND [(Last Pressure = Previous Pressure) for 2 minutes or more]
2. For Loop: unknown
3. A While Loop. Without more information, a For Loop is impossible.

Scenario 4

Output a voltage ramp starting at zero, increasing incremental by 0.5 V every second, until the output voltage is equal to 5 V.

1. While Loop: Voltage = 5 V
2. For Loop: 11 iterations
3. Either one is a good choice.

End of Exercise 3-4

F. Timing a VI

When a loop finishes executing an iteration, it immediately begins executing the next iteration, unless it reaches a stop condition. Most often, you need to control the frequency or timing of the iteration. For example, if you are acquiring data, and you want to acquire the data once every 10 seconds, you need a way to time the loop iterations so they occur once every 10 seconds.

Even if you do not need the execution to occur at a certain frequency, you need to provide the processor with time to complete other tasks, such as processing the user interface. This section introduces some methods for timing your loops.

Wait Functions

Place a wait function are placed inside a loop to allow a VI to sleep for a set amount of time. This allows your processor to address other tasks during the wait time. Wait functions use the operating system millisecond clock.



The Wait Until Next ms Multiple function, shown at left, monitors a millisecond counter and waits until the millisecond counter reaches a multiple of the amount you specify. Use this function to synchronize activities. Place this function in a loop to control the loop execution rate. For this function to be effective, your code execution time must be less than the time specified for this function. The execution rate for the first iteration of the loop is indeterminate.



The Wait (ms) function, shown at left, waits until the millisecond counter counts to an amount equal to the input you specify. This function guarantees that the loop execution rate is at least the amount of the input you specify.



Note The Time Delay Express VI behaves similar to the Wait (ms) function with the addition of built-in error clusters. Refer to Lesson 4, *Implementing a VI* for more information about error clusters.

Elapsed Time



In some cases, it is useful to determine how much time elapses after some point in your VI. The Elapsed Time Express VI, shown at left, indicates the amount of time that elapses after the specified start time. This VI allows you to keep track of time while the VI continues to execute. This function does not provide the processor with time to complete other tasks. You learn more about this Express VI, as you use it in the Weather Station course project.

G. Iterative Data Transfer

When programming with loops, you often must access data from previous iterations of the loop. For example, if you are acquiring one piece of data in each iteration of a loop and must average every five pieces of data, you must remember the data from previous iterations of the loop. Shift registers transfer values from one loop iteration to the next.



Note Feedback Nodes are another method used in LabVIEW for retaining information from a previous iteration. Refer to the Feedback Node topic in the *LabVIEW Help* for more information about feedback nodes.

Shift registers are similar to static variables in text-based programming languages.

Use shift registers when you want to pass values from previous iterations through the loop to the next iteration. A shift register appears as a pair of terminals, shown as follows, directly opposite each other on the vertical sides of the loop border.

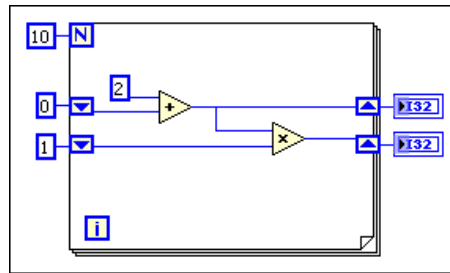


The terminal on the right side of the loop contains an up arrow and stores data on the completion of an iteration. LabVIEW transfers the data connected to the right side of the register to the next iteration. After the loop executes, the terminal on the right side of the loop returns the last value stored in the shift register.

Create a shift register by right-clicking the left or right border of a loop and selecting **Add Shift Register** from the shortcut menu.

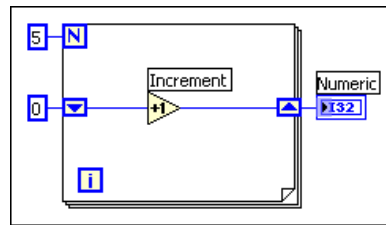
A shift register transfers any data type and automatically changes to the data type of the first object wired to the shift register. The data you wire to the terminals of each shift register must be the same type.

You can add more than one shift register to a loop. If you have multiple operations that use previous iteration values within your loop, use multiple shift registers to store the data values from those different processes in the structure, as shown in the following figure.



Initializing Shift Registers

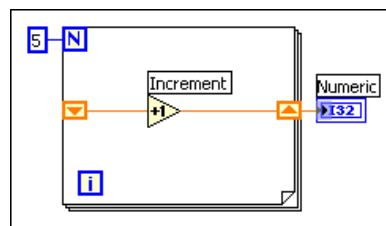
Initializing a shift register resets the value the shift register passes to the first iteration of the loop when the VI runs. Initialize a shift register by wiring a control or constant to the shift register terminal on the left side of the loop, as shown in the following figure.



In the previous figure, the For Loop executes five times, incrementing the value the shift register carries by one each time. After five iterations of the For Loop, the shift register passes the final value, 5, to the indicator and the VI quits. Each time you run the VI, the shift register begins with a value of 0.

If you do not initialize the shift register, the loop uses the value written to the shift register when the loop last executed or the default value for the data type if the loop has never executed.

Use an uninitialized shift register to preserve state information between subsequent executions of a VI. The following figure shows an uninitialized shift register.



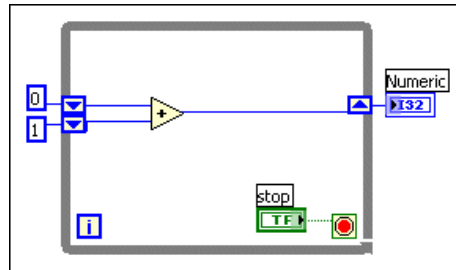
In the previous figure, the For Loop executes five times, incrementing the value the shift register carries by one each time. The first time you run the

VI, the shift register begins with a value of 0, which is the default value for a 32-bit integer. After five iterations of the For Loop, the shift register passes the final value, 5, to the indicator, and the VI quits. The next time you run the VI, the shift register begins with a value of 5, which was the last value from the previous execution. After five iterations of the For Loop, the shift register passes the final value, 10, to the indicator. If you run the VI again, the shift register begins with a value of 10, and so on. Uninitialized shift registers retain the value of the previous iteration until you close the VI.

Stacked Shift Registers

Stacked shift registers let you access data from previous loop iterations. Stacked shift registers remember values from multiple previous iterations and carry those values to the next iterations. To create a stacked shift register, right-click the left terminal and select **Add Element** from the shortcut menu.

Stacked shift registers can occur only on the left side of the loop because the right terminal transfers the data generated only from the current iteration to the next iteration, as shown in the following figure.



If you add another element to the left terminal in the previous figure, values from the last two iterations carry over to the next iteration, with the most recent iteration value stored in the top shift register. The bottom terminal stores the data passed to it from the previous iteration.

Exercise 3-5 Average Temperature VI

Goal

Use a For Loop and shift registers to average data.

Scenario

The Temperature Monitor VI acquires and displays temperature. Modify the VI to average the last three temperature measurements and display the running average on the waveform chart.

Design

Figure 3-24 and Figure 3-25 shows the Temperature Monitor VI front panel and block diagram.

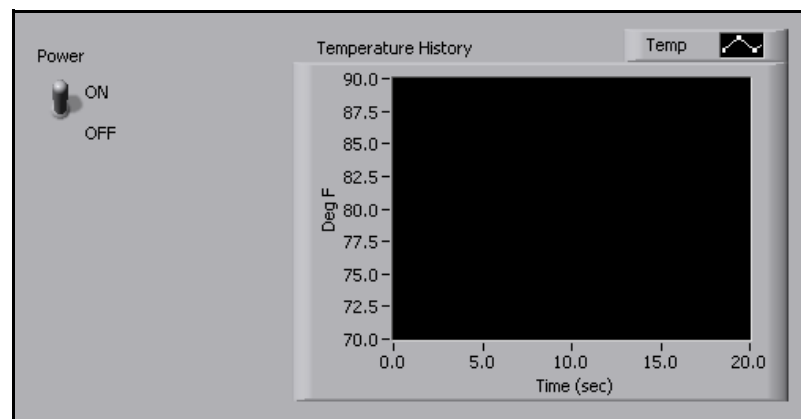


Figure 3-24. Temperature Monitor VI Front Panel

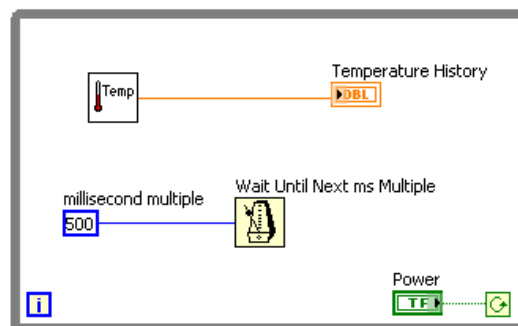


Figure 3-25. Temperature Monitor VI Block Diagram

To modify this VI, you need to retain the temperature values from the previous two iterations, and average the values. Use a shift register with an additional element to retain data from the previous two iterations. Initialize the shift register with a reading from the temperature sensor. Chart only the average temperature.

Implementation

1. Test the VI.

Hardware

- Open the `Temperature Monitor.vi` in the `C:\Exercises\LabVIEW Basics I\Average Temperature` directory.
- Select **File»Save As** and save the VI as `Average Temperature.vi` in the `C:\Exercises\LabVIEW Basics I\Average Temperature` directory.
- On the DAQ Signal Accessory, flip the temperature sensor noise switch to On. This switch introduces noise to the temperature reading.
- Run the VI.
- Place your finger on the temperature sensor of the DAQ Signal Accessory to increase the temperature reading. You can quickly move your finger across the sensor to increase the reading even more through friction. Notice the number of spikes in the reading.
- Skip to step 2.

No Hardware

- Open `Temperature Monitor (Demo).vi` in the `C:\Exercises\LabVIEW Basics I\Average Temperature` directory.
 - Select **File»Save As** and rename the VI `Average Temperature.vi` in the `C:\Exercises\LabVIEW Basics I\Average Temperature` directory.
 - Run the VI. Notice the variation in the simulated temperature reading.
2. Stop the VI by changing the state of the Power switch. Notice that the Power switch immediately switches back to the On state. The mechanical action of the switch controls this behavior.

In the following steps, modify the VI to reduce the number of temperature spikes.

3. Display the block diagram.
4. Modify the block diagram as shown in Figure 3-26.

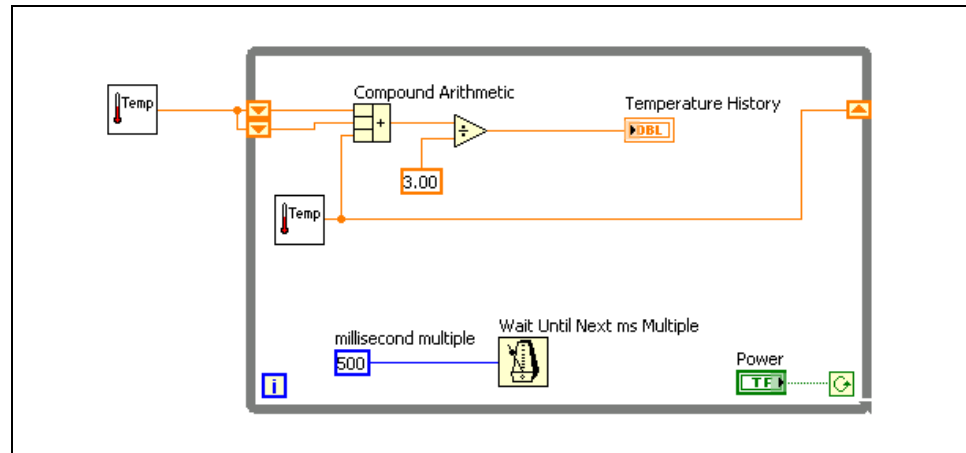


Figure 3-26. Average Temperature VI Block Diagram

- Right-click the right or left border of the While Loop and select **Add Shift Register** from the shortcut menu to create a shift register.
- Right-click the left terminal of the shift register and select **Add Element** from the shortcut menu to add an element to the shift register.



- Press the <Ctrl> key while you click the Thermometer VI and drag it outside the While Loop to create a copy of the subVI.

The Thermometer VI returns one temperature measurement from the temperature sensor and initializes the left shift registers before the loop starts.



- Place the Compound Arithmetic function on the block diagram.

Configure this function to return the sum of the current temperature and the two previous temperature readings.

- Use the Positioning tool to resize the Compound Arithmetic function to have three left terminals.



- Place the Divide function on the block diagram. This function returns the average of the last three temperature readings.

- Wire the functions together as shown in Figure 3-26.

3.00

- Right-click the y terminal of the Divide function, select **Create» Constant**.
 - Enter 3 and press the <Enter> key.
5. Save the VI.

Testing

1. Run the VI.
2. Place your finger of the temperature sensor on the DAQ Signal Accessory to increase the temperature reading.

During each iteration of the While Loop, the Thermometer VI takes one temperature measurement. The VI adds this value to the last two measurements stored in the left terminals of the shift register. The VI divides the result by three to find the average of the three measurements, the current measurement plus the previous two. The VI displays the average on the waveform chart. Notice that the VI initializes the shift register with a temperature measurement.
3. Stop the VI by changing the state of the Power switch.
4. Close the VI.

End of Exercise 3-5

H. Plotting Data

You already used charts and graphs to plot simple data. This section explains more about using and customizing charts and graphs.

Waveform Charts

The waveform chart is a special type of numeric indicator that displays one or more plots of data typically acquired at a constant rate. Waveform charts can display single or multiple plots. Figure 3-27 shows the elements of a multiplot waveform chart. Two plots are displayed: Raw Data and Running Avg.

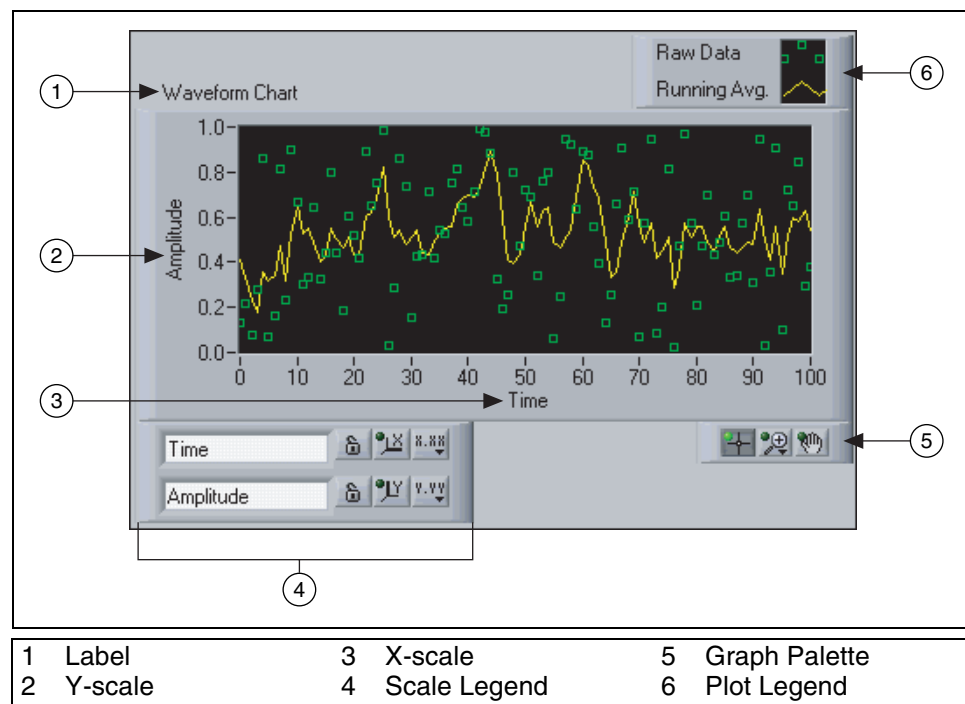


Figure 3-27. Waveform Charts

You can configure how the chart updates to display new data. Right-click the chart and select **Advanced»Update Mode** from the shortcut menu to set the chart update mode. The chart uses the following modes to display data:

- **Strip Chart**—Shows running data continuously scrolling from left to right across the chart with old data on the left and new data on the right. A strip chart is similar to a paper tape strip chart recorder. **Strip Chart** is the default update mode.
- **Scope Chart**—Shows one item of data, such as a pulse or wave, scrolling partway across the chart from left to right. For each new value, the chart plots the value to the right of the last value. When the plot reaches the right border of the plotting area, LabVIEW erases the plot

and begins plotting again from the left border. The retracing display of a scope chart is similar to an oscilloscope.

- **Sweep Chart**—Works similarly to a scope chart except it shows the old data on the right and the new data on the left separated by a vertical line. LabVIEW does not erase the plot in a sweep chart when the plot reaches the right border of the plotting area. A sweep chart is similar to an EKG display.

Figure 3-28 shows an example of each chart update mode. The scope chart and sweep chart have retracing displays similar to an oscilloscope. Because retracing a plot requires less overhead, the scope chart and the sweep chart display plots significantly faster than the strip chart.

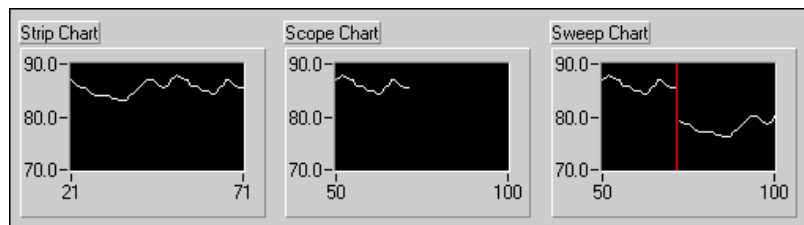


Figure 3-28. Chart Update Modes

Wiring Charts

You can wire a scalar output directly to a waveform chart. The waveform chart terminal shown in Figure 3-29 matches the input data type.

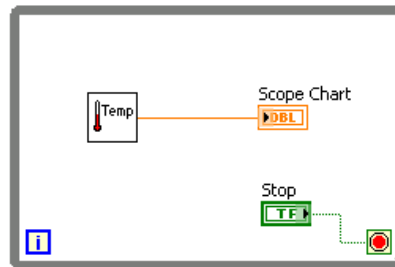


Figure 3-29. Wiring a Single Plot to a Waveform Chart

Waveform charts can display multiple plots together using the Bundle function located on the **Cluster** palette. In Figure 3-30, the Bundle function bundles the outputs of the three VIs to plot on the waveform chart.

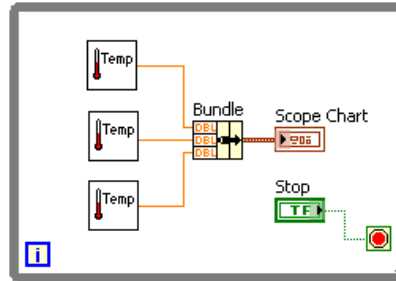


Figure 3-30. Wiring multiple plots to a Waveform Chart

The waveform chart terminal changes to match the output of the Bundle function. To add more plots, use the Positioning tool to resize the Bundle function. Refer to Lesson 4, *Relating Data* for more information about bundles.

Waveform Graphs

VIs with a graph usually collect the data in an array and then plot the data to the graph. Figure 3-31 shows the elements of a graph.

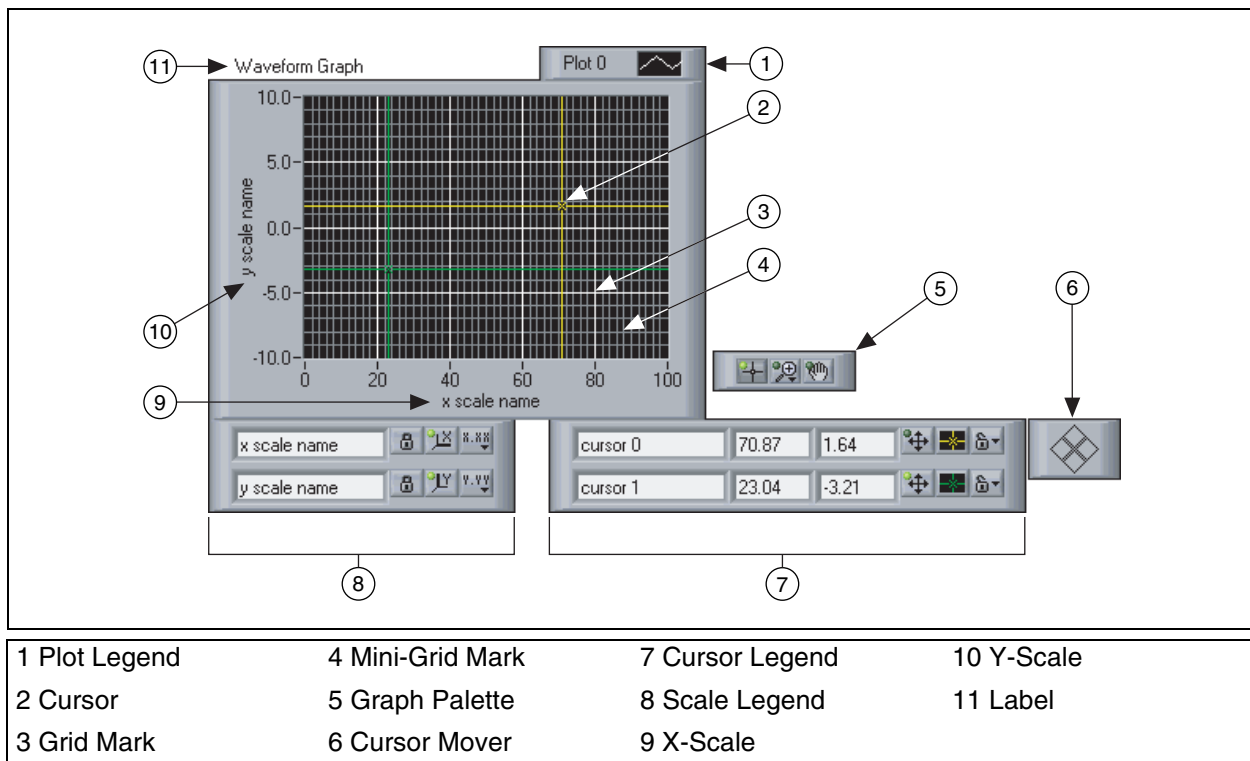


Figure 3-31. Waveform Graph

The graphs located on the Graph Indicators palette include the waveform graph and XY graph. The waveform graph plots only single-valued functions, as in $y = f(x)$, with points evenly distributed along the x-axis,

such as acquired time-varying waveforms. XY graphs display any set of points, evenly sampled or not.

Resize the plot legend to display multiple plots. Use multiple plots to save space on the front panel and to make comparisons between plots. XY and waveform graphs automatically adapt to multiple plots.

Single Plot Waveform Graphs

The waveform graph accepts several data types for single-plot waveform graphs. The graph accepts a single array of values, interprets the data as points on the graph, and increments the x index by one starting at $x = 0$. The graph accepts a cluster of an initial x value, a delta x , and an array of y data. The graph also accepts the waveform data type, which carries the data, start time, and delta t of a waveform.

Refer to the Waveform Graph VI in the `labview\examples\general\graphs\gengraph.llb` for examples of the data types that a waveform graph accepts.

Multiplot Waveform Graphs

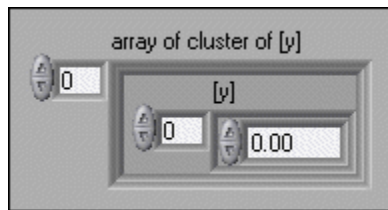
The waveform graph accepts several data types for displaying multiple plots. The waveform graph accepts a 2D array of values, where each row of the array is a single plot. The graph interprets the data as points on the graph and increments the x index by one, starting at $x = 0$. Wire a 2D array data type to the graph, right-click the graph, and select **Transpose Array** from the shortcut menu to handle each column of the array as a plot. This is particularly useful when you sample multiple channels from a DAQ device because the device can return the data as 2D arrays with each channel stored as a separate column.

Refer to the (Y) Multi Plot 1 graph in the Waveform Graph VI in the `labview\examples\general\graphs\gengraph.llb` for an example of a graph that accepts this data type.

The waveform graph also accepts a cluster of an initial x value, a delta x value, and a 2D array of y data. The graph interprets the y data as points on the graph and increments the x index by delta x , starting at the initial x value. This data type is useful for displaying multiple signals that are sampled at the same regular rate. Refer to the (Xo = 10, dX = 2, Y) Multi Plot 2 graph in the Waveform Graph VI in the `labview\examples\general\graphs\gengraph.llb` for an example of a graph that accepts this data type.

The waveform graph accepts a plot array where the array contains clusters. Each cluster contains a 1D array that contains the y data. The inner array

describes the points in a plot, and the outer array has one cluster for each plot. The following front panel shows this array of the y cluster.



Use a plot array instead of a 2D array if the number of elements in each plot is different. For example, when you sample data from several channels using different time amounts from each channel, use this data structure instead of a 2D array because each row of a 2D array must have the same number of elements. The number of elements in the interior arrays of an array of clusters can vary. Refer to the (Y) Multi Plot 2 graph in the Waveform Graph VI in the `labview\examples\general\graphs\gengraph.llb` for an example of a graph that accepts this data type.

The waveform graph accepts a cluster of an initial x value, a delta x value, and array that contains clusters. Each cluster contains a 1D array that contains the y data. You use the Bundle function to bundle the arrays into clusters and you use the Build Array function to build the resulting clusters into an array. You also can use the Build Cluster Array function, which creates arrays of clusters that contain the inputs you specify. Refer to the (Xo = 10, dX = 2, Y) Multi Plot 3 graph in the Waveform Graph VI in the `labview\examples\general\graphs\gengraph.llb` for an example of a graph that accepts this data type.

The waveform graph accepts an array of clusters of an x value, a delta x value, and an array of y data. This is the most general of the multiple-plot waveform graph data types because you can indicate a unique starting point and increment for the x -scale of each plot. Refer to the (Xo = 10, dX = 2, Y) Multi Plot 1 graph in the Waveform Graph VI in the `labview\examples\general\graphs\gengraph.llb` for an example of a graph that accepts this data type.

The waveform graph also accepts the dynamic data type, which is for use with Express VIs. In addition to the data associated with a signal, the dynamic data type includes attributes that provide information about the signal, such as the name of the signal or the date and time the data was acquired. Attributes specify how the signal appears on the waveform graph. When the dynamic data type includes multiple channels, the graph displays a plot for each channel and automatically formats the plot legend and x -scale time stamp.

Single Plot XY Graphs

The XY graph accepts three data types for single-plot XY graphs. The XY graph accepts a cluster that contains an x array and a y array. Refer to the (X and Y arrays) Single Plot graph in the XY Graph VI in the `labview\examples\general\graphs\gengraph.llb` for an example of a graph that accepts this data type.

The XY graph also accepts an array of points, where a point is a cluster that contains an x value and a y value. Refer to the (Array of Pts) Single Plot graph in the XY Graph VI in the `labview\examples\general\graphs\gengraph.llb` for an example of a graph that accepts this data type. The XY graph also accepts an array of complex data, in which the real part is plotted on the x -axis and the imaginary part is plotted on the y -axis.

Multiplot XY Graphs

The XY graph accepts three data types for displaying multiple plots. The XY graph accepts an array of plots, where a plot is a cluster that contains an x array and a y array. Refer to the (X and Y arrays) Multi Plot graph in the XY Graph VI in the `labview\examples\general\graphs\gengraph.llb` for an example of a graph that accepts this data type.

The XY graph also accepts an array of clusters of plots, where a plot is an array of points. A point is a cluster that contains an x value and a y value. Refer to the (Array of Pts) Multi Plot graph in the XY Graph VI in the `labview\examples\general\graphs\gengraph.llb` for an example of a graph that accepts this data type. The XY graph also accepts an array of clusters of plots, where a plot is an array of complex data, in which the real part is plotted on the x -axis and the imaginary part is plotted on the y -axis.

Exercise 3-6 Temperature Multiplot VI

Goal

Plot multiple data sets on a single waveform chart and customize the chart view.

Scenario

Modify the VI from Exercise 3-5 to plot both the current temperature and the running average on the same chart. In addition, allow the user to examine a portion of the plot while the data is being acquired.

Design

Figure 3-32 shows the front panel for the existing VI (Average Temperature VI) and Figure 3-33 shows the block diagram.

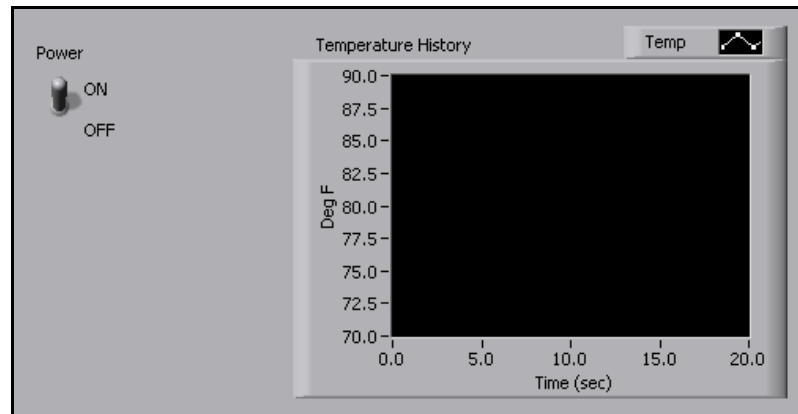


Figure 3-32. Average Temperature VI Front Panel

To allow the user to examine a portion of the plot while the data is being acquired, display the Scale Legend and Graph Palette for the waveform chart. Also, expand the legend to show additional plots.

To modify the block diagram in Figure 3-33, you must modify the chart terminal to accept multiple pieces of data. Use a Bundle function to combine the average temperature and the current temperature into a cluster to pass to the Temperature History chart terminal.

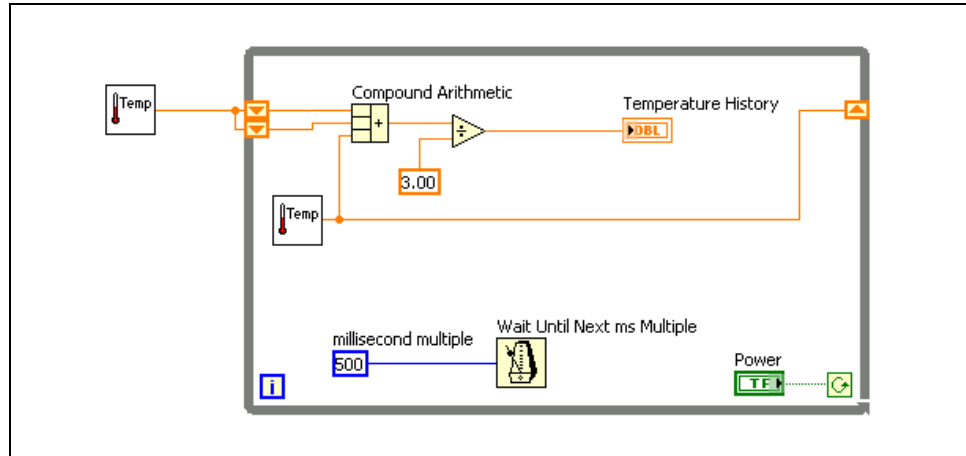


Figure 3-33. Average Temperature VI Block Diagram

Modify the block diagram first, then modify the front panel.

Implementation

1. Open `Average Temperature.vi` in the `C:\Exercises\LabVIEW Basics I\Average Temperature` directory. You created this VI in the previous exercise.
2. Select **File»Save As** and rename the VI `Temperature Multiplot.vi` in the `C:\Exercises\LabVIEW Basics I\Temperature Multiplot` directory. Create the directory if it does not exist. Select **Substitute Copy for Original** option.

In the steps below, you modify the block diagram similar to that shown in Figure 3-35.

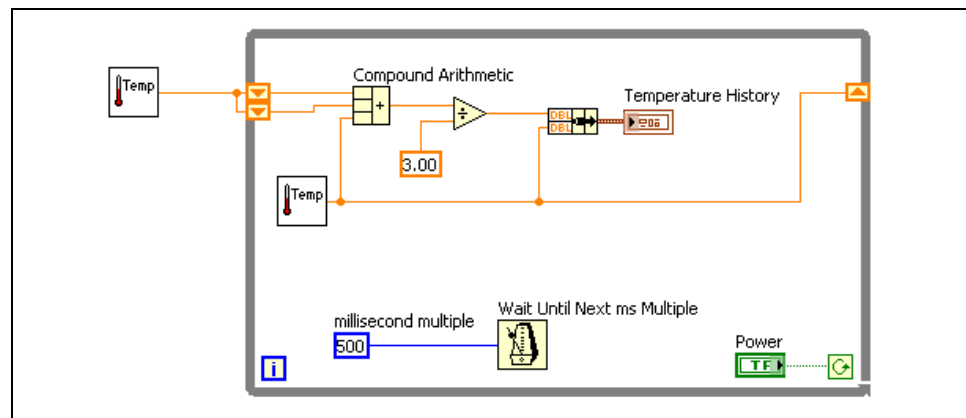


Figure 3-34. Temperature Multiplot VI Block Diagram

3. Open the block diagram.
4. Pass the current temperature and the average temperature to the Temperature History chart terminal.
 - Delete the wire connecting the Divide function to the Temperature History chart terminal.
 - Place a Bundle function between the Divide function and the Temperature History chart terminal. If necessary, enlarge the While Loop to make space.
 - Wire the output of the Divide function to the top input of the Bundle function.
 - Wire the current temperature to the bottom input of the Bundle function. The current temperature is the output of the Thermometer subVI inside the While Loop.

- Wire the output of the Bundle function to the Temperature History chart terminal.

In the following steps, modify the front panel similar to the one shown in Figure 3-34.

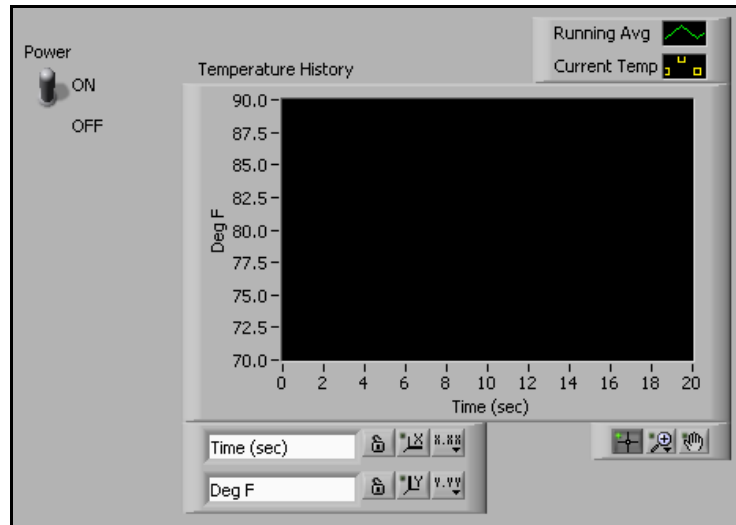


Figure 3-35. Temperature Multiplot VI Front Panel

5. Open the front panel.
6. Show both plots in the plot legend of the Waveform Chart.
 - Use the Positioning tool to resize the plot legend to two objects, using the top middle resizing node.
 - Rename the top plot `Running Avg`.
 - Rename the bottom plot `Current Temp`.
 - Change the plot type of `Current Temp`. Use the Operating tool to select the plot in the plot legend and choose the plots you want.



Tip The order of the plots listed in the plot legend is the same as the order of the items wired to the Bundle function on the block diagram.

7. Show the Scale Legend and Graph Palette of the Waveform Chart.
 - Right-click the Temperature History Waveform Chart and select **Visible Items»Scale Legend** from the shortcut menu.
 - Right-click the Temperature History Waveform Chart and select **Visible Items»Graph Palette** from the shortcut menu.

8. Save the VI.

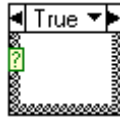
Testing

1. Run the VI. Use the tools in the Scale Legend and Graph palette to examine the data as it is generated.
2. Change the Power switch to Off to stop the VI.
3. Close the VI when you are finished.

End of Exercise 3-6

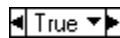
I. Case Structures

A Case structure, shown as follows, has two or more subdiagrams, or cases.



Only one subdiagram is visible at a time, and the structure executes only one case at a time. An input value determines which subdiagram executes. The Case structure is similar to switch statements or if...then...else statements in text-based programming languages.

The case selector label at the top of the Case structure, shown as follows, contains the name of the selector value that corresponds to the case in the center and decrement and increment arrows on each side.



Click the decrement and increment arrows to scroll through the available cases. You also can click the down arrow next to the case name and select a case from the pull-down menu.

Wire an input value, or selector, to the selector terminal, shown as follows, to determine which case executes.



You must wire an integer, Boolean value, string, or enumerated type value to the selector terminal. You can position the selector terminal anywhere on the left border of the Case structure. If the data type of the selector terminal is Boolean, the structure has a `TRUE` case and a `FALSE` case. If the selector terminal is an integer, string, or enumerated type value, the structure can have any number of cases.

Specify a default case for the Case structure to handle out-of-range values. Otherwise, you must explicitly list every possible input value. For example, if the selector is an integer and you specify cases for 1, 2, and 3, you must specify a default case to execute if the input value is 4 or any other unspecified integer value.

Right-click the Case structure border to add, duplicate, remove, or rearrange cases and to select a default case.

Selecting a Case

Figure 3-36 shows a VI that uses a Case structure to execute different code dependent on whether a user selects degrees Celsius or Fahrenheit for their desired temperature units. The top block diagram shows the True case in the foreground. In the middle block diagram, the False case is selected. To select a case, enter the value in the case selector identifier or use the Labeling tool to edit the values. After you select another case, that case appears at the front, as shown in the bottom block diagram of Figure 3-36.

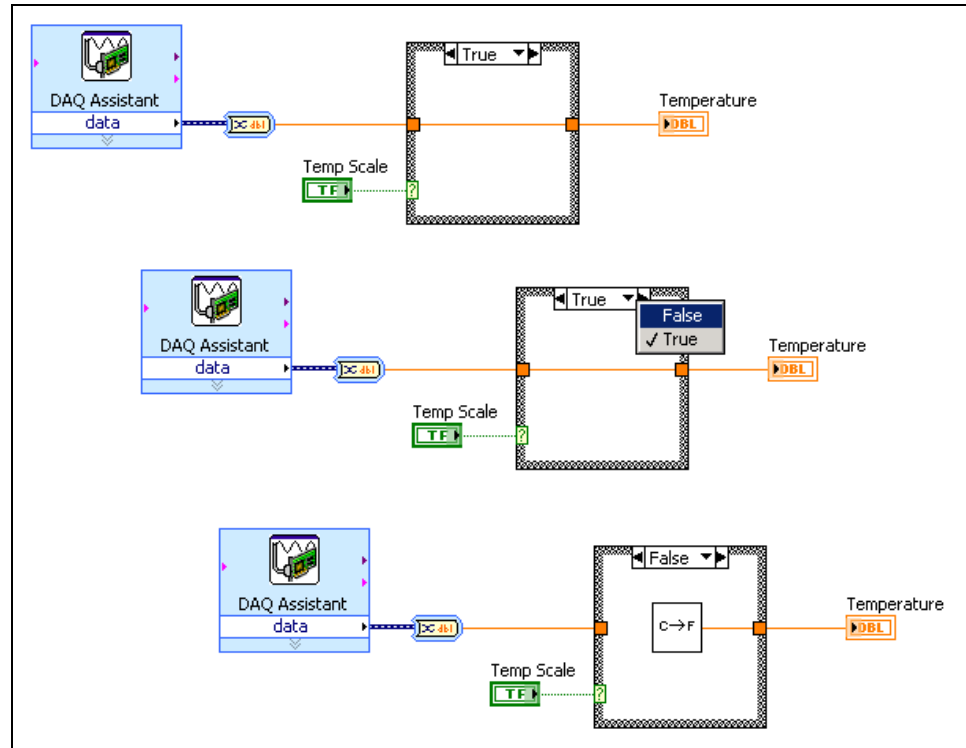


Figure 3-36. Changing the case view of a Case Structure

If you enter a selector value that is not the same type as the object wired to the selector terminal, the value appears red to indicate that you must delete or edit the value before the structure can execute, and the VI will not run. Also, because of the possible round-off error inherent in floating-point arithmetic, you cannot use floating-point numbers as case selector values. If you wire a floating-point value to the case, LabVIEW rounds the value to the nearest even integer. If you type a floating-point value in the case selector label, the value appears red to indicate that you must delete or edit the value before the structure can execute.

Input and Output Tunnels

You can create multiple input and output tunnels for a Case structure. Inputs are available to all cases, but cases do not need to use each input. However, you must define each output tunnel for each case.

Consider the following example: A Case structure on the block diagram has an output tunnel, but in at least one of the cases, there is no output value wired to the tunnel. If you run this case, LabVIEW does not know what value to place in the output. LabVIEW indicates this error by leaving the center of the tunnel white. The unwired case might not be the case that is currently visible on the block diagram.

To correct this error, move to the case(s) that contain(s) the unwired output tunnel and wire an output to the tunnel. You also can right-click the output tunnel and select **Use Default If Unwired** from the shortcut menu to use the default value for the tunnel data type for all unwired tunnels. When the output is properly wired in all cases, the output tunnel is a solid color.

Avoid using the **Use Default If Unwired** option. Using this option does not document the block diagram well, and can confuse other programmers using your code. The Use Default If Unwired option also makes debugging your code difficult. If you choose to use this option, be aware that the default value used is the default value for the data type that is wired to the tunnel. For example, if the tunnel is a Boolean data type, the default value is FALSE. Use the Table 3-3 for assistance.

Table 3-3. Data Type Default Values

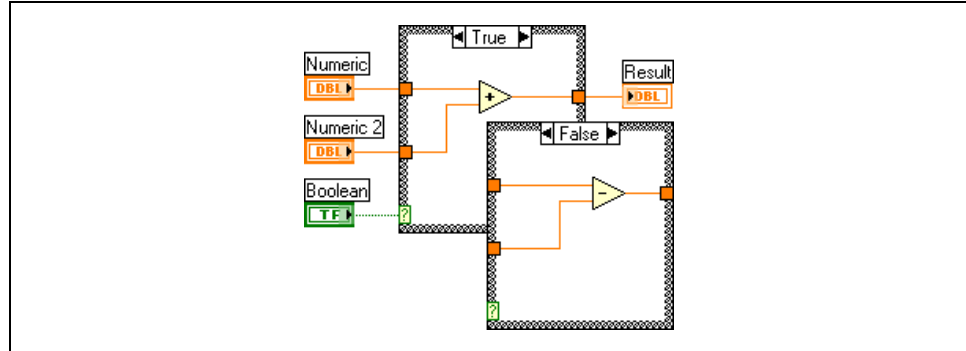
Data Type	Default Value
Numeric	0
Boolean	FALSE
String	empty

Examples

In the following examples, the numeric values pass through tunnels to the Case structure and are either added or subtracted, depending on the value wired to the selector terminal.

Boolean Case Structure

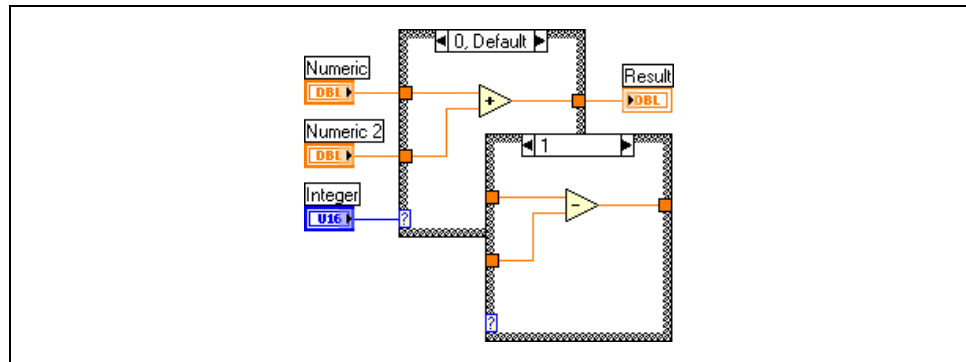
The following example is a Boolean Case structure. The cases overlap each other to simplify the illustration.



If the Boolean control wired to the selector terminal is True, the VI adds the numeric values. Otherwise, the VI subtracts the numeric values.

Integer Case Structure

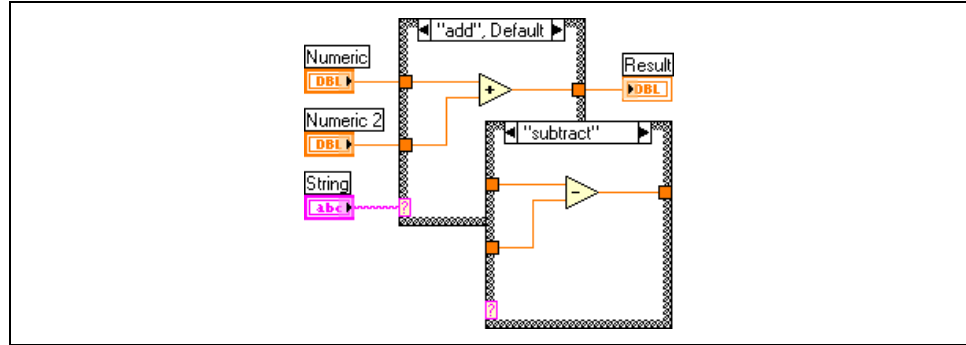
The following example is an integer Case structure.



Integer is a text ring control located on the **Controls»Text Controls** palette that associates numeric values with text items. If the text ring control wired to the selector terminal is 0 (add), the VI adds the numeric values. If the value is 1 (subtract), the VI subtracts the numeric values. If the text ring control is any other value than 0 (add) or 1 (subtract), the VI adds the numeric values, because that is the default case.

String Case Structure

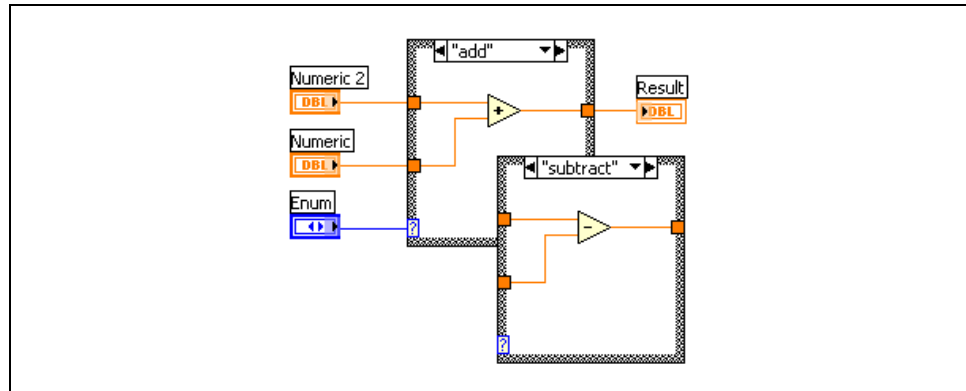
The following example is a string Case structure.



If **String** is add, the VI adds the numeric values. If **String** is subtract, the VI subtracts the numeric values.

Enumerated Case Structure

The following example is an enumerated Case structure.



An enumerated type control gives users a list of items from which to select. The data type of an enumerated type control includes information about the numeric values and string labels in the control. When you wire an enumerated type control to the selector terminal of a Case structure, the case selector displays a case for each item in the enumerated type control. The Case structure executes the appropriate case subdiagram based on the current item in the enumerated type control. In the previous block diagram, if **Enum** is add, the VI adds the numeric values. If **Enum** is subtract, the VI subtracts the numeric values.

Exercise 3-7 Project: Determine Warnings VI

Goal

Modify a VI to use a Case structure to make a software decision.

Scenario

You created a VI where a user inputs a temperature, a maximum temperature and a minimum temperature. A warning string is generated depending on the relationship of the given inputs. However, a situation could occur that causes the VI to work incorrectly. The user could enter a maximum temperature that is less than the minimum temperature. Modify the VI so that a different string is generated to alert the user to the error: "Upper Limit < Lower Limit." Set the warning Boolean to True to indicate the error.

Design

Modify the flowchart created for the original Determine Warnings VI as shown in Figure 3-37.

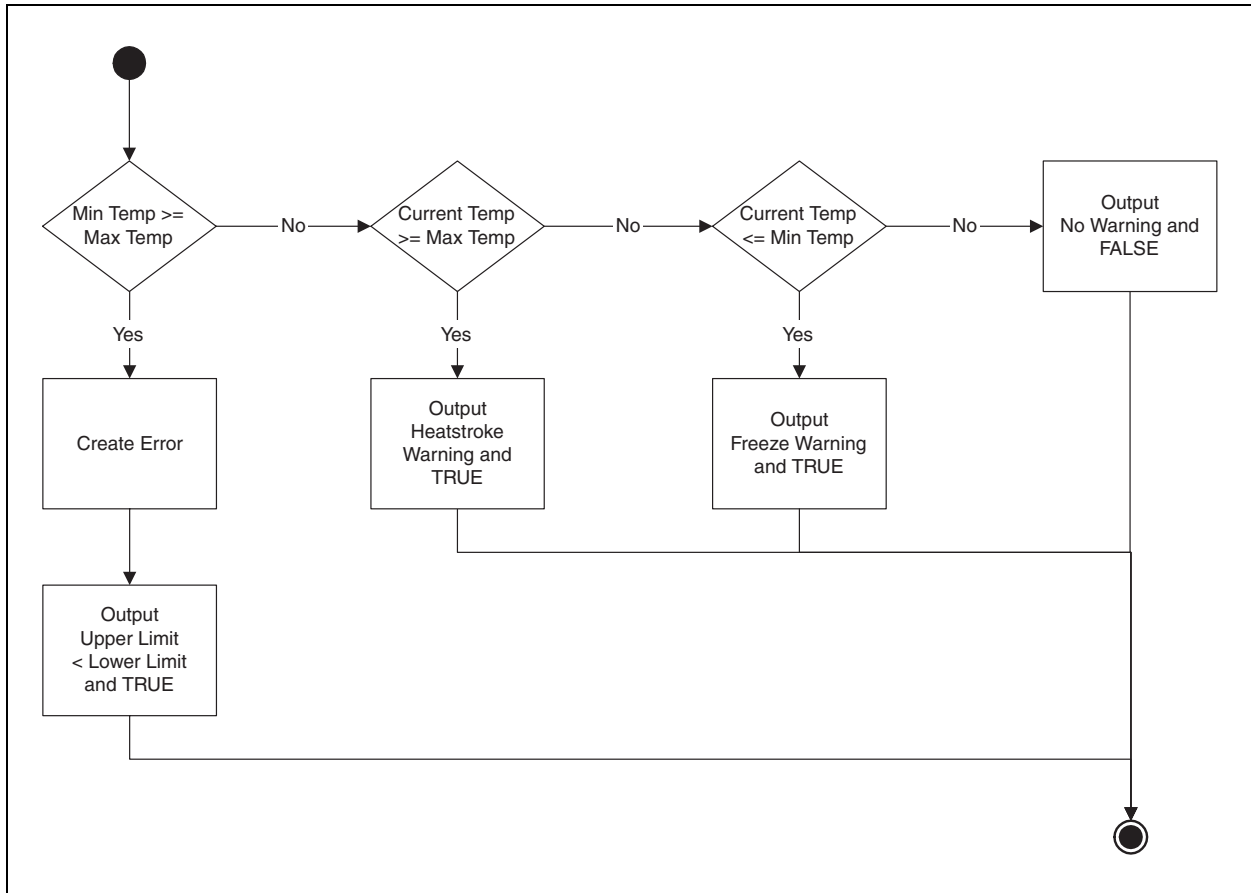


Figure 3-37. Modified Determine Warnings Flowchart

The original block diagram for the Determine Warnings VI appears in Figure 3-38. This VI must have a Case structure added to execute the code if the maximum temperature is greater than or equal to the minimum temperature. Otherwise, the code will not execute. Instead, a new string is generated and the Warning? Boolean indicator is set to True.

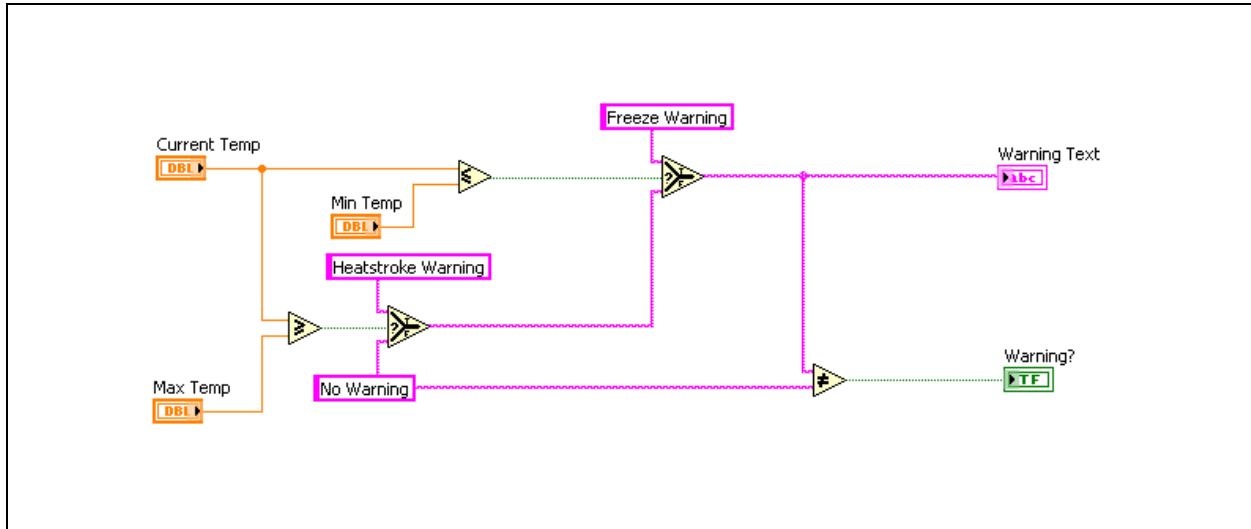


Figure 3-38. Determine Warnings VI Block Diagram

Implementation

Follow the instructions given below to modify the block diagram similar to that shown in Figure 3-39. This VI is part of the temperature weather station project.

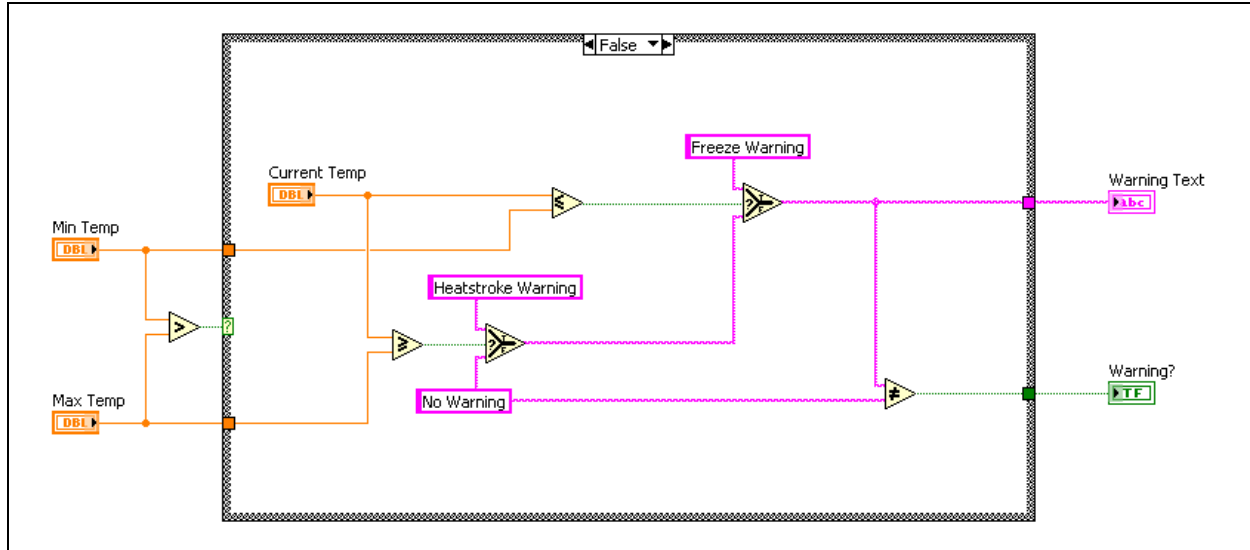


Figure 3-39. Determine Warnings VI Block Diagram

1. Open the Determine Warnings VI.
 - Select **File>Open Project**.
 - Open the `Weather Station.lvproj` in the `C:\Exercises\LabVIEW_Basics_I\Course Project` directory.
 - Double-click the **Determine Warnings.vi** in the **Project Explorer** window to open the VI.
2. Open the block diagram.
3. Create space on the block diagram to add the case structure.

The Max Temp and Min Temp controls and the Warning Text and Warning? indicators should be outside of the new Case structure, because both cases of the case structure use these indicators and controls.

 - Select the Min Temp and Max Temp control terminals.



Tip To select more than one item press the <Shift> key while you select the items.

- While the terminals are still selected, use the left arrow key on the keyboard to move the controls further to the left of the block diagram.
 - Select the **Warning Text** and **Warning?** indicator terminals.
 - Align the terminals by selecting **Align Objects»Left Edges**.
 - While the terminals are still selected, use the right arrow key on the keyboard to move the controls further to the right of the block diagram.
4. Compare Min Temp and Max Temp.



- Place the Greater? function on the block diagram.
 - Wire the **Min Temp** output to the **x** input on the Greater? function.
 - Wire the **Max Temp** output to the **y** input on the Greater? function.
 - Place a Case structure around the block diagram code, except for the excluded terminals.
 - Wire the output of the Greater? function to the case selector of the Case structure.
5. If the Min Temp is less than the Max Temp, execute the code that determines the warning string and boolean.
- While the True case is visible, right-click the border of the Case structure, and select **Make This Case False** from the shortcut menu. When you create a Case structure around existing code, the code is automatically placed in the True case.

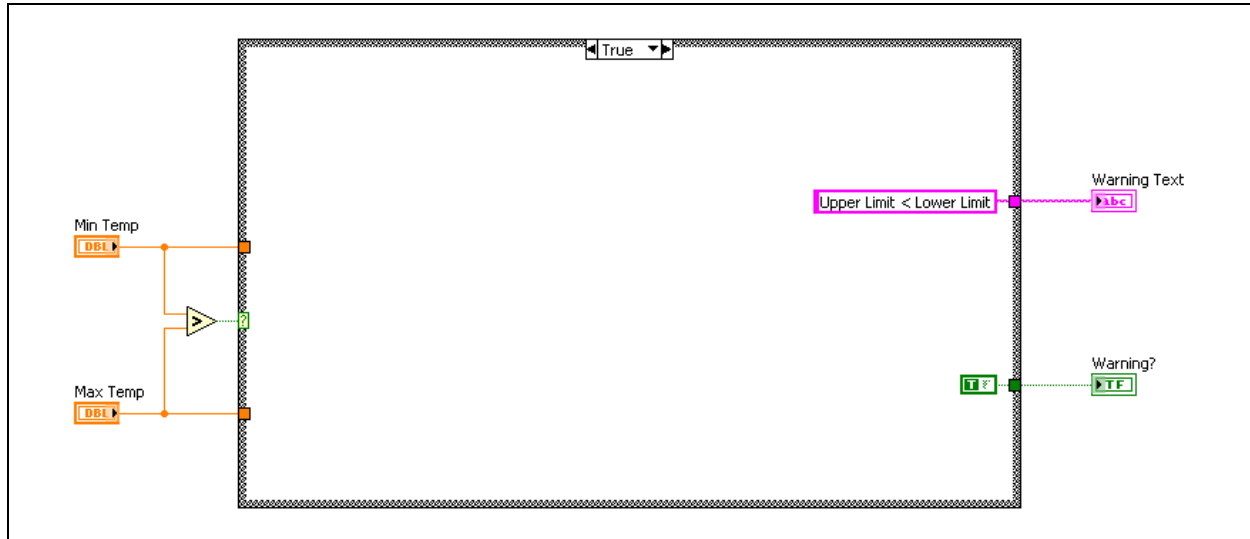


Figure 3-40. Determine Warnings VI Block Diagram

6. If the Min Temp is greater than the Max Temp, create a custom string for the Warning Text indicator and set the Warning? indicator to True, as shown in Figure 3-40.
 - Select the **True** case.
 - Right-click the string output tunnel.
 - Select **Create»Constant**.
 - Enter Upper Limit < Lower Limit in the constant.
 - Right-click the Boolean output tunnel.
 - Select **Create»Constant**.
 - Use the Operating tool to change the constant to a True constant.
7. Save the VI.

Testing

1. Switch to the front panel of the VI.
2. Resize the **Warning Text** indicator to a length to accommodate the new string.
3. Test the VI by entering values from Table 3-4 for **Current Temp**, **Max Temp**, and **Min Temp**, and running for each set.

Table 3-4 shows the expected Warning Text and Warning? Boolean value for each set of inputs.

Table 3-4. Testing Values for Determine Warnings.vi

Current Temp	Max Temp	Min Temp	Warning Text	Warning?
30	30	10	Heatstroke Warning	True
25	30	10	No Warning	False
10	30	10	Freeze Warning	True
25	20	30	Upper limit < Lower Limit	True

4. Save and close the VI.
5. Save and close the project.

End of Exercise 3-7

J. Formula Nodes

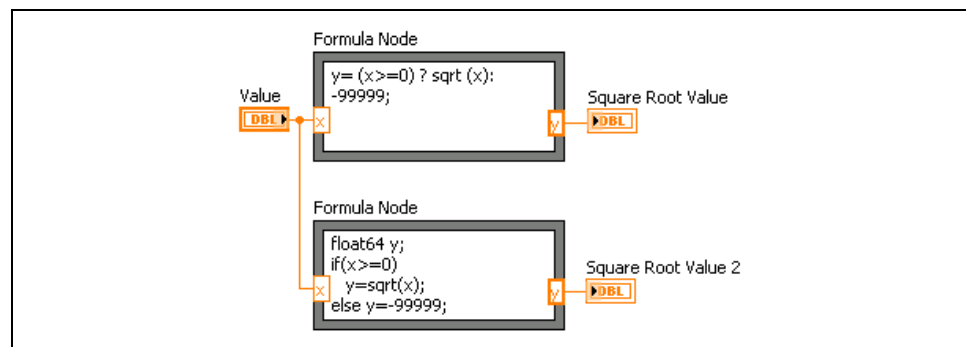
When you want to use a complicated equation in LabVIEW, you do not have to wire together various arithmetic functions on the block diagram. You can develop equations in a familiar, mathematical environment and then integrate the equations into an application.

The Formula Node is a convenient text-based node you can use to perform mathematical operations on the block diagram. You do not have to access any external code or applications, and you do not have to wire low-level arithmetic functions to create equations. In addition to text-based equation expressions, the Formula Node can accept text-based versions of if statements, while loops, for loops, and do loops, which are familiar to C programmers. These programming elements are similar to what you find in C programming but are not identical.

Formula Nodes are useful for equations that have many variables or are otherwise complicated and for using existing text-based code. You can copy and paste the existing text-based code into a Formula Node rather than recreating it graphically.

Create the input and output terminals of the Formula Node by right-clicking the border of the node and selecting **Add Input** or **Add Output** from the shortcut menu, then enter the variable for the input or output. Enter the equation in the structure. Each equation statement must terminate with a semicolon (;).

You also can use Formula Nodes for decision making. The following block diagram shows two different ways of using an if-then statement in a Formula Node. The two structures produce the same result.



The Formula Node can perform many different operations. Refer to the *LabVIEW Help* for more information about functions, operations, and syntax for the Formula Node.



Note The Formula Express VI uses a calculator interface to create mathematical formulas. You can use this Express VI to perform most math functions that a basic scientific calculator can compute. Refer to the *LabVIEW Help* for more information about the Formula Express VI.

Exercise 3-8 Optional: Square Root VI

Goal

Create a VI that uses a Case structure to make a software decision.

Scenario

Build a VI that calculates the square root of a number the user enters. If the number is negative, display the following message to the user:
 Error... Cannot find the square root of a negative number.

Design

Inputs and Outputs

Table 3-5. Inputs and Outputs

Type	Name	Properties
Input	Number	DBL; default value of 25
Output	Square Root Value	DBL

Flowchart

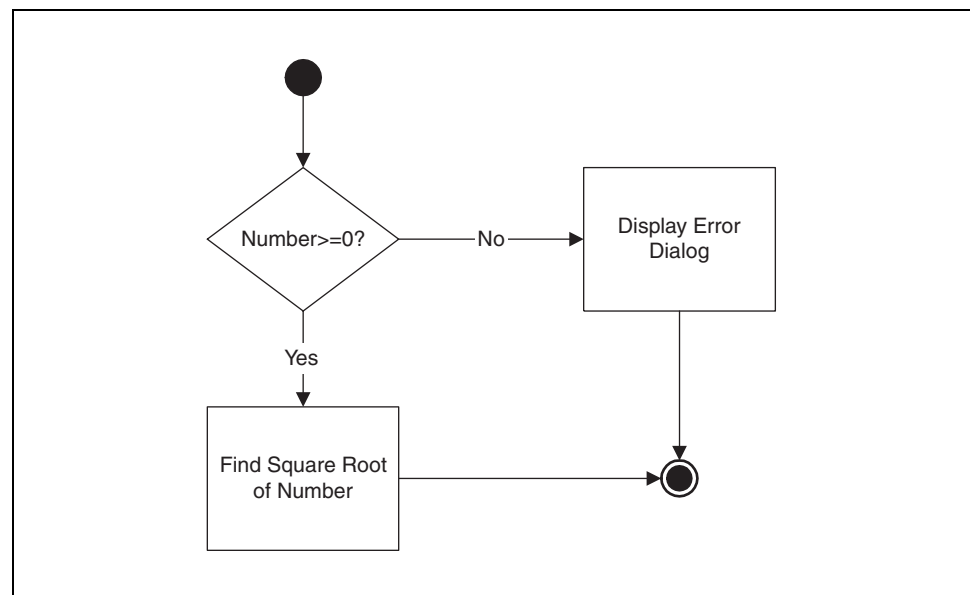


Figure 3-41. Square Root VI Flowchart

Implementation

1. Open a blank VI and build the front panel shown in Figure 3-42.



Figure 3-42. Square Root VI Front Panel

2. Place a numeric control on the front panel.
 - Name the numeric control **Number**.
 - Place a numeric indicator on the front panel.
 - Rename the numeric indicator **Square Root Value**.

Build the block diagram shown in Figure 3-43.

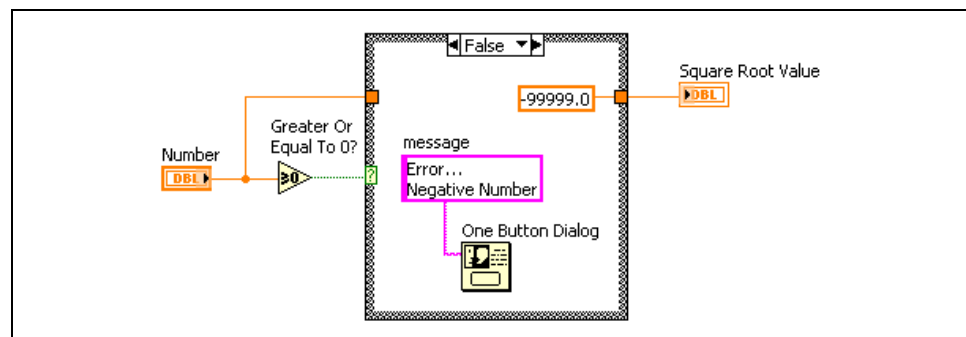


Figure 3-43. Square Root VI Block Diagram

3. Determine whether **Number** is greater than or equal to zero, because you cannot calculate the square root of a negative number.
 - Place the Greater or Equal to 0? function to the right of the **Number** terminal. This function returns True if **Number** is greater than or equal to 0.
 - Wire **Number** to the input of the Greater or Equal to 0? function.
4. If **Number** is less than 0, display a dialog box that informs the user of the error.
 - Place the Case structure on the block diagram.
 - Click the decrement or increment button to select the False case.
 - Place a numeric constant in the False case.



- Right-click the numeric constant and select **Representation»I32**.
- Enter -99999 in the numeric constant.
- Wire the **numeric** constant to the right edge of the case structure.
- Wire the new tunnel to the Square Root Value terminal.



abc

- Place the One Button Dialog function in the False case. This function displays a dialog box that contains a specified message.
- Right-click the **message** input of the One Button Dialog function and select **Create»Constant** from the shortcut menu
- Enter `Error...Negative Number` in the constant.
- Finish wiring the False case as shown in Figure 3-43.

5. If Number is greater than or equal to 0, calculate the square root of the number.



- Select the True case of the Case structure.
- Place the Square Root function in the True case. This function returns the square root of **Number**.
- Wire the function as shown in Figure 3-44.

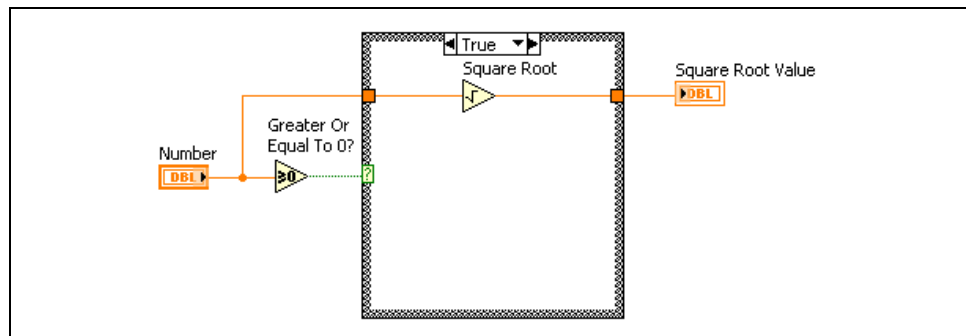


Figure 3-44. True case of Square Root VI

6. Save the VI as `Square Root.vi` in the `C:\Exercises\LabVIEW Basics I\Square Root` directory.

Testing

1. Display the front panel.
2. Enter a positive number into the **Number** control.

3. Run the VI.
4. Enter a negative number into the Number control.



Caution Do *not* run this VI continuously. Under certain circumstances, continuously running this VI could result in an endless loop.

5. Run the VI.

If **Number** is positive, the VI executes the True case and returns the square root of **Number**. If **Number** is negative, the VI executes the False case, returns **-99999**, and displays a dialog box with the message **Error...Negative Number**.

6. Close the VI.

End of Exercise 3-8

Exercise 3-9 Optional: Determine Warnings Challenge VI

Lesson Challenge Exercise

Goal

Modify an existing VI to use the Formula Node or a Case structure to make a software decision.

Scenario

In the Determine Warnings VI from Exercise 3-2, you used the Select function to pass a string based on decision. Revise this block diagram to use either a Formula Node or a Case structure (or a combination of both) to complete the same purpose.

Design

Inputs and Outputs

Table 3-6. Determine Warnings VI Inputs and Outputs

Type	Name	Properties
Numeric Control	Current Temp	Double-Precision, Floating-Point
Numeric Control	Max Temp	Double-Precision, Floating-Point
Numeric Control	Min Temp	Double-Precision, Floating-Point
String Indicator	Warning Text	Three potential values: Heatstroke Warning, No Warning, and Freeze Warning
Round LED	Warning?	

Flowchart

Figure 3-45 shows the flowchart you used in Exercise 3-2 to create the Determine Warnings VI.

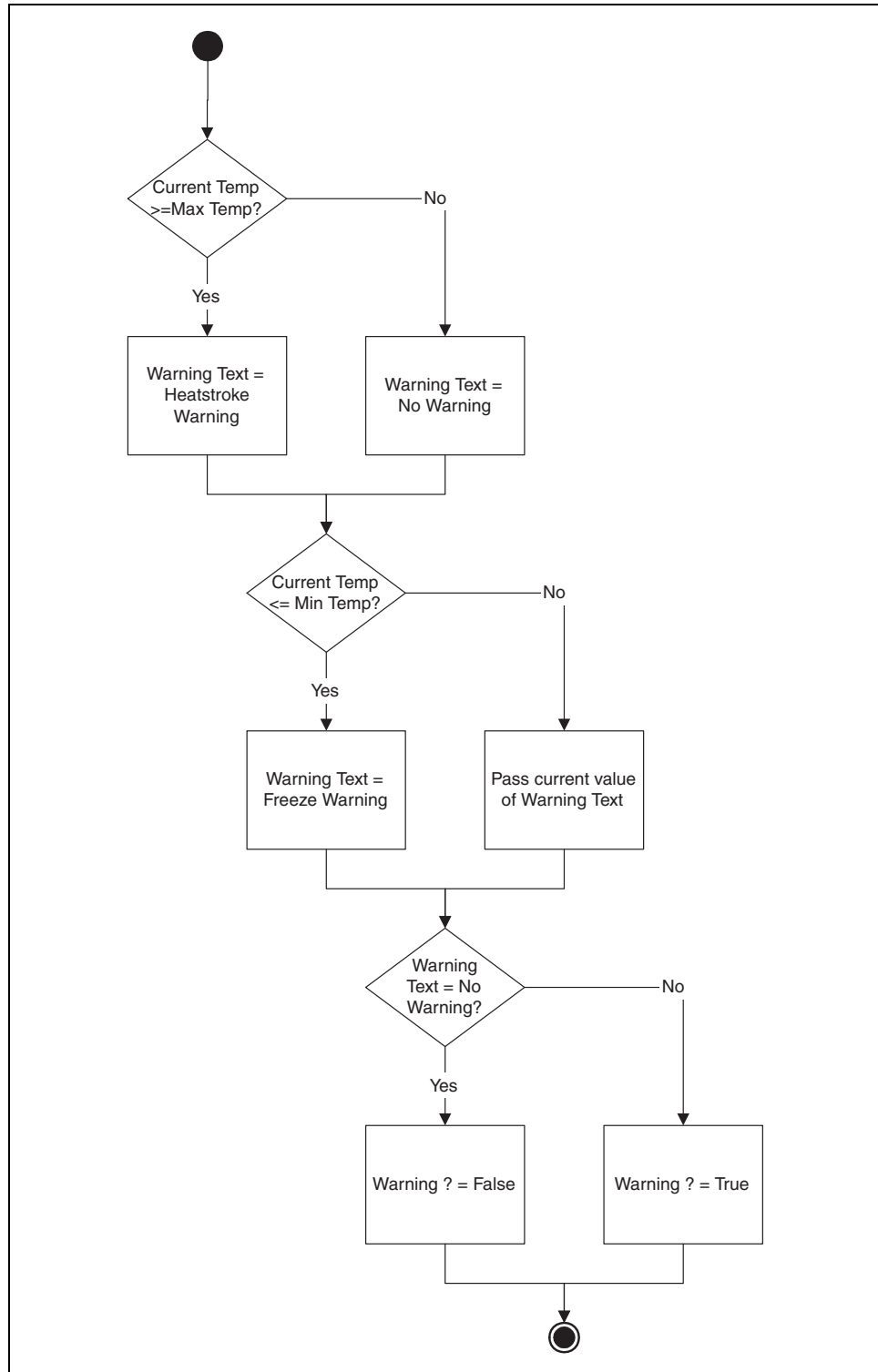


Figure 3-45. Determine Warnings VI Flowchart

Implementation

As part of the challenge, no implementation instructions are given for this exercise. The VI you should start from is located in the `C:\Exercises\LabVIEW Basics I\Determine Warnings Challenge` directory.

If you need assistance, open the solution VIs. The solutions are located in `C:\Solutions\LabVIEW Basics I\Exercise 3-9` directory.

The solution to this exercise is not used in the course project, so you do not need to add it to the `Weather Station.proj` file.

End of Exercise 3-9

Exercise 3-10 Optional: Determine More Warnings VI

Goal

Manipulate strings using string functions.

Scenario

You have a VI that determines whether a Heatstroke Warning or a Freeze Warning has occurred, based on temperature input. You must expand this VI so that it also determines whether a High Wind Warning has occurred based on a wind speed reading and a maximum wind speed setting. The warnings must be displayed as a single string. For example, if a Heatstroke Warning and a High Wind Warning has occurred, the string should read: Heatstroke and High Wind Warning.

Design

Inputs and Outputs

Table 3-7. Determine More Warnings VI Inputs and Outputs

Type	Name	Properties
Numeric Control	Current Temp	Double-Precision
Numeric Control	Max Temp	Double-Precision
Numeric Control	Min Temp	Double-Precision
Numeric Control	Current Wind Speed	Double-Precision
Numeric Control	Max Wind Speed	Double-Precision
String Indicator	Warning Text	Potential values: Heatstroke Warning, Freeze Warning, Heatstroke and High Wind Warning, Freeze and High Wind Warning, High Wind Warning and No Warning
Boolean Indicator	Warning?	Boolean

Flowchart

The flowchart shown in Figure 3-46 was used for the Determine Warnings VI. In this VI, wind data is not taken. Modify this flowchart to determine the High Wind Warning as well.

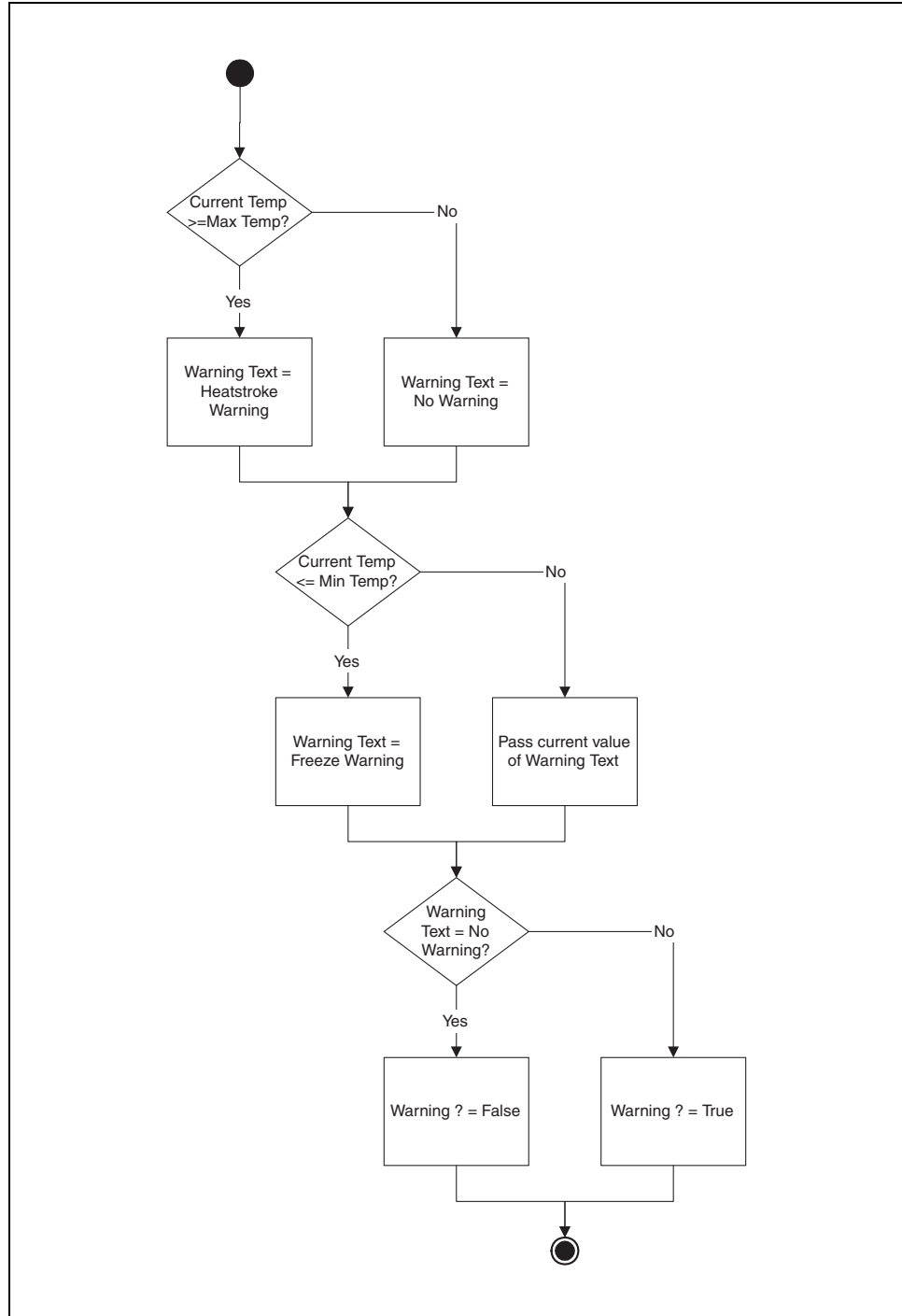


Figure 3-46. Determine Warnings VI Flow Chart

The flowchart shown in Figure 3-47 is a modified version of the flowchart, designed to determine the High Wind Warning, in addition to the warnings already determined.

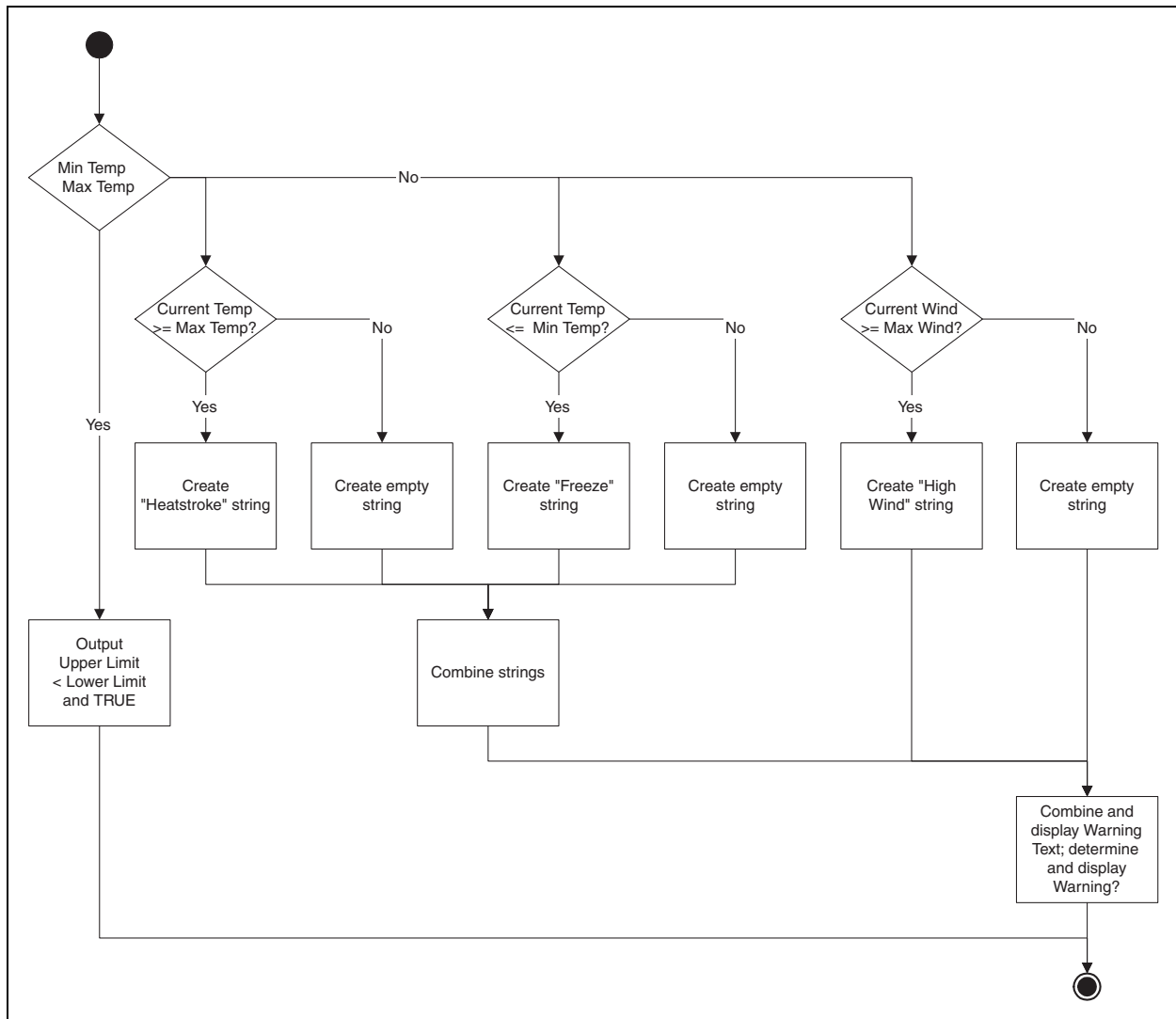


Figure 3-47. Determine More Warnings VI Flow Chart

VI Architecture

There are many ways to write this program. In this exercise, you use case structures to determine what string to pass, and Concatenate Strings functions to merge strings together.

Implementation

A portion of this VI has already been built for you. The front panel of the VI is shown in Figure 3-48. This front panel retrieves from the user the current temperature, the maximum temperature, the minimum temperature, the current wind speed and the maximum wind speed and displays to the user the warning string and the warning Boolean. This VI is not used in the Weather Station project in this course.

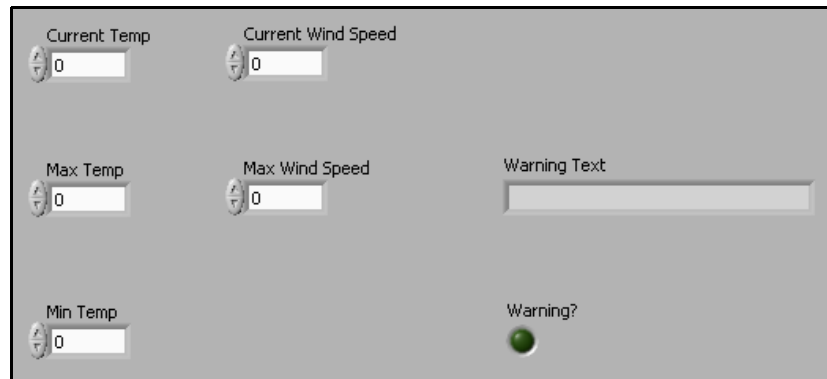


Figure 3-48. Determine More Warnings Front Panel

1. Open the `Determine More Warnings.vi` in the `C:\Exercises\LabVIEW Basics I\Determine More Warnings` directory.

Create a block diagram similar to Figure 3-49.

2. Open the block diagram.
3. Use Figures 3-49 through 3-53 to assist you in building the block diagram code.
4. You use the following block diagram objects in this exercise:



- Case structure.
- Empty String constant.
- Space constant.
- Equal function.
- Concatenate Strings function.

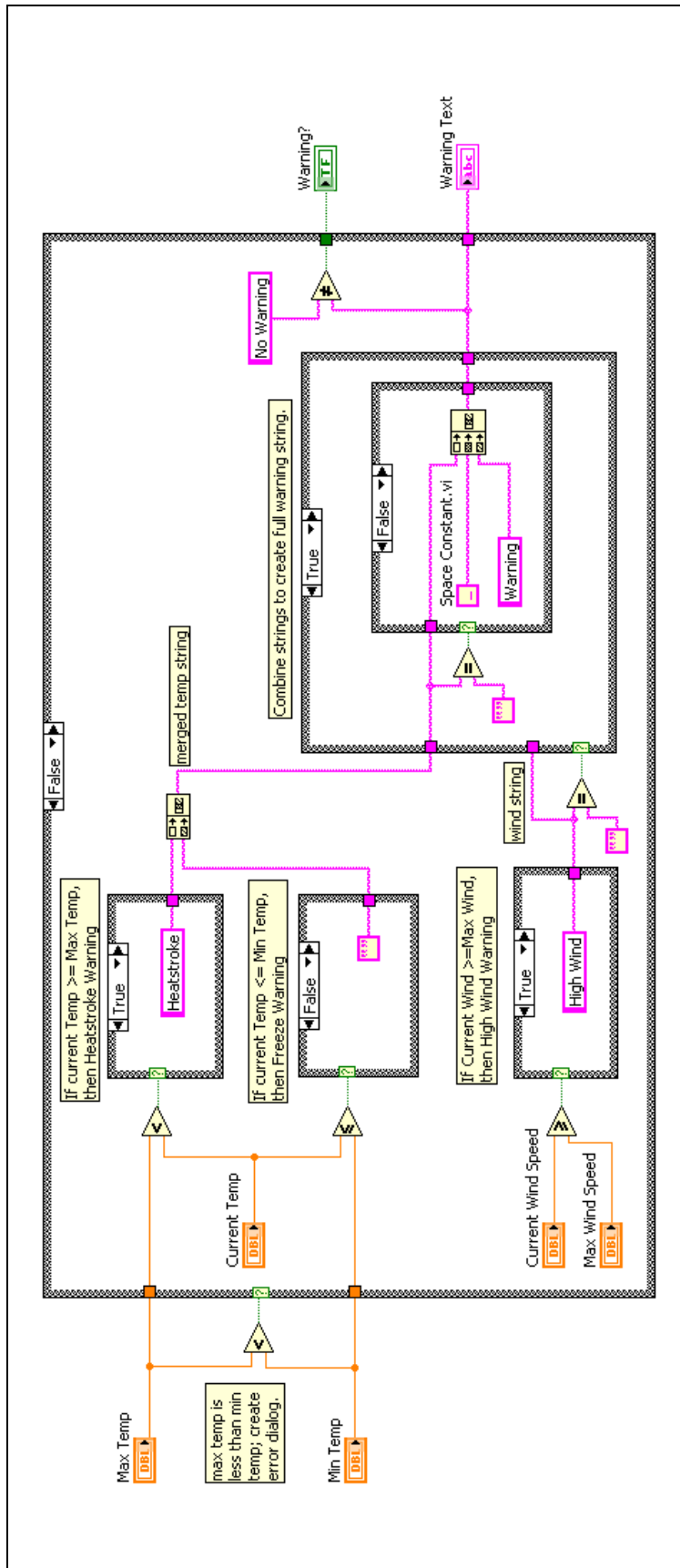


Figure 3-49. Determine More Warnings Block Diagram

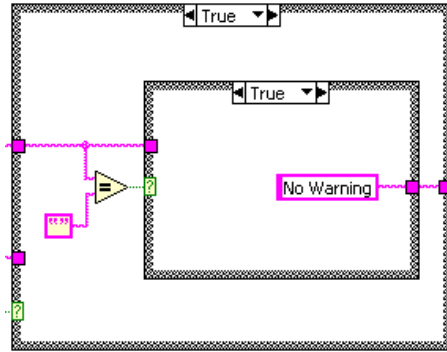


Figure 3-50. True Cases for When Temperature and Wind Warnings Are Not Generated

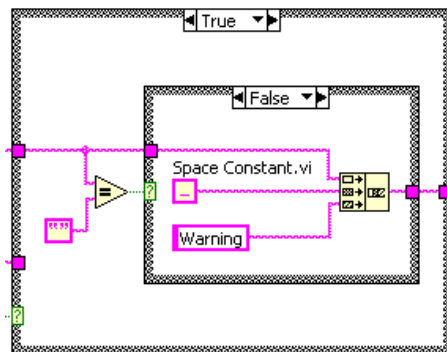


Figure 3-51. True Case for When a Temperature Warning is Generated

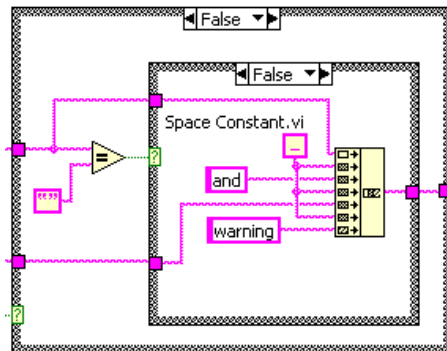


Figure 3-52. False Cases for When Wind and Temperature Warnings Are Generated

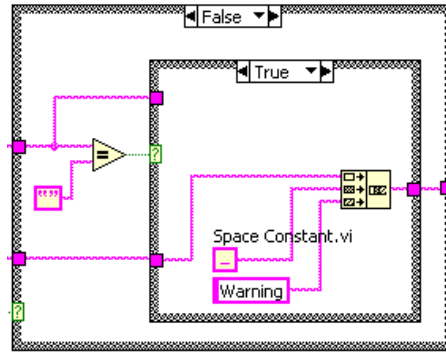


Figure 3-53. False Case for When a Wind Warning is Generated

5. Save the VI.

Testing

1. Test the following values to be sure your VI works as expected.

Name	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6
Current Temp	20	30	10	30	10	20
Max Temp	25	25	25	25	25	25
Min Temp	15	15	15	15	15	15
Current Wind Speed	25	25	25	35	35	35
Max Wind Speed	30	30	30	30	30	30
Warning Text	No Warning	Heatstroke Warning	Freeze Warning	Heatstroke and High Wind Warning	Freeze and High Wind Warning	High Wind Warning,
Warning?	False	True	True	True	True	True

2. Close and save the VI when you are finished.

End of Exercise 3-10

Self-Review: Quiz

1. Which of the following identifies the control or indicator on the block diagram?
 - a. Caption
 - b. Location
 - c. Label
 - d. Value
2. Which of the following structures must run at least one time?
 - a. While Loop
 - b. For Loop
3. Which of the following is **ONLY** available on the block diagram?
 - a. Control
 - b. Constant
 - c. Indicator
 - d. Connector Pane
4. Which of the following mechanical actions causes a Boolean in the false state to change to true when you press it and stay true until you release it and LabVIEW has read the value?
 - a. Switch Until Released
 - b. Switch When Released
 - c. Latch Until Released
 - d. Latch When Released

Self-Review: Quiz Answers

1. Which of the following identifies the control or indicator on the block diagram?
 - a. Caption
 - b. Location
 - c. **Label**
 - d. Value
2. Which of the following structures must run at least one time?
 - a. **While Loop**
 - b. For Loop
3. Which of the following is **ONLY** available on the block diagram?
 - a. Control
 - b. **Constant**
 - c. Indicator
 - d. Connector Pane
4. Which of the following mechanical actions causes a Boolean in the false state to change to true when you press it and stay true until you release it and LabVIEW has read the value?
 - a. Switch Until Released
 - b. Switch When Released
 - c. **Latch Until Released**
 - d. Latch When Released

Notes

Relating Data

Sometimes it is beneficial to group related data to one another. Arrays and clusters are used for this purpose. Arrays combine data of the same datatype into one data structure, and clusters combine data of multiple datatypes into one data structure. Type definitions are frequently used to define custom arrays and clusters. This lesson explains arrays, clusters and type definitions, and applications where using these can be beneficial.

Topics

- A. Arrays
- B. Clusters
- C. Type Definitions

A. Arrays

An array consists of elements and dimensions. Elements are the data that make up the array. A dimension is the length, height, or depth of an array. An array can have one or more dimensions and as many as $(2^{32}) - 1$ elements per dimension, memory permitting.

You can build arrays of numeric, Boolean, path, string, waveform, and cluster data types. Consider using arrays when you work with a collection of similar data and when you perform repetitive computations. Arrays are ideal for storing data you collect from waveforms or data generated in loops, where each iteration of a loop produces one element of the array.

Restrictions

You cannot create arrays of arrays. However, you can use a multidimensional array or create an array of clusters where each cluster contains one or more arrays. Also, you cannot create an array of subpanel controls, tab controls, .NET controls, ActiveX controls, charts, or multiplot XY graphs.

Refer to the *Clusters* section of this lesson for more information about clusters.

An example of a simple array is a text array that lists the nine planets of our solar system. LabVIEW represents this as a 1D array of strings with nine elements.

Array elements are ordered. An array uses an index so you can readily access any particular element. The index is zero-based, which means it is in the range 0 to $n - 1$, where n is the number of elements in the array. For example, $n = 9$ for the nine planets, so the index ranges from 0 to 8. Earth is the third planet, so it has an index of 2.

Figure 4-1 shows an example of an array of numerics. The first element shown in the array (3.00) is at index 1, and the second element (1.00) is at index 2. The element at index 0 is not shown in this image, because element 1 is selected in the index display. The element selected in the index display always refers to the element shown in the upper left corner of the element display.

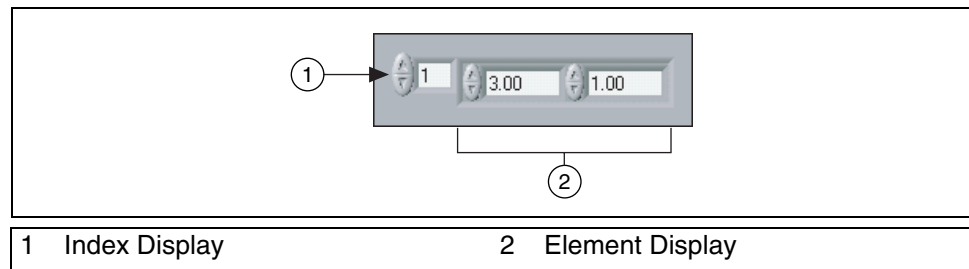


Figure 4-1. Array Control of Numerics

Creating Array Controls and Indicators

Create an array control or indicator on the front panel by placing an array shell on the front panel, as shown in the following figure, and dragging a data object or element, which can be a numeric, Boolean value, string, path, refnum, or cluster control or indicator, into the array shell.

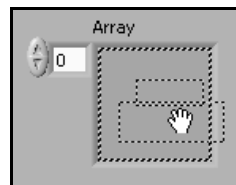


Figure 4-2. Placing a Numeric Control in an Array Shell

If you attempt to drag an invalid control or indicator such as an XY graph into the array shell, you are unable to drop the control or indicator in the array shell.

You must insert an object in the array shell before you use the array on the block diagram. Otherwise, the array terminal appears black with an empty bracket and has no data type associated with it.

Two-Dimensional Arrays

The previous examples use 1D arrays. A 2D array stores elements in a grid. It requires a column index and a row index to locate an element, both of which are zero-based. The following figure shows an 8 column by 8 row 2D array, which contains $8 \times 8 = 64$ elements.

		Column Index					
		0	1	2	3	4	5
Row Index	0						
	1						
	2						
	3						

To create a multidimensional array on the front panel, right-click the index display and select Add Dimension from the shortcut menu. You also can resize the index display until you have as many dimensions as you want.

Initializing Arrays

You can initialize an array, or leave it uninitialized. When an array is initialized, you defined the number of elements in each dimension, and the contents of each element. An uninitialized array has dimension but no elements. Figure 4-3 shows an uninitialized 2D array control. Notice that the elements are all greyed out. This indicates that the array is uninitialized.

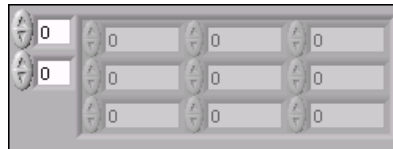


Figure 4-3. Two Dimensional Uninitialized Array

In Figure 4-4, six elements are initialized. In a two-dimensional array, once a single element in a row has been initialized, all elements in that row that are also in columns that are initialized are automatically initialized and populated with the default value for the data type. For example, in Figure 4-4, if you type 4 into the element in the first column, third row, the elements in the second and third column in the third row are automatically populated with a 0.

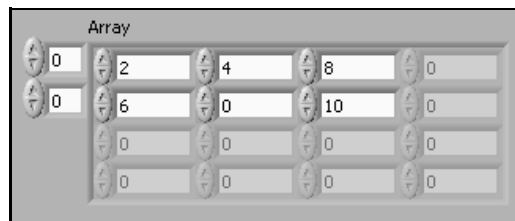


Figure 4-4. An Initialized Two Dimensional Array with six elements

Creating Array Constants

To create an array constant on the block diagram, select an array constant on the **Functions** palette, place the array shell on the block diagram, and place a string constant, numeric constant, or cluster constant in the array shell. You can use an array constant to store constant data or as a basis for comparison with another array. Array constants also are useful for passing data into a subVI.

Auto-Indexing Array Inputs



If you wire an array to or from a For Loop or While Loop, you can link each iteration of the loop to an element in that array by enabling auto-indexing. The tunnel image changes from a solid square to the image shown at left to

indicate auto-indexing. Right-click the tunnel and select **Enable Indexing** or **Disable Indexing** from the shortcut menu to toggle the state of the tunnel.

Array Inputs

If you enable auto-indexing on an array wired to a For Loop input terminal, LabVIEW sets the count terminal to the array size so you do not need to wire the count terminal. Because you can use For Loops to process arrays an element at a time, LabVIEW enables auto-indexing by default for every array you wire to a For Loop. Disable auto-indexing if you do not need to process arrays one element at a time.

In Figure 4-5, the For Loop executes a number of times equal to the number of elements in the array. Normally, if the count terminal of the For Loop is not wired, the run arrow is broken. However, in this case the run arrow is not broken.

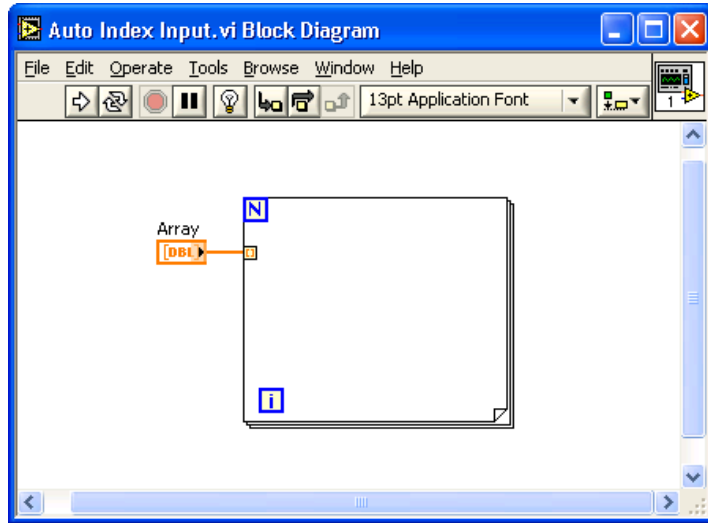


Figure 4-5. Array used to set For Loop Count

If you enable auto-indexing for more than one tunnel or if you wire the count terminal, the count becomes the smaller of the choices. For example, if two auto-indexed arrays enter the loop, with 10 and 20 elements respectively, and you wire a value of 15 to the count terminal, the loop executes 10 times, and the loop indexes only the first 10 elements of the second array.

Array Outputs

When you auto-index an array output tunnel, the output array receives a new element from every iteration of the loop. Therefore, auto-indexed output arrays are always equal in size to the number of iterations. The wire from the output tunnel to the array indicator becomes thicker as it changes to an array at the loop border, and the output tunnel contains square brackets representing an array, as shown Figure 4-6.

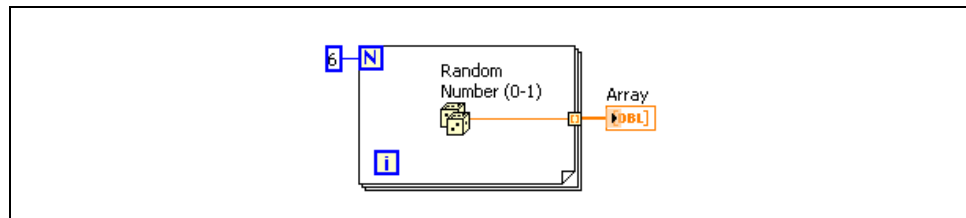


Figure 4-6. Auto-Indexed Output

Right-click the tunnel at the loop border and select Enable Indexing or Disable Indexing from the shortcut menu to enable or disable auto-indexing. Auto-indexing for While Loops is disabled by default.

For example, disable auto-indexing if you need only the last value passed out of the tunnel.

Creating Two-Dimensional Arrays

You can use two For Loops, one inside the other, to create a 2D array. The outer For Loop creates the row elements, and the inner For Loop creates the column elements, as shown in Figure 4-7.

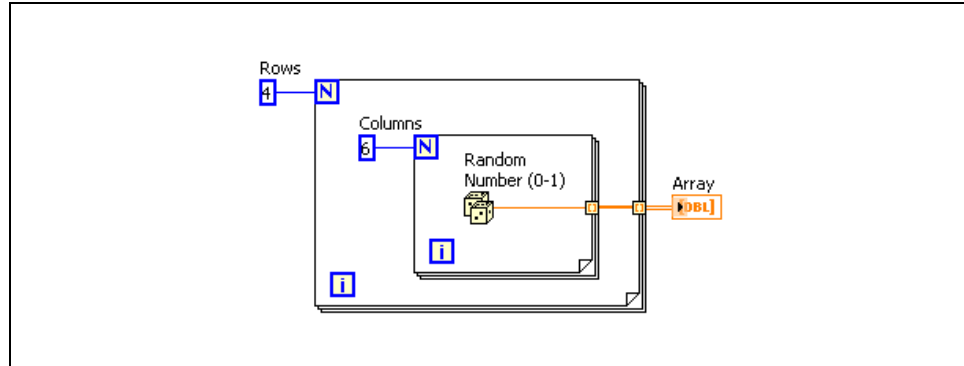


Figure 4-7. Creating a Two-Dimensional Array

Exercise 4-1 Concept: Manipulating Arrays

Goal

Manipulate arrays using various LabVIEW Functions.

Description

You are given a VI and asked to enhance it for a variety of purposes. For each part of this exercise, begin with the `Array Investigation.vi` located in the `C:\Exercises\LabVIEW Basics I\Manipulating Arrays` directory. The front panel of this VI is shown in Figure 4-8.

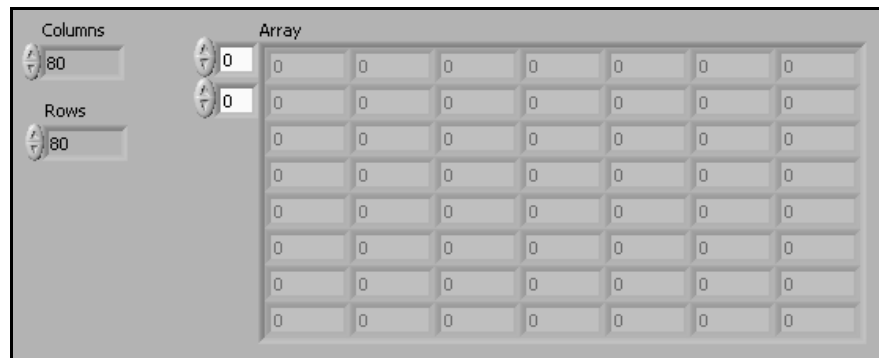


Figure 4-8. Array Investigation VI Front Panel

The block diagram of this VI is shown in Figure 4-9.

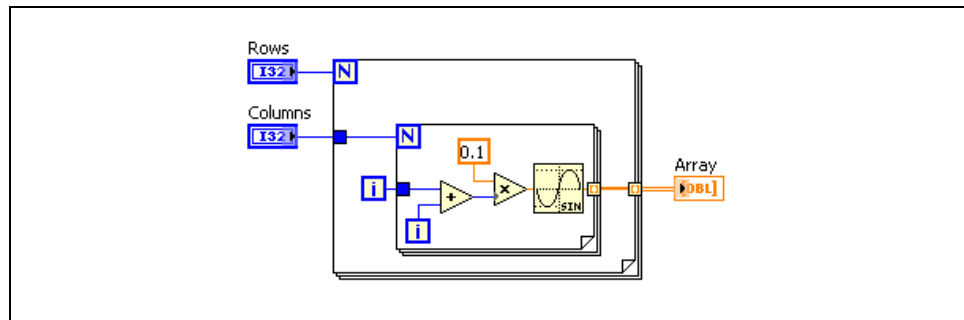


Figure 4-9. Array Investigation VI Block Diagram

In this exercise, you are given the scenario for each part first. If you want detailed implementation instructions, they are given for each part at the end of this exercise.

Part 1: Iterate, Modify, and Chart Array

Modify the Array Investigation VI so that after the array is created, the array is indexed into For Loops where you multiply each element of the array by 100 and coerce each element to the nearest whole number. The resulting two dimensional array is graphed to an Intensity Chart.

Part 2: Simplified Iterate, Modify, and Chart Array

Modify the Array Investigation VI or the solution from Part 1 to accomplish the same goals without using the nested For Loops.

Part 3: Create Subset Arrays

Modify the Array Investigation VI so that the VI creates a new array that contains the contents of the third row, and another new array that contains the contents of the second column.

Part 1: Implementation

Modify Array Investigation VI so that after the array is created, the array is indexed into For Loops where each element of the array is multiplied by 100, and coerced to the nearest whole number. The resulting two dimensional array is graphed to an Intensity Chart.

1. Open `Array Investigation.vi` located in the `C:\Exercises\LabVIEW Basics I\Manipulating Arrays` directory.
2. Save the VI as `Array Investigation Part 1.vi`.
3. Place an intensity chart on the front panel of the VI, as shown in Figure 4-10.

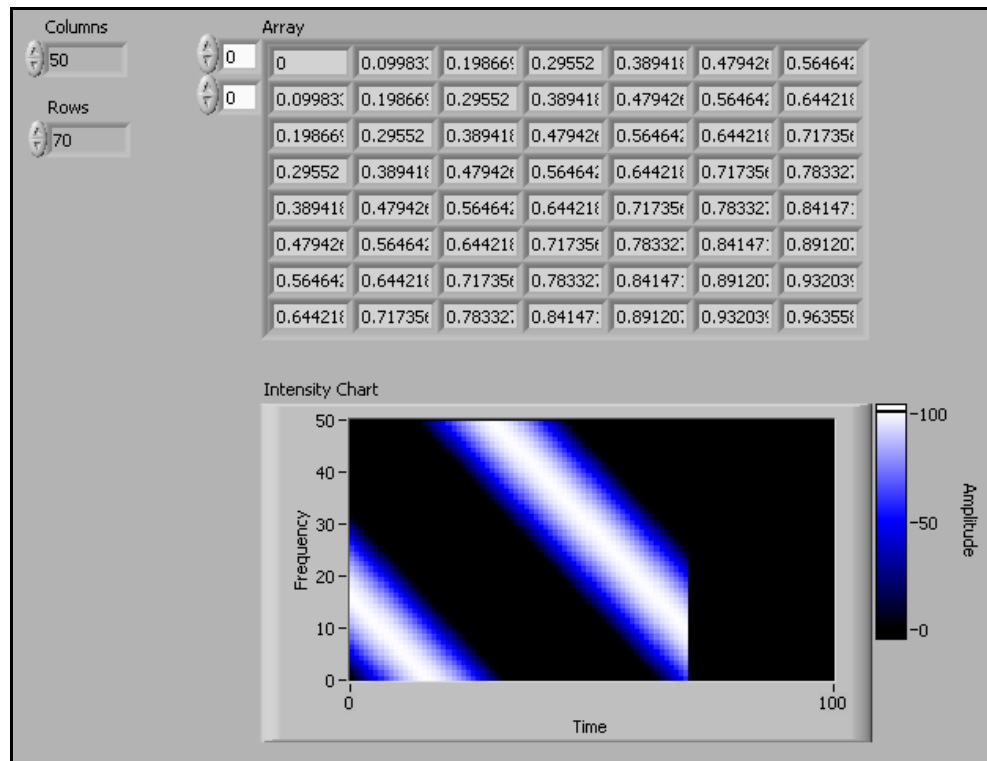


Figure 4-10. Array Investigation Part 1 VI Front Panel

4. Open the block diagram of the VI.

In the following steps, you create a block diagram similar to that in Figure 4-11.

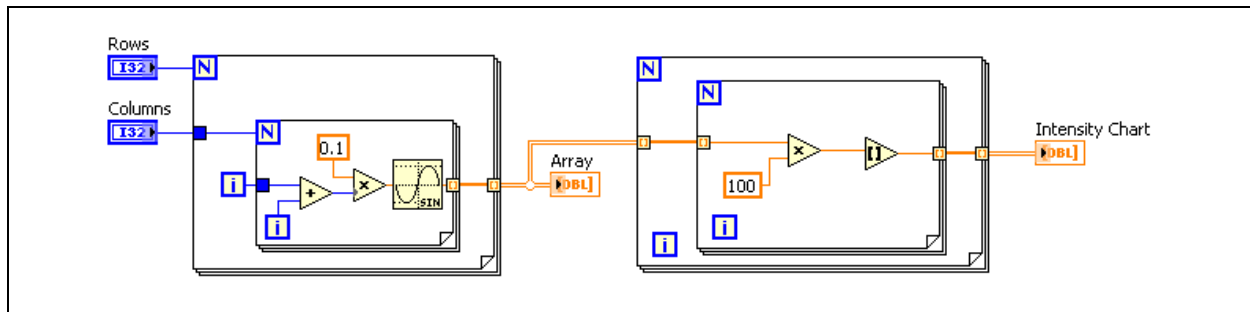


Figure 4-11. Array Investigation Part 1 VI Block Diagram

5. Iterate Array.



- Place a For Loop to the right of the existing code.
- Place a second For Loop inside the first For Loop.
- Wire the Array indicator terminal to the interior For Loop border. This creates an auto-indexed input tunnel on both For Loops.

6. Multiply each element of the array by 100.



- Place a Multiply function in the interior For Loop.
- Wire the indexed input tunnel to the x terminal of the Multiply function.
- Right-click the y terminal and select **Create»Constant** from the shortcut menu.
- Enter 100 in the constant.

7. Round each element to the nearest whole number.



- Place a Round To Nearest function to the right of the Multiple function.
- Wire the output of the Multiply function to the input of the Round To Nearest function.

8. Create a 2D array on the output of the For Loops to recreate the modified array.

- Wire the output of the Round To Nearest function to the outer For Loop. This creates an auto-indexed output tunnel on both For Loops.

9. Wire the output array to the Intensity Chart.
10. Switch to the front panel.
11. Save the VI.
12. Enter values for Rows and Columns.
13. Run the VI.

Part 2: Implementation

Modify Part 1 to accomplish the same goals without using the nested For Loops.

1. Open Array Investigation Part 1.vi if it is not still open from the last part.
2. Save the VI as Array Investigation Part 2.VI.
3. Open the block diagram.
4. Right-click the interior For Loop, containing the Multiply and Round to Nearest function, and select **Remove For Loop**.
5. Right-click the remaining For Loop and select **Remove For Loop** from the shortcut menu. Your block diagram should resemble the one shown in Figure 4-12.

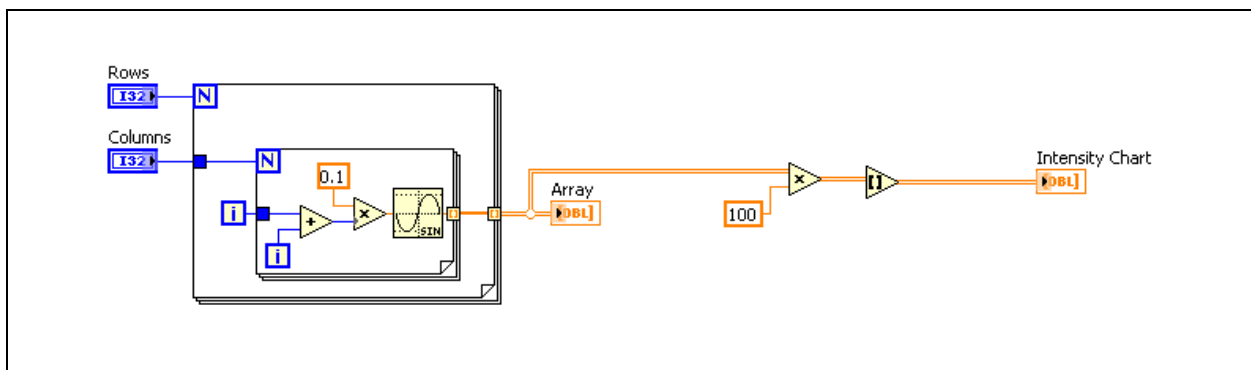


Figure 4-12. Array Investigation Part 2 VI Block Diagram

6. Save the VI.
7. Switch to the front panel.
8. Enter values for Rows and Columns.
9. Run the VI.

Notice that the VI behaves the same as in Part 1. This is because mathematical functions are polymorphic. For example, because the **x** input of the Multiply function is a two-dimensional array, and the **y** input is a scalar, the Multiply function multiplies each element in the array by the scalar, and outputs an array of the same dimension as the **x** input.

Part 3: Implementation

Modify Array Investigation VI so that the VI creates a new array that contains the contents of the third row, and another new array that contains the contents of the second column.

1. Open `Array Investigation.vi` located in the `C:\Exercises\LabVIEW Basics I\Manipulating Arrays` directory.
2. Save the VI as `Array Investigation Part 3.VI`.
3. Open the block diagram of the VI.

In the following steps, you build a block diagram similar to that shown in Figure 4-13.

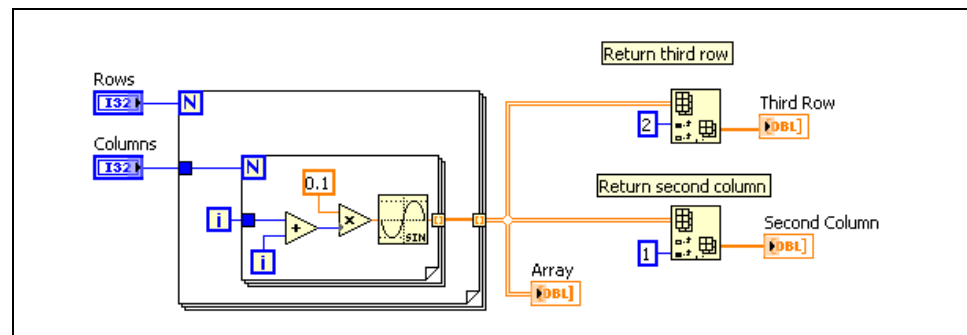


Figure 4-13. Array Investigation Part 3 VI Block Diagram

4. Retrieve the third row of data from Array using the Index Array function.



- Place the Index Array function on the block diagram.
- Wire Array to the **array** terminal of Index Array.



Tip The Index Array function accepts an n-dimension array. After you wire the input array to the Index Array function, the input and output terminal names change to match the dimension of the array wired. Therefore, wire the input array to the Index Array function before wiring any other terminals.

- Right-click the **index(row)** terminal of Index Array.

- Select **Create»Constant** from the shortcut menu.
 - Enter 2 in the constant to retrieve the third row because the index begins at zero.
 - Right-click the **subarray** output of Index Array.
 - Select **Create»Indicator** from the shortcut menu.
 - Rename the indicator to `Third Row`.
5. Retrieve the second column of data from Array using the Index Array function.



- Place another Index Array function on the block diagram.
 - Wire **Array** to the **array** terminal of Index Array.
 - Right-click the **disable index(col)** terminal of Index Array.
 - Select **Create»Constant**.
 - Enter 1 in the constant to retrieve the second column because the index begins at zero.
 - Right-click the **subarray** output of Index Array.
 - Select **Create»Indicator**.
 - Rename the indicator to `Second Column`.
6. Save the VI.
7. Switch to the front panel.
8. Enter values for Rows and Columns.
9. Run the VI.

End of Exercise 4-1

B. Clusters

Clusters group data elements of mixed types. An example of a cluster is the LabVIEW error cluster, which combines a Boolean value, a numeric value, and a string. A cluster is similar to a record or a struct in text-based programming languages.

Refer to the *Error Checking and Error Handling* topic of the *LabVIEW Help* for more information about using error clusters.

Bundling several data elements into clusters eliminates wire clutter on the block diagram and reduces the number of connector pane terminals that subVIs need. The connector pane has, at most, 28 terminals. If your front panel contains more than 28 controls and indicators that you want to pass to another VI, group some of them into a cluster and assign the cluster to a terminal on the connector pane.

Most clusters on the block diagram have a pink wire pattern and data type terminal. Clusters of numeric values, sometimes referred to as points, have a brown wire pattern and data type terminal. You can wire brown numeric clusters to Numeric functions, such as Add or Square Root, to perform the same operation simultaneously on all elements of the cluster.

Clusters differ from arrays in that they are a fixed size. Like an array, a cluster is either a control or an indicator. A cluster cannot contain a mixture of controls and indicators.

Creating Cluster Controls and Indicators

Create a cluster control or indicator on the front panel by placing a cluster shell on the front panel, as shown in the following figure, and dragging a data object or element, which can be a numeric, Boolean values, string, path, refnum, array, or cluster control or indicator, into the cluster shell. Resize the cluster shell by dragging the cursor while you place the cluster shell on the front panel.

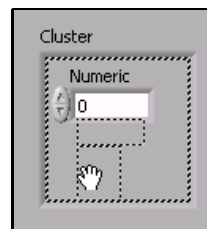


Figure 4-14. Creation of a Cluster Control

Figure 4-15 is an example of a cluster containing three controls: a string, a Boolean switch, and a numeric. A cluster is either a control or an indicator; it cannot contain a mixture of controls and indicators.

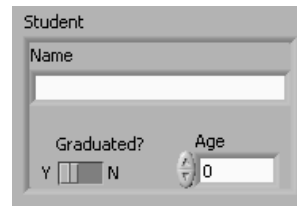


Figure 4-15. Cluster Control Example

Creating Cluster Constants

To create a cluster constant on the block diagram complete the following steps:

1. Select a cluster constant on the **Functions** palette.
2. Place the cluster shell on the block diagram.
3. Place a string constant, numeric constant, or cluster constant in the cluster shell.

You can use a cluster constant to store constant data or as a basis for comparison with another cluster.

If you have a cluster control or indicator on the front panel and you want to create a cluster constant containing the same elements on the block diagram, you can either drag that cluster from the front panel to the block diagram or right-click the cluster on the front panel and select **Create»Constant** from the shortcut menu.

Cluster Order

Cluster elements have a logical order unrelated to their position in the shell. The first object you place in the cluster is element 0, the second is element 1, and so on. If you delete an element, the order adjusts automatically. The cluster order determines the order in which the elements appear as terminals on the Bundle and Unbundle functions on the block diagram. You can view and modify the cluster order by right-clicking the cluster border and selecting Reorder Controls In Cluster from the shortcut menu. The toolbar and cluster change, as shown in Figure 4-16.

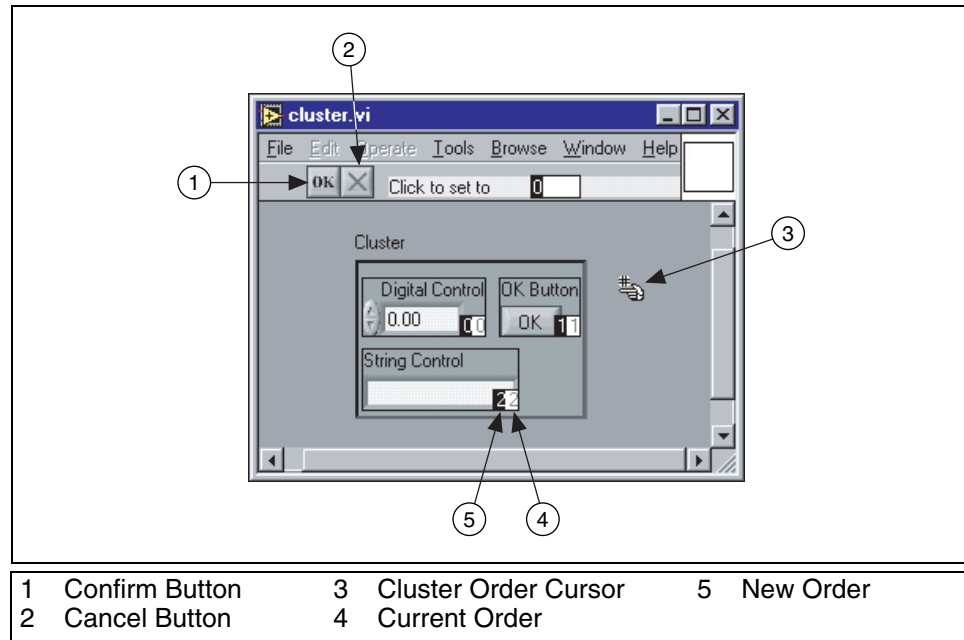


Figure 4-16. Reordering a Cluster

The white box on each element shows its current place in the cluster order. The black box shows the new place in the order for an element. To set the order of a cluster element, enter the new order number in the **Click to set to** text box and click the element. The cluster order of the element changes, and the cluster order of other elements adjusts. Save the changes by clicking the **Confirm** button on the toolbar. Revert to the original order by clicking the **Cancel** button.

To wire clusters to each other, both clusters must have the same number of elements. Corresponding elements, determined by the cluster order, must have compatible data types. For example, if a double-precision floating-point numeric value in one cluster corresponds in cluster order to a string in the another cluster, the wire on the block diagram appears broken and the VI does not run. If the numeric values are different representations, LabVIEW coerces them to the same representation.

Using Cluster Functions

Use the Cluster functions to create and manipulate clusters. For example, you can perform tasks similar to the following:

- Extract individual data elements from a cluster.
- Add individual data elements to a cluster.
- Break a cluster out into its individual data elements.

Use the **Bundle** function to assemble a cluster, the **Bundle** and **Bundle by Name** functions to modify a cluster, and the **Unbundle** and **Unbundle by Name** functions to disassemble clusters.

You also can create the Bundle, Bundle by Name, Unbundle, and Unbundle by Name functions by right-clicking a cluster terminal on the block diagram and selecting **Cluster Palette** from the shortcut menu. The Bundle and Unbundle functions automatically contain the correct number of terminals. The Bundle by Name and Unbundle by Name functions appear with the first element in the cluster. Use the Positioning tool to resize the Bundle by Name and Unbundle by Name functions to show the other elements of the cluster.

Assembling Clusters

Use the Bundle function to assemble a cluster from individual elements or to change the values of individual elements in an existing cluster without having to specify new values for all elements. Use the Positioning tool to resize the function or right-click an **element** input and select **Add Input** from the shortcut menu.

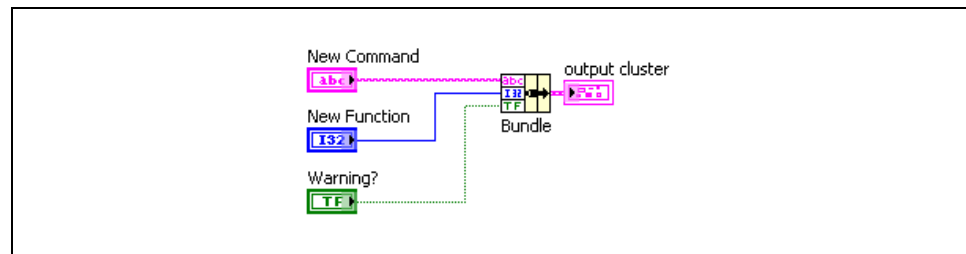


Figure 4-17. Assembling a Cluster on the Block Diagram

Modifying a Cluster

If you wire the **cluster** input, you can wire only the **elements** you want to change. For example, the Input Cluster shown in Figure 4-18 contains three controls.

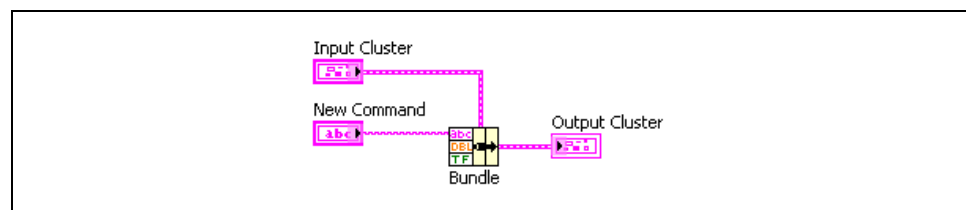


Figure 4-18. Bundle Used to Modify a Cluster

If you know the cluster order, you can use the Bundle function to change the **Command** value by wiring the elements shown in the following figure.

You can also use the Bundle by Name function to replace or access labeled elements of an existing cluster. Bundle by Name works similarly to the Bundle function, but instead of referencing cluster elements by their cluster order, it references them by their owned labels. You can access only

elements with owned labels. The number of inputs does not need to match the number of elements in **output cluster**.

Use the Operating tool to click an input terminal and select an element from the pull-down menu. You also can right-click the input and select the element from the **Select Item** shortcut menu.

In Figure 4-19, you can use the Bundle by Name function to change **Command** and **Function**.

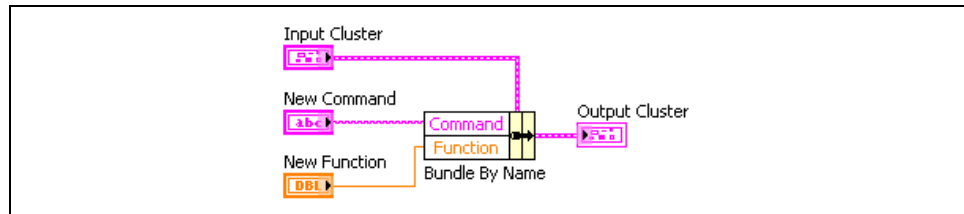


Figure 4-19. Bundle By Name Used to Modify a Cluster

Use the Bundle by Name function for data structures that might change during development. If you add a new element to the cluster or modify its order, you do not need to rewire the Bundle by Name function because the names still are valid.

Disassembling Clusters

Use the Unbundle function to split a cluster into its individual elements.

Use the Unbundle by Name function to return the cluster elements whose names you specify. The number of output terminals does not depend on the number of elements in the input cluster.

Use the Operating tool to click an output terminal and select an element from the pull-down menu. You also can right-click the output terminal and select the element from the **Select Item** shortcut menu.

For example, if you use the Unbundle function with the following cluster, it has four output terminals that correspond to the four controls in the cluster. You must know the cluster order so you can associate the correct Boolean terminal of the unbundled cluster with the corresponding switch in the cluster. In this example, the elements are ordered from top to bottom starting with element 0. If you use the Unbundle by Name function, you can have an arbitrary number of output terminals and access individual elements by name in any order.

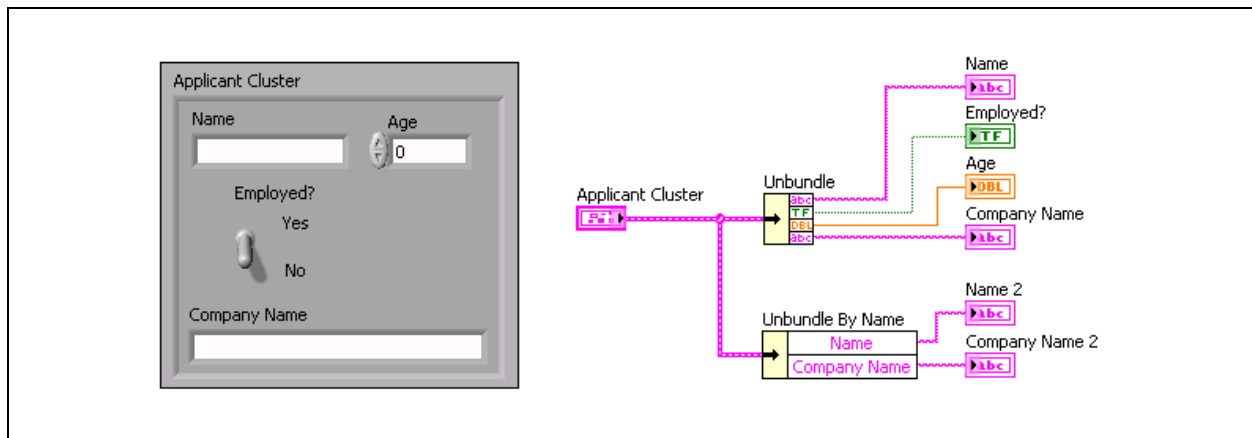


Figure 4-20. Unbundle and Unbundle By Name

Error Clusters

LabVIEW contains a custom cluster called the error cluster. LabVIEW uses error clusters to pass error information. An error cluster contains the following elements:

- **status**—Boolean value that reports TRUE if an error occurred.
- **code**—32-bit signed integer that identifies the error numerically.
- **source**—String that identifies where the error occurred.

Refer to Lesson 5, *Debugging VIs*, for more information about using error clusters.

Exercise 4-2 Concept: Clusters

Goal

Create clusters on the front panel, reorder clusters, and use the cluster functions to assemble and disassemble clusters.

Description

In this exercise, follow the instructions to experiment with clusters, cluster order and cluster functions. The VI built has no practical applications, but is useful for understanding cluster concepts.

1. Open a blank VI.
2. Save the VI as `Cluster Experiment.vi` in the `C:\Exercises\LabVIEW Basics I\Clusters` directory.

In the following steps, you build a front panel similar to Figure 4-21.

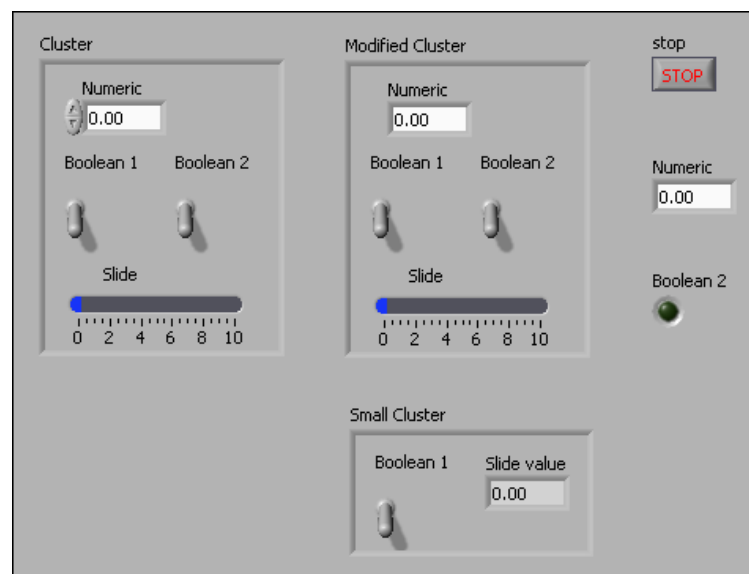


Figure 4-21. Cluster Experiment VI Front Panel

3. Place a Stop button on the front panel.
4. Place a Numeric indicator on the front panel.
5. Place a Round LED on the front panel.
6. Rename the LED `Boolean 2`.
7. Create a cluster, containing a numeric, two toggle switches, and a slide.

- Place a cluster shell on the front panel.
 - Place a numeric control in the cluster.
 - Place two vertical toggle switches in the cluster.
 - Rename the Boolean to `Boolean 1`.
 - Place a horizontal fill slide in the cluster.
8. Create **Modified Cluster**, containing the same contents as **Cluster**, but indicators instead of controls.
- Create a copy of **Cluster**.
 - Relabel the copy `Modified Cluster`.
 - Right-click the shell of **Modified Cluster**, and select **Change to Indicator** from the shortcut menu.
9. Create **Small Cluster**, containing a Boolean indicator and a numeric indicator.
- Create a copy of **Modified Cluster**.
 - Relabel the copy **Small Cluster**.
 - Delete the second toggle switch.
 - Delete the horizontal fill slide indicator.
 - Right-click **Small Cluster** and select **Autosizing»Size to Fit**.
 - Relabel the numeric indicator to `Slide value`.
 - Resize the cluster as needed.
10. Verify the cluster order of **Cluster**, **Modified Cluster**, and **Small Cluster**.
- Right-click the boundary of **Cluster** and select **Reorder Controls in Cluster** from the shortcut menu.
 - Confirm the cluster order shown in Figure 4-22.
 - Right-click the boundary of **Modified Cluster** and select **Reorder Controls in Cluster** from the shortcut menu.

- Confirm the cluster orders shown in Figure 4-22. **Modified Cluster** should have the same cluster order as **Cluster**.
- Right-click the boundary of **Small Cluster** and select **Reorder Controls in Cluster** from the shortcut menu.
- Confirm the cluster orders shown in Figure 4-22.

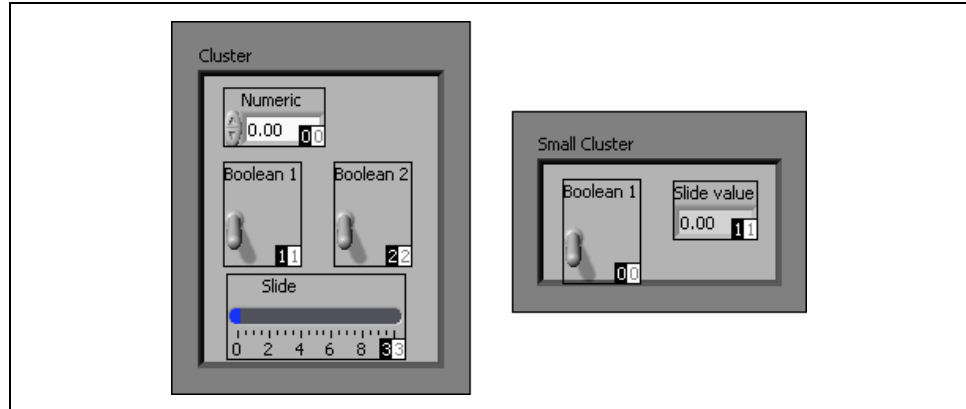


Figure 4-22. Cluster Orders

In the following steps, build the block diagram shown in Figure 4-23.

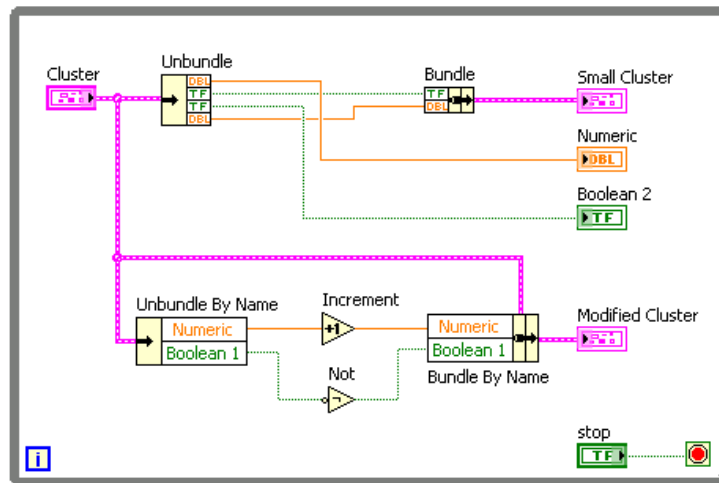


Figure 4-23. Cluster Experiment VI Block Diagram



11. Place the While Loop from the **Structures** category of the **Functions** palette on the block diagram.

12. Disassemble Cluster.



- Place the Unbundle function on the block diagram.

- Wire the Cluster to the input cluster of the Unbundle function to resize the function automatically.

13. Assemble Small Cluster.



- Place the Bundle function on the block diagram.
- Wire the Bundle function as shown in Figure 4-23.

14. Assemble Modified Cluster.



- Place the Unbundle by Name function on the block diagram.
- Wire the Cluster to the Unbundle by Name function.
- Resize the Unbundle by Name function to have two output terminals.
- Select **Numeric** in the first node, and **Boolean 1** in the second node. If a label name is not correct, use the Operating tool to select the correct item.



- Place the Increment function on the block diagram.
- Wire the Numeric output of the Unbundle By Name function to the **Increment** input. This function adds one to the value of **Numeric**.



- Place the Not function on the block diagram.
- Wire the Boolean 1 output of the Unbundle By Name function to the **x** input of the Not function. This function returns the logical opposite of the value of **Boolean**.



- Place the Bundle by Name function on the block diagram.
- Wire Cluster to the input cluster input.
- Resize this function to have two input terminals.
- Select Numeric in the first node, and Boolean 1 in the second node. If a label name is not correct, use the Operating tool to select the correct item.
- Wire the output of the Increment function to Numeric.
- Wire the output of the Not function to Boolean 1.
- Wire the output of the Bundle By Name function to the Modified Cluster indicator.

15. Complete the block diagram and wire the objects as shown in Figure 4-23.
16. Save the VI.
17. Display the front panel.
18. Run the VI.
19. Enter different values in **Cluster** and run the VI again. Notice how values entered in **Cluster** affect the **Modified Cluster** and **Small Cluster** indicators. Is this the behavior you expected?
20. Try changing the cluster order of **Modified Cluster**. Run the VI. How did the changed order affect the behavior?
21. Close the VI. Do not save changes.

End of Exercise 4-2

Exercise 4-3 Project: Weather Station UI VI

Goal

Create a cluster.

Scenario

In the weather station project, use a cluster to group the temperature data.

Design

Create a cluster constant on the block diagram containing the following:

Table 4-1. Temperature Cluster Constant

Name	Datatype	Default Value	Order
Temperature	Double-Precision Numeric	0	0
T Upper Limit	Double-Precision Numeric	0	1
T Lower Limit	Double-Precision Numeric	0	2
Warning	String	No Warning	3

Notice that the data placed in the cluster are the input and output data of the temperature weather station. Using a cluster to pass the data simplifies the block diagram.

Implementation

1. Open the weather station project.
 - Select **File»Open Project**.
 - Navigate to the C:\Exercises\LabVIEW_Basics_I\Course Project directory, the open Weather Station.lvproj.
2. Open the user interface for the weather station project.
 - In the Project Explorer window, double-click the **Weather Station UI.vi**.
3. Open the block diagram.

In the following steps, build a block diagram similar to that shown in Figure 4-24.

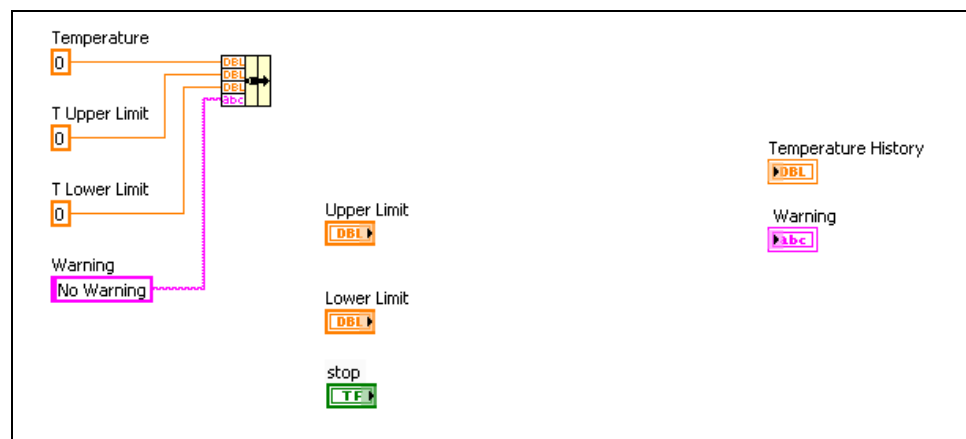


Figure 4-24. Weather Station UI Block Diagram

4. Create the Temperature constant.
 - Place a Numeric Constant on the block diagram.
 - Right-click the Numeric Constant and select **Representation»Double Precision**.
 - Right-click the Numeric Constant and select **Visible Items»Label**.
 - Enter Temperature into the label.
5. Create the T Upper Limit constant.
 - Press the <Ctrl> key and click and drag the Temperature constant to make a copy.

- Rename the new constant T Upper Limit.
6. Create the T Lower Limit constant.
- Press the <Ctrl> key and click and drag the T Upper Limit constant to make a copy.
 - Rename the new constant T Lower Limit.
7. Create the Warning constant.
- Place a String Constant on the block diagram.
 - Right-click the String Constant.
 - Enter No Warning in the String Constant.
 - Select **Visible Items»Label**.
 - Enter Warning into the label.
8. Combine the constants into a cluster.
- Place a Bundle function on the block diagram.
 - Expand the Bundle function to have four input nodes.
 - Wire the constants to the Bundle function as shown in Figure 4-24.
9. Save the VI.

You continue building this block diagram in the next exercise.

End of Exercise 4-3

C. Type Definitions

Type definitions are frequently used to define custom arrays and clusters.

Custom Controls

Use custom controls and indicators to extend the available set of front panel objects. You can create custom user interface components for an application that vary cosmetically from built-in LabVIEW controls and indicators. You can save a custom control or indicator you created in a directory or LLB and use the custom control or indicator on other front panels. You also can create an icon for the custom control or indicator and add it to the **Controls** palette.

Refer to the *LabVIEW Help* topic *Creating Custom Controls, Indicators, and Type Definitions* for more information about creating and using custom controls and type definitions.

Use the Control Editor window to customize controls and indicators. For example, you can change the size, color, and relative position of the elements of a control or indicator and import images into the control or indicator.

You can display the Control Editor window in the following ways:

- Right-click a control or indicator on the front panel and select **Advanced»Customize** from the shortcut menu.
- Use the Positioning tool to select a control or indicator on the front panel and select **Edit»Customize Control**.
- Use the New dialog box.

The Control Editor appears with the selected front panel object in its window. The Control Editor has two modes, edit mode and customize mode.

The Control Editor window toolbar indicates whether you are in edit mode



or in customize mode. The Control Editor window opens in edit mode.



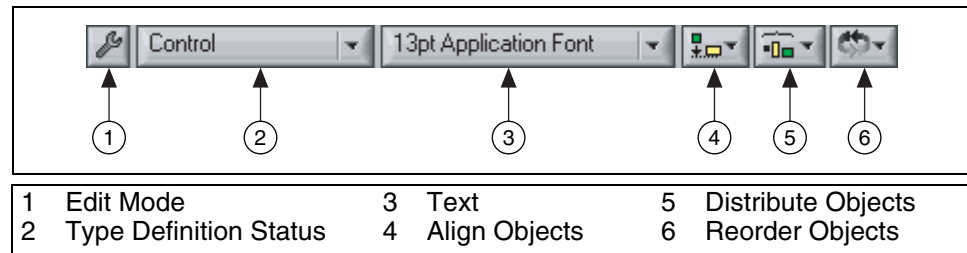
Click the **Edit Mode** button to change to customize mode. Click the **Customize Mode** button to return to edit mode. You also can switch between modes by selecting **Operate»Change to Customize Mode** or **Operate»Change to Edit Mode**.

Use edit mode to change the size or color of a control or indicator and to select options from its shortcut menu, just as you do in edit mode on a front panel.

Use customize mode to make extensive changes to controls or indicators by changing the individual parts of a control or indicator.

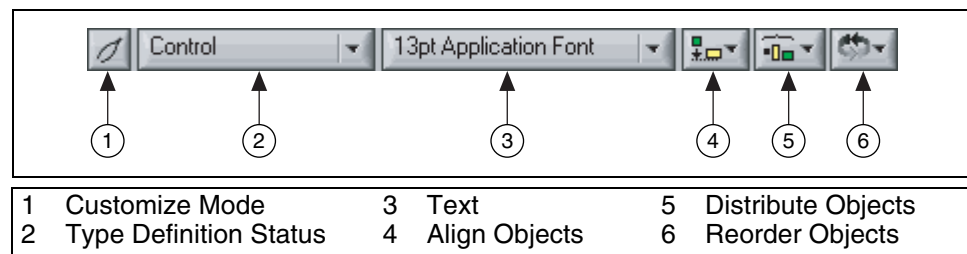
Edit Mode

In edit mode, you can right-click the control and manipulate its settings as you would in the LabVIEW programming environment.



Customize Mode

In customize mode, you can move the individual components of the control around with respect to each other. For a listing of what you can manipulate in customize mode, select **Window»Show Parts Window**.



One way to customize a control is to change its type definition status. You can save a control as a control, a type definition, or a strict type definition, depending on the selection visible in the **Type Def. Status** ring. The control option is the same as a control you would select from the **Controls** palette. You can modify it in any way you need to, and each copy you make and change retains its individual properties.

Saving Controls

After creating a custom control, you can save it for use later. By default, controls saved on disk have a `.ctl` extension.

You also can use the Control Editor to save controls with your own default settings. For example, you can use the Control Editor to modify the defaults of a waveform graph, save it, and later recall it in other VIs.

Type Definition

Use type definitions and strict type definitions to link all the instances of a custom control or indicator to a saved custom control or indicator file. You

can make changes to all instances of the custom control or indicator by editing only the saved custom control or indicator file, which is useful if you use the same custom control or indicator in several VIs.

When you place a custom control or indicator in a VI, no connection exists between the custom control or indicator you saved and the instance of the custom control or indicator in the VI. Each instance of a custom control or indicator is a separate, independent copy. Therefore, changes you make to a custom control or indicator file do not affect VIs already using that custom control or indicator. If you want to link instances of a custom control or indicator to the custom control or indicator file, save the custom control or indicator as a type definition or strict type definition. All instances of a type definition or a strict type definition link to the original file from which you created them.

When you save a custom control or indicator as a type definition or strict type definition, any data type changes you make to the type definition or strict type definition affect all instances of the type definition or strict type definition in all the VIs that use it. Also, cosmetic changes you make to a strict type definition affect all instances of the strict type definition on the front panel.

Type definitions identify the correct data type for each instance of a custom control or indicator. When the data type of a type definition changes, all instances of the type definition automatically update. In other words, the data type of the instances of the type definition change in each VI where the type definition is used. However, because type definitions identify only the data type, only the values that are part of the data type update. For example, on numeric controls, the data range is not part of the data type. Therefore, type definitions for numeric controls do not define the data range for the instances of the type definitions. Also, because the item names in ring controls do not define the data type, changes to ring control item names in a type definition do not change the item names in instances of the type definition. However, if you change the item names in the type definition for an enumerated type control, the instances update because the item names are part of the data type. An instance of a type definition can have its own unique label, description, default value, size, color, or style of control or indicator, such as a knob instead of a slide.

If you change the data type in a type definition, LabVIEW converts the old default value in instances of the type definition to the new data type, if possible. LabVIEW cannot preserve the instance default value if the data type changes to an incompatible type, such as replacing a numeric control or indicator with a string control or indicator. When the data type of a type definition changes to a data type incompatible with the previous type definition, LabVIEW sets the default value of instances to the default value for the new data type. For example, if you change a type definition from a

numeric to a string type, LabVIEW replaces any default values associated with the old numeric data type with empty strings.

A strict type definition forces everything about an instance to be identical to the strict type definition, except the label, description, and default value. As with type definitions, the data type of a strict type definition remains the same everywhere you use the strict type definition. Strict type definitions also define other values, such as range checking on numeric controls and the item names in ring controls. The only VI Server properties available for strict type definitions are those that affect the appearance of the control or indicator, such as Visible, Disabled, Key Focus, Blinking, Position, and Bounds.

You cannot prevent an instance of a strict type definition from automatically updating unless you remove the link between the instance and the strict type definition.

Type definitions and strict type definitions are typically used to create a custom control using a cluster of many controls. If you need to add a new control and pass a new value to every subVI, you can add the new control to the custom control cluster, instead of having to add the new control to each subVIs front panel and making new wires and terminals.

Exercise 4-4 Project: Weather Station CTL

Goal

Create a type-defined enumerated control.

Scenario

In the weather station project, you use an enumerated control to define the states of the state machine. In this exercise create a type-define enum for the enumerated control. Type-defining the control makes it easier to add more states to the state machine in the future.

Design

The enumerated control contains the following data:

Table 4-2. Enumerated Control Data

Items	Digital Display
Acquisition	0
Analysis	1
Data Log	2
Time Check	3

Notice that these items correspond to the states of the state transition diagram designed for this project.

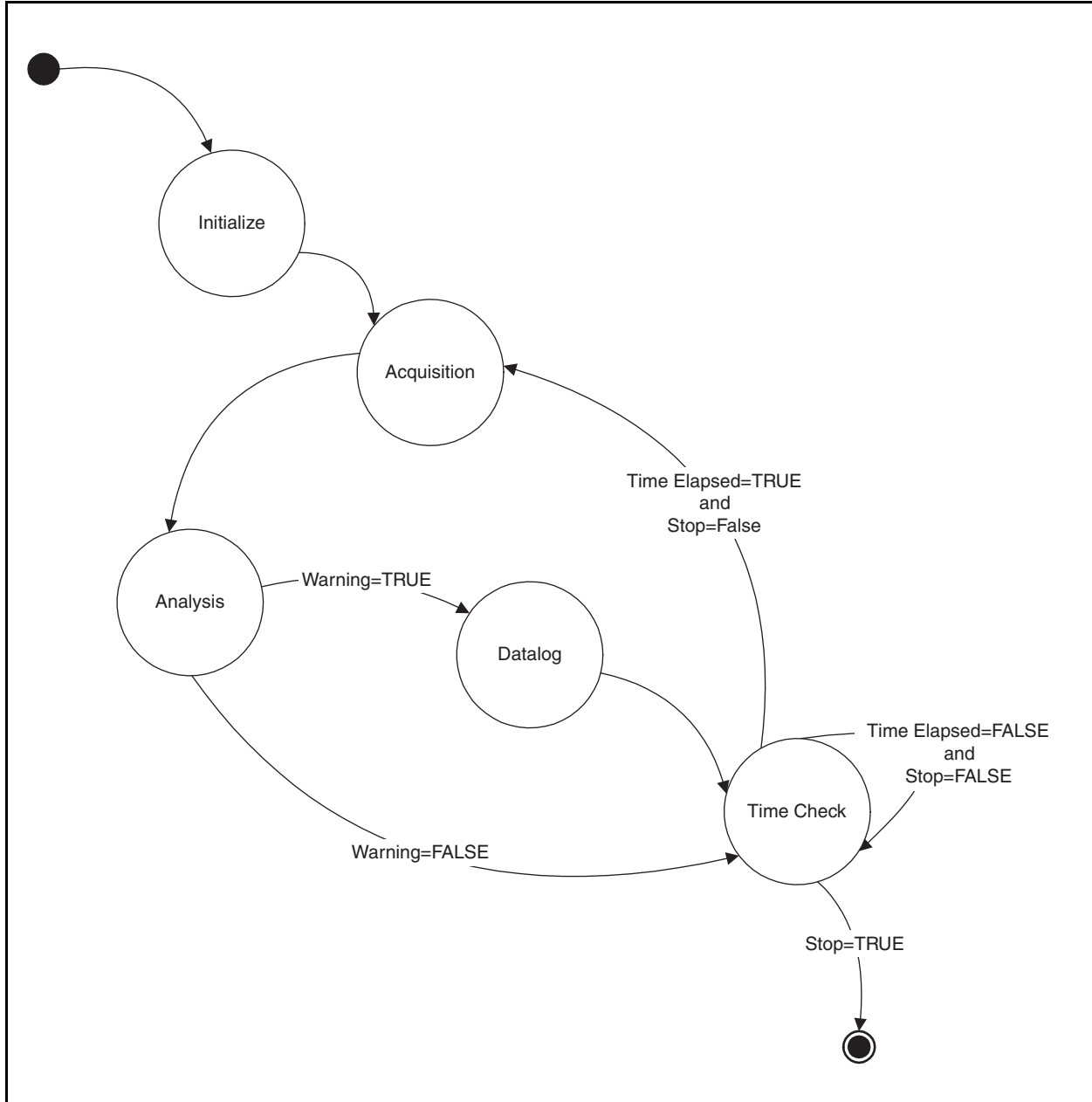


Figure 4-25. Project State Transition Diagram

Implementation

1. Open the weather station project.
 - Select **File»Open Project**.
 - Navigate to the C:\Exercises\LabVIEW_Basics_I\Course Project directory, the open Weather Station.lvproj.
2. Open the user interface for the weather station project.
 - In the **Project Explorer** window, double-click the **Weather Station UI.vi**.
3. Create an enumerated control on the front panel.
 - Place an Enum anywhere on the front panel. You will remove this control later in this exercise.
 - Right-click the **Enum** and select **Properties**.
 - Switch to the **Edit Items** tab.
 - Enter the item names shown in Table 4-2. Figure 4-26 shows an example of the Enum Properties dialog box, with the item names entered. Be careful not to add any blank entries to the list.

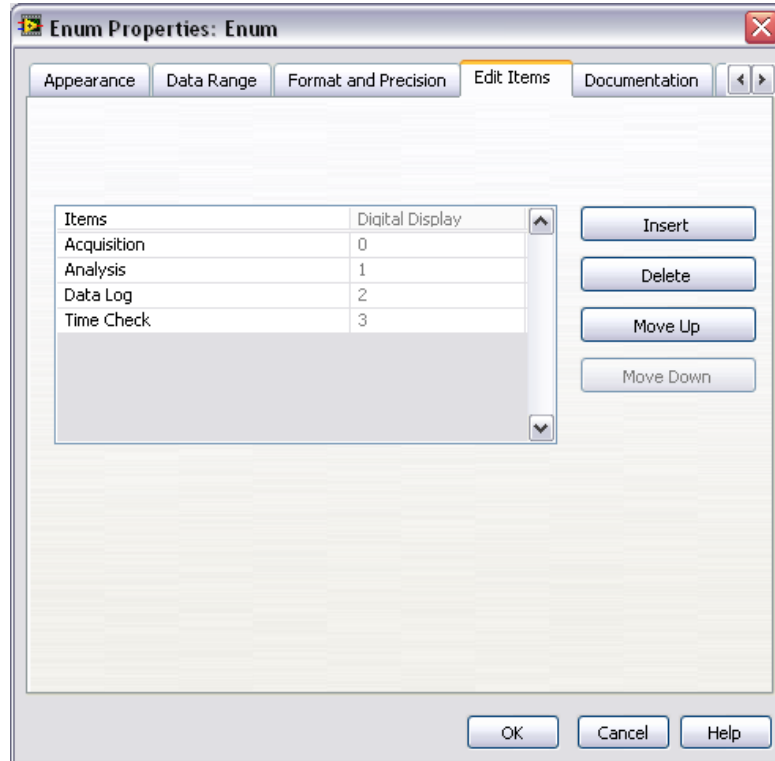


Figure 4-26. Enum Control Dialog Box

- Click **OK** to close the dialog box when you are finished.
4. Create a type definition of the enumerated control.
- Right-click the control and select **Advanced»Customize**. The Control Editor window opens.
 - Select **Type Def.** from the **Type Definition Status** pull down menu, as shown in Figure 4-27.

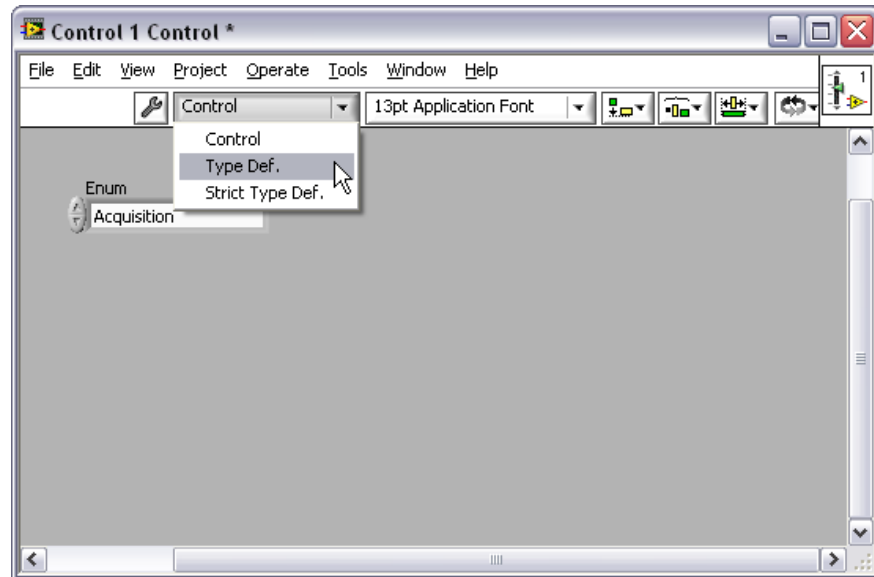


Figure 4-27. Control Editor Window

- Select **File»Save**.
 - Name the control `Weather Station States.ctl` in the `C:\Exercises\LabVIEW Basics I\Course Project\Supporting Files` directory.
 - Select **File»Close** to close the **Control Editor** window.
 - Click **Yes** when prompted to replace the original control.
5. Right-click the **Enum**. Notice that new options are available in the shortcut menu allowing you to interact with the type-definition.
 6. Remove the Enum control from the front panel by changing it to a constant on the block diagram.
 - Press <Ctrl-E> to open the block diagram.
 - Right-click the **Enum** terminal and select **Change to Constant**.

Notice that you can now select between the states on the block diagram. Also notice that the control on the front panel has been removed.

7. Save and close the VI.

You continue building this block diagram in later exercises.

End of Exercise 4-4

Self-Review: Quiz

1. You can create an array of arrays.
 - a. True
 - b. False
2. You have two input arrays wired to a For Loop. Auto-indexing is enabled on both tunnels. One array has 10 elements, the second array has 5 elements. A value of 7 is wired to the Count terminal, as shown in Figure 4-28. What is the value of the Iterations indicator after running this VI?

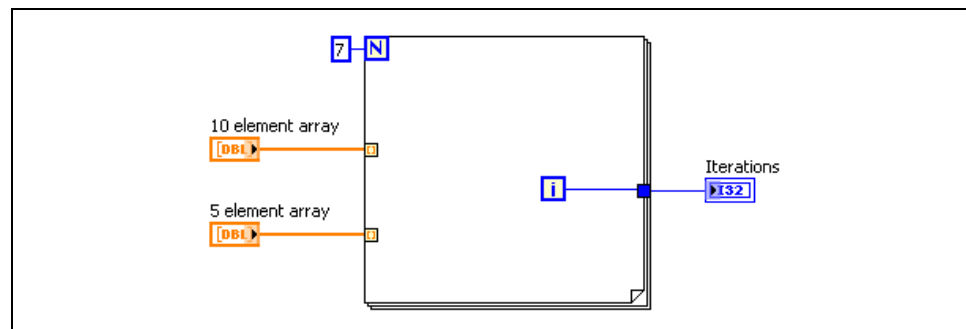


Figure 4-28. What is the Value of the Iteration Indicator?

3. You customize a control, select **Control** from the **Type Def. Status** pull-down menu, and save the control as a `.ctl` file. You then use an instance of the custom control on your front panel. If you open the `.ctl` file and modify the control, does the control on the front panel change?
 - a. Yes
 - b. No
4. You are inputting data that represents a circle. The circle data includes an x position, a y position and a radius. All three pieces of data are double precision. In the future, you might need to store the color of the circle, represented as an integer. How should you represent the circle on your front panel?
 - a. Three separate controls for the two positions and the radius.
 - b. A cluster containing all of the data.
 - c. A custom control containing a cluster.
 - d. A type definition containing a cluster.
 - e. An array with three elements.

Self-Review: Quiz Answers

1. You can create an array of arrays.
 - a. True
 - b. **False**
2. You have two input arrays wired to a For Loop. Auto-indexing is enabled on both tunnels. One array has 10 elements, the second array has 5 elements. A value of 7 is wired to the Count terminal, as shown in the following figure. What is the value of the Iterations indicator after running this VI?

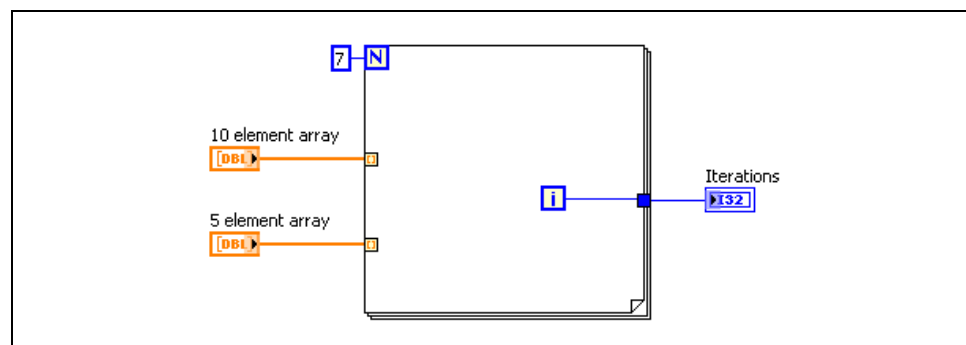


Figure 4-29. What is the value of the Iteration indicator?

Value of Iterations = 4

LabVIEW does not exceed the array size. This helps to protect against programming error. LabVIEW mathematical functions work the same way—if you wire a 10 element array to the *x* input of the Add function, and a 5 element array to the *y* input of the Add function, the output is a 5 element array.

Although the for loop runs 5 times, the iterations are zero based, therefore the value of the Iterations indications is 4.

3. You customize a control, select **Control** from the **Type Def. Status** pull-down menu, and save the control as a `.ctl` file. You then use an instance of the custom control on your front panel. If you open the `.ctl` file and modify the control, does the control on the front panel change?
 - a. Yes
 - b. **No**

4. You are inputting data that represents a circle. The circle data includes an x position, a y position and a radius. All three pieces of data are double precision. In the future, you might need to store the color of the circle, represented as an integer. How should you represent the circle on your front panel?
 - a. Three separate controls for the two positions and the radius.
 - b. A cluster containing all of the data.
 - c. A custom control containing a cluster.
 - d. A type definition containing a cluster.**
 - e. An array with three elements.

Notes

Notes

Debugging VIs

To run a VI, you must wire all the subVIs, functions, and structures with the correct data types for the terminals. Sometimes a VI produces data or runs in a way you do not expect. You can use LabVIEW to configure how a VI runs and to identify problems with block diagram organization or with the data passing through the block diagram.

Topics

- A. Correcting Broken VIs
- B. Debugging Techniques
- C. Undefined or Unexpected Data
- D. Error Checking and Error Handling

A. Correcting Broken VIs

If a VI does not run, it is a broken, or nonexecutable, VI. The **Run** button appears broken, shown as follows, when the VI you are creating or editing contains errors.



If the button still appears broken when you finish wiring the block diagram, the VI is broken and cannot run.

Finding Causes for Broken VIs

Warnings do not prevent you from running a VI. They are designed to help you avoid potential problems in VIs. Errors, however, can break a VI. You must resolve any errors before you can run the VI.

Click the broken **Run** button or select **View»Error List** to find out why a VI is broken. The **Error list** window lists all the errors. The **Items with errors** section lists the names of all items in memory, such as VIs and project libraries that have errors. If two or more items have the same name, this section shows the specific application instance for each item. The **errors and warnings** section lists the errors and warnings for the VI you select in the **Items with errors** section. The **Details** section describes the errors and in some cases recommends how to correct the errors. Click the **Help** button to display a topic in the *LabVIEW Help* that describes the error in detail and includes step-by-step instructions for correcting the error.

Click the **Show Error** button or double-click the error description to highlight the area on the block diagram or front panel that contains the error.

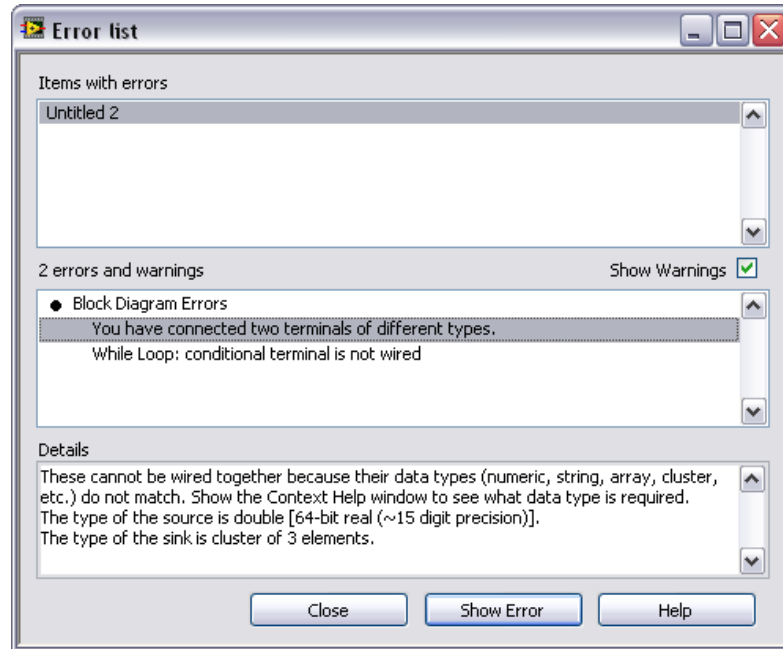


Figure 5-1. Example of the Error List Dialog Box

Common Causes of Broken VIs

The following list contains common reasons why a VI is broken while you edit it:

- The block diagram contains a broken wire because of a mismatch of data types or a loose, unconnected end.

Refer to the *Correcting Broken Wires* topic of the *LabVIEW Help* for information about correcting broken wires.

- A required block diagram terminal is unwired.

Refer to the *Using Wires to Link Block Diagram Objects* topic of the *LabVIEW Help*, for information about setting required inputs and outputs.

- A subVI is broken or you edited its connector pane after you placed its icon on the block diagram of the VI.

Refer to the *Creating SubVIs* topic of the *LabVIEW Help* for information about subVIs.

B. Debugging Techniques

If a VI is not broken, but you get unexpected data, you can use the following techniques to identify and correct problems with the VI or the block diagram data flow:

- Wire the error in and error out parameters at the bottom of most built-in VIs and functions. These parameters detect errors encountered in each node on the block diagram and indicate if and where an error occurred. You also can use these parameters in the VIs you build.
- To eliminate all VI warnings, select **View»Error List** and place a checkmark in the **Show Warnings** checkbox to see all warnings for the VI. Determine the causes and correct them in the VI.
- Triple-click the wire with the Operating tool to highlight its entire path and to ensure that the wires connect to the proper terminals.
- Use the **Context Help** window to check the default values for each function and subVI on the block diagram. VIs and functions pass default values if recommended or optional inputs are unwired. For example, a Boolean input might be set to TRUE if unwired.
- Use the **Find** dialog box to search for subVIs, text, and other objects to correct throughout the VI.
- Select **View»Browse Relationships»This VI's Hierarchy** to find unwired subVIs. Unlike unwired functions, unwired VIs do not always generate errors unless you configure an input to be required. If you mistakenly place an unwired subVI on the block diagram, it executes when the block diagram does. Consequently, the VI might perform extra actions.
- Use execution highlighting to watch the data move through the block diagram.
- Single-step through the VI to view each action of the VI on the block diagram.
- Use the Probe tool to observe intermediate data values and to check the error output of VIs and functions, especially those performing I/O.
- Use breakpoints to pause execution, so you can single-step or insert probes.
- Suspend the execution of a subVI to edit values of controls and indicators, to control the number of times it runs, or to go back to the beginning of the execution of the subVI.
- Determine if the data that one function or subVI passes is undefined. This often happens with numbers. For example, at one point in the VI an operation could have divided a number by zero, thus returning **Inf** (infinity), whereas subsequent functions or subVIs were expecting numbers.

- If the VI runs more slowly than expected, confirm that you turned off execution highlighting in subVIs. Also, close subVI front panels and block diagrams when you are not using them because open windows can affect execution speed.
- Check the representation of controls and indicators to see if you are receiving overflow because you converted a floating-point number to an integer or an integer to a smaller integer. For example, you might wire a 16-bit integer to a function that only accepts 8-bit integers. This causes the function to convert the 16-bit integer to an 8-bit representation, potentially causing a loss of data.
- Determine if any For Loops inadvertently execute zero iterations and produce empty arrays.
- Verify you initialized shift registers properly unless you intend them to save data from one execution of the loop to another.
- Check the cluster element order at the source and destination points. LabVIEW detects data type and cluster size mismatches at edit time, but it does not detect mismatches of elements of the same type.
- Check the node execution order.
- Check that the VI does not contain hidden subVIs. You inadvertently might have hidden a subVI by placing one directly on top of another node or by decreasing the size of a structure without keeping the subVI in view.
- Check the inventory of subVIs the VI uses against the results of **View»Browse Relationships»This VI's SubVIs** and **View»Browse Relationships»Unopened SubVIs** to determine if any extra subVIs exist. Also open the VI Hierarchy window to see the subVIs for a VI. To help avoid incorrect results caused by hidden VIs, specify that inputs to VIs are required.

Execution Highlighting

View an animation of the execution of the block diagram by clicking the **Highlight Execution** button, shown as follows.



Execution highlighting shows the movement of data on the block diagram from one node to another using bubbles that move along the wires. Use execution highlighting in conjunction with single-stepping to see how data values move from node to node through a VI.



Note Execution highlighting greatly reduces the speed at which the VI runs.

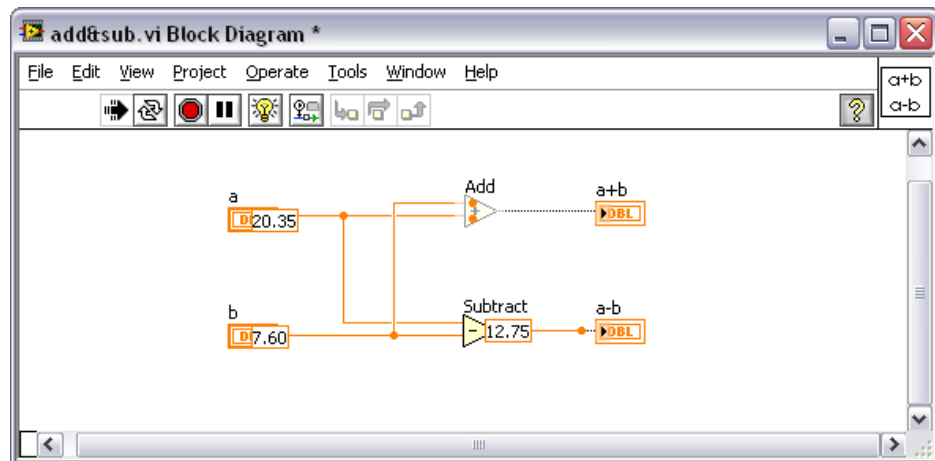


Figure 5-2. Example of Execution Highlighting in Use

Single-Stepping

Single-step through a VI to view each action of the VI on the block diagram as the VI runs. The single-stepping buttons, shown as follows, affect execution only in a VI or subVI in single-step mode.



Enter single-step mode by clicking the **Step Over** or **Step Into** button on the block diagram toolbar. Move the cursor over the **Step Over**, **Step Into**, or **Step Out** button to view a tip strip that describes the next step if you click that button. You can single-step through subVIs or run them normally.

If you single-step through a VI with execution highlighting on, an execution glyph, shown as follows, appears on the icons of the subVIs that are currently running.



Probe Tools

Use the Probe tool, shown as follows, to check intermediate values on a wire as a VI runs.



Use the Probe tool if you have a complicated block diagram with a series of operations, any one of which might return incorrect data. Use the Probe tool

with execution highlighting, single-stepping, and breakpoints to determine if and where data is incorrect. If data is available, the probe immediately updates during single-stepping or when you pause at a breakpoint. When execution pauses at a node because of single-stepping or a breakpoint, you also can probe the wire that just executed to see the value that flowed through that wire.

Types of Probes

You can check intermediate values on a wire when a VI runs by using a generic probe, by using an indicator on the **Controls** palette to view the data, by using a supplied probe, by using a customized supplied probe, or by creating a new probe.

Generic

Use the generic probe to view the data that passes through a wire. Right-click a wire and select **Custom Probe»Generic Probe** from the shortcut menu to use the generic probe.

The generic probe displays the data. You cannot configure the generic probe to respond to the data.

LabVIEW displays the generic probe when you right-click a wire and select **Probe**, unless you already specified a custom or supplied probe for the data type.

You can debug a custom probe similar to a VI. However, a probe cannot probe its own block diagram, nor the block diagram of any of its subVIs. When debugging probes, use the generic probe.

Using Indicators to View Data

You also can use an indicator to view the data that passes through a wire. For example, if you view numeric data, you can use a chart within the probe to view the data. Right-click a wire, select **Custom Probe»Controls** from the shortcut menu, and select the indicator you want to use. You also can click the **Select a Control** icon on the **Controls** palette and select any custom control or type definition saved on the computer or in a shared directory on a server. LabVIEW treats type definitions as custom controls when you use them to view probed data.

If the data type of the indicator you select does not match the data type of the wire you right-clicked, LabVIEW does not place the indicator on the wire.

Supplied

Supplied probes are VIs that display comprehensive information about the data that passes through a wire. For example, the VI Refnum Probe returns

information about the VI name, the VI path, and the hex value of the reference. You also can use a supplied probe to respond based on the data that flows through the wire. For example, use an Error probe on an error cluster to receive the status, code, source, and description of the error and specify if you want to set a conditional breakpoint if an error or warning occurs.

The supplied probes appear at the top of the **Custom Probe** shortcut menu. Right-click a wire and select **Custom Probe** from the shortcut menu to select a supplied probe. Only probes that match the data type of the wire you right-click appear on the shortcut menu.

Refer to the Using Supplied Probes VI in the `labview\examples\general\probes.llb` for an example of using supplied probes.

Custom

Use the Custom Probe Wizard to create a probe based on an existing probe or to create a new probe. Right-click a wire and select **Custom Probe»New** from the shortcut menu to display the Custom Probe Wizard. Create a probe when you want to have more control over how LabVIEW probes the data that flows through a wire. When you create a new probe, the data type of the probe matches the data type of the wire you right-clicked. If you want to edit the probe you created, you must open it from the directory where you saved it.

After you select a probe from the **Custom Probe** shortcut menu, navigate to it using the **Select a Control** palette option, or create a new probe using the Custom Probe Wizard, that probe becomes the default probe for that data type, and LabVIEW loads that probe when you right-click a wire and select **Probe** from the shortcut menu. LabVIEW only loads probes that exactly match the data type of the wire you right-click. That is, a double precision floating-point numeric probe cannot probe a 32-bit unsigned integer wire even though LabVIEW can convert the data.



Note If you want a custom probe to be the default probe for a particular data type, save the probe in the `user.lib_probes\default` directory. Do not save probes in the `vi.lib_probes` directory because LabVIEW overwrites those files when you upgrade or reinstall.

Breakpoints

Use the Breakpoint tool, shown as follows, to place a breakpoint on a VI, node, or wire on the block diagram and pause execution at that location.



When you set a breakpoint on a wire, execution pauses after data passes through the wire. Place a breakpoint on the block diagram to pause execution after all nodes on the block diagram execute.

When a VI pauses at a breakpoint, LabVIEW brings the block diagram to the front and uses a marquee to highlight the node or wire that contains the breakpoint. When you move the cursor over an existing breakpoint, the black area of the Breakpoint tool cursor appears white.

When you reach a breakpoint during execution, the VI pauses and the **Pause** button appears red. You can take the following actions:

- Single-step through execution using the single-stepping buttons.
- Probe wires to check intermediate values.
- Change values of front panel controls.
- Click the **Pause** button to continue running to the next breakpoint or until the VI finishes running.

Suspending Execution

Suspend execution of a subVI to edit values of controls and indicators, to control the number of times the subVI runs before returning to the caller, or to go back to the beginning of the execution of the subVI. You can cause all calls to a subVI to start with execution suspended, or you can suspend a specific call to a subVI.

To suspend all calls to a subVI, open the subVI and select **Operate» Suspend when Called**. The subVI automatically suspends when another VI calls it. If you select this menu item when single-stepping, the subVI does not suspend immediately. The subVI suspends when it is called.

To suspend a specific subVI call, right-click the subVI node on the block diagram and select **SubVI Node Setup** from the shortcut menu. Place a checkmark in the **Suspend when called** checkbox to suspend execution only at that instance of the subVI.

The **VI Hierarchy** window, which you display by selecting **View»VI Hierarchy**, indicates whether a VI is paused or suspended. An arrow glyph, shown as follows, indicates a VI that is running regularly or single-stepping.



A pause glyph, shown as follows, indicates a paused or suspended VI.



A green pause glyph, or a hollow glyph in black and white, indicates a VI that pauses when called. A red pause glyph, or a solid glyph in black and white, indicates a VI that is currently paused. An exclamation point glyph, shown as follows, indicates that the subVI is suspended.



A VI can be suspended and paused at the same time.

Determining the Current Instance of a SubVI

When you pause a subVI, the **Call list** pull-down menu on the toolbar lists the chain of callers from the top-level VI down to the subVI. This list is not the same list you see when you select **Browse»This VI's Callers**, which lists all calling VIs regardless of whether they are currently running. Use the **Call list** menu to determine the current instance of the subVI if the block diagram contains more than one instance. When you select a VI from the **Call list** menu, its block diagram opens, and LabVIEW highlights the current instance of the subVI.

C. Undefined or Unexpected Data

Undefined data, which are NaN (not a number) or Inf (infinity), invalidate all subsequent operations. Floating-point operations return the following two symbolic values that indicate faulty computations or meaningless results:

- NaN (not a number) represents a floating-point value that invalid operations produce, such as taking the square root of a negative number.
- Inf (infinity) represents a floating-point value that operations produce, such as dividing a number by zero.

LabVIEW does not check for overflow or underflow conditions on integer values. Overflow and underflow for floating-point numbers is in accordance with IEEE 754, *Standard for Binary Floating-Point Arithmetic*.

Floating-point operations propagate NaN and Inf reliably. When you explicitly or implicitly convert NaN or Inf to integers or Boolean values, the values become meaningless. For example, dividing 1 by zero produces Inf. Converting Inf to a 16-bit integer produces the value 32,767, which appears to be a normal value.

Before you convert data to integer data types, use the Probe tool to check intermediate floating-point values for validity. Check for NaN by wiring the Comparison function Not A Number/Path/Refnum? to the value you suspect is invalid.

Do not rely on special values such as NaN, Inf, or empty arrays to determine if a VI produces undefined data. Instead, confirm that the VI produces defined data by making the VI report an error if it encounters a situation that is likely to produce undefined data.

For example, if you create a VI that uses an incoming array to auto-index a For Loop, determine what you want the VI to do when the input array is empty. Either produce an output error code, substitute defined data for the value that the loop creates, or use a Case structure that does not execute the For Loop if the array is empty.

D. Error Checking and Error Handling

No matter how confident you are in the VI you create, you cannot predict every problem a user can encounter. Without a mechanism to check for errors, you know only that the VI does not work properly. Error checking tells you why and where errors occur.

Automatic Error Handling

Each error has a numeric code and a corresponding error message.

By default, LabVIEW automatically handles any error when a VI runs by suspending execution, highlighting the subVI or function where the error occurred, and displaying an error dialog box.

To disable automatic error handling for the current VI, select **File»VI Properties** and select **Execution** from the **Category** pull-down menu. To disable automatic error handling for any new, blank VIs you create, select **Tools»Options** and select **Block Diagram** from the **Category** list. To disable automatic error handling for a subVI or function within a VI, wire its **error out** parameter to the **error in** parameter of another subVI or function or to an **error out** indicator.

Manual Error Handling

You can choose other error handling methods. For example, if an I/O VI on the block diagram times out, you might not want the entire application to stop and display an error dialog box. You also might want the VI to retry for a certain period of time. In LabVIEW, you can make these error handling decisions on the block diagram of the VI.

Use the LabVIEW error handling VIs and functions on the **Dialog & User Interface** palette and the **error in** and **error out** parameters of most VIs and functions to manage errors. For example, if LabVIEW encounters an error, you can display the error message in different kinds of dialog boxes. Use error handling in conjunction with the debugging tools to find and manage errors.

VIs and functions return errors in one of two ways—with numeric error codes or with an error cluster. Typically, functions use numeric error codes, and VIs use an error cluster, usually with error inputs and outputs.

When you perform any kind of input and output (I/O), consider the possibility that errors might occur. Almost all I/O functions return error information. Include error checking in VIs, especially for I/O operations (file, serial, instrumentation, data acquisition, and communication), and provide a mechanism to handle errors appropriately.

Use the LabVIEW error handling VIs, functions, and parameters to manage errors. For example, if LabVIEW encounters an error, you can display the error message in a dialog box. Or you can fix the error programmatically then erase the error by wiring the error out output of the subVI or function to the error in input of the Clear Errors VI. Use error handling in conjunction with the debugging tools to find and manage errors. National Instruments strongly recommends using error handling.

Error Clusters

Use the error cluster controls and indicators to create error inputs and outputs in subVIs.

The **error in** and **error out** clusters include the following components of information:

- **status** is a Boolean value that reports TRUE if an error occurred.
- **code** is a 32-bit signed integer that identifies the error numerically. A nonzero error code coupled with a **status** of FALSE signals a warning rather than an error.
- **source** is a string that identifies where the error occurred.

Error handling in LabVIEW follows the dataflow model. Just as data values flow through a VI, so can error information. Wire the error information from the beginning of the VI to the end. Include an error handler VI at the end of the VI to determine if the VI ran without errors. Use the **error in** and **error out** clusters in each VI you use or build to pass the error information through the VI.

As the VI runs, LabVIEW tests for errors at each execution node. If LabVIEW does not find any errors, the node executes normally. If LabVIEW detects an error, the node passes the error to the next node without executing that part of the code. The next node does the same thing, and so on. At the end of the execution flow, LabVIEW reports the error.

Explain Error

When an error occurs, right-click within the cluster border and select **Explain Error** from the shortcut menu to open the **Explain Error** dialog box. The **Explain Error** dialog box contains information about the error. The shortcut menu includes an **Explain Warning** option if the VI contains warnings but no errors.

You also can access the **Explain Error** dialog box from the **Help»Explain Error** menu.

VIs and functions return errors in one of two ways—with numeric error codes or with an error cluster. Typically, functions use numeric error codes, and VIs use an error cluster, usually with error inputs and outputs.

Using Case Structures for Error Handling

The following example is Case structure with an error cluster used to define the cases.

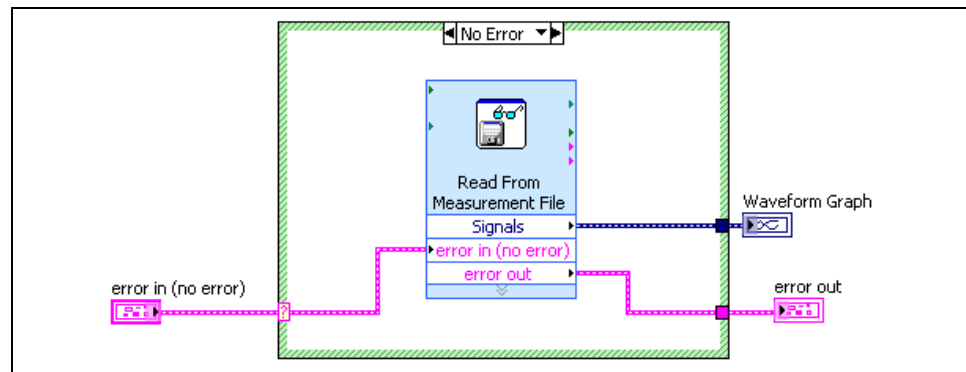


Figure 5-3. No Error Case

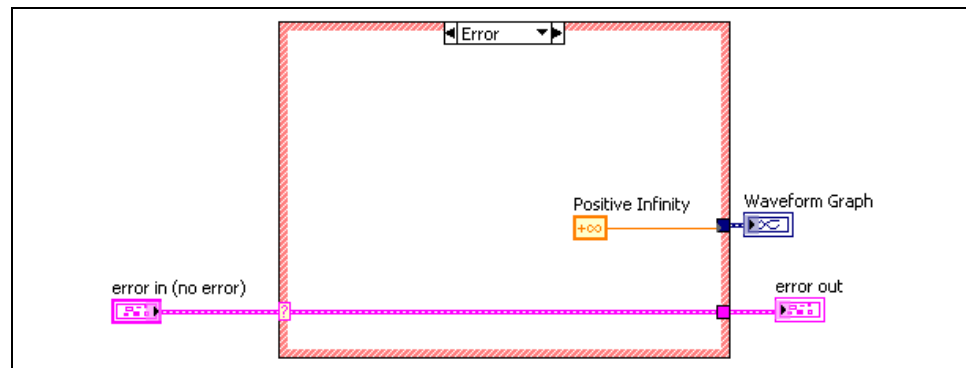


Figure 5-4. Error Case

When you wire an error cluster to the selector terminal of a Case structure, the case selector label displays two cases—Error and No Error—and the border of the Case structure changes color—red for Error and green for No Error. If an error occurs, the Case structure executes the Error subdiagram.

When an error cluster is wired to the selection terminal, the Case structure recognizes only the **status** Boolean of the cluster.

Using While Loops for Error Handling

You can wire an error cluster to the conditional terminal of a While Loop to stop the iteration of the While Loop. When you wire the error cluster to the conditional terminal, only the TRUE or FALSE value of the **status**

parameter of the error cluster is passed to the terminal. When an error occurs, the While Loop stops.

When an error cluster is wired to the conditional terminal, the shortcut menu items **Stop if True** and **Continue if True** change to **Stop on Error** and **Continue while Error**.

In Figure 5-5, the error cluster and a stop button are used together to determine when to stop the loop. This is the recommended method for stopping most loops.

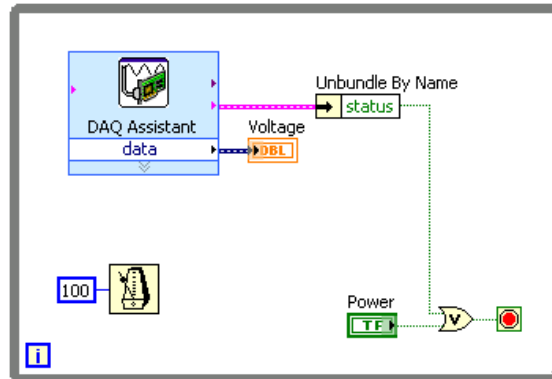


Figure 5-5. Stopping a While Loop

Exercise 5-1 Concept: Debugging

Goal:

Use the debugging tools built into LabVIEW.

Description:

Complete the following steps to load a broken VI and correct the error. Use single-stepping and execution highlighting to step through the VI.

1. Open and examine the Debug Exercise (Main) VI.
 - Select **File>Open**.
 - Open `Debug Exercise (Main).vi` in the `C:\Exercises\LabVIEW Basics I\Debugging` directory.

The following front panel appears.

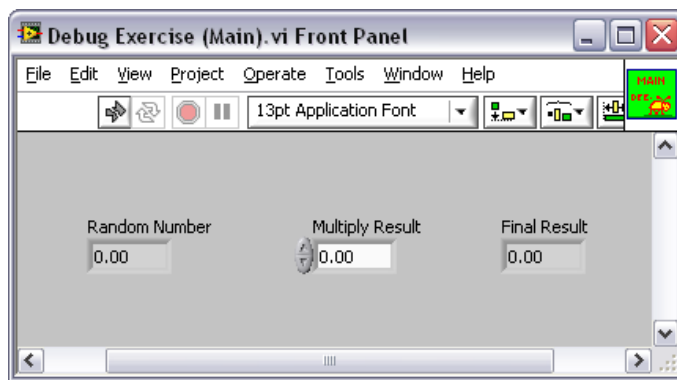


Figure 5-6. Debug Exercise (Main).vi Front Panel



- Notice the **Run** button on the toolbar appears broken, shown at left, indicating that the VI is broken and cannot run.

2. Display and examine the block diagram of Debug Exercise (Main) VI.
 - Select **Window»Show Block Diagram** to display the block diagram shown in Figure 5-7.

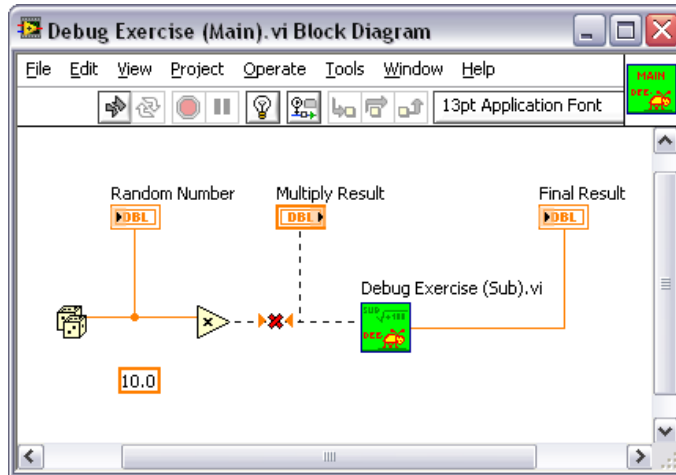


Figure 5-7. Debug Exercise (Main).vi Block Diagram



- The Random Number (0-1) function produces a random number between 0 and 1.



123

- The Multiply function multiplies the random number by 10.0.
- The numeric constant is the number to multiply by the random number.



- The Debug Exercise (Sub) VI, located in the C:\Exercises\LabVIEW Basics I\Debugging\Supporting Files directory, adds 100.0 and calculates the square root of the value.

3. Find and fix each error.

- Click the broken **Run** button to display the **Error list** window, which lists all the errors.
- Select an error description in the **Error list** window. The **Details** section describes the error and in some cases recommends how to correct the error.
- Click the **Help** button to display a topic in the *LabVIEW Help* that describes the error in detail and includes step-by-step instructions for correcting the error.
- Click the **Show Error** button or double-click the error description to highlight the area on the block diagram that contains the error.

- Use the **Error list** window to fix each error.
- 4. Select **File»Save** to save the VI.
- 5. Display the front panel by clicking it or by selecting **Window»Show Front Panel**.
- 6. Click the **Run** button to run the VI several times.
- 7. Select **Window»Show Block Diagram** to display the block diagram.
- 8. Animate the flow of data through the block diagram.



- Click the **Highlight Execution** button, shown at left, on the toolbar to enable execution highlighting.



- Click the **Step Into** button, shown at left, to start single-stepping. Execution highlighting shows the movement of data on the block diagram from one node to another using bubbles that move along the wires. Nodes blink to indicate they are ready to execute.



- Click the **Step Over** button, shown at left, after each node to step through the entire block diagram. Each time you click the **Step Over** button, the current node executes and pauses at the next node.

- Data appear on the front panel as you step through the VI. The VI generates a random number and multiplies it by 10.0. The subVI adds 100.0 and takes the square root of the result.



- When a blinking border surrounds the entire block diagram, click the **Step Out** button, shown at left, to stop single-stepping through the Debug Exercise (Main) VI.

- 9. Single-step through the VI and its subVI.

- Click the **Step Into** button to start single-stepping.



- When the Debug Exercise (Sub) VI blinks, click the **Step Into** button. Notice the run button on the subVI.



- Display the Debug Exercise (Main) VI block diagram by clicking it. A green glyph, shown at left, appears on the subVI icon on the Debug Exercise (Main) VI block diagram, indicating that the subVI is running.

- Display the Debug Exercise (Sub) VI block diagram by clicking it.

- e. Click the **Step Out** button twice to finish single-stepping through the subVI block diagram. The Debug Exercise (Main) VI block diagram is active.
- f. Click the **Step Out** button to stop single-stepping.

10. Use a probe to check intermediate values on a wire as a VI runs.



- Use the Probe tool, shown at left, to click any wire. The **Probe** window appears.

LabVIEW numbers the **Probe** window automatically and displays the same number in a glyph on the wire you clicked.

- Single-step through the VI again. The **Probe** window displays data passed along the wire.

11. Place breakpoints on the block diagram to pause execution at that location.



- Use the Breakpoint tool, shown at left, to click nodes or wires. Place a breakpoint on the block diagram to pause execution after all nodes on the block diagram execute.
- Click the **Run** button to run the VI. When you reach a breakpoint during execution, the VI pauses and the **Pause** button on the toolbar appears red.



- Click the **Continue** button, shown at left, to continue running to the next breakpoint or until the VI finishes running.
- Use the Breakpoint tool to click the breakpoints you set and remove them.

12. Click the **Highlight Execution** button to disable execution highlighting.

13. Select **File»Close** to close the VI and all open windows.

End of Exercise 5-1

Self Review: Quiz

1. How do you disable automatic error handling?
 - a. Select **Operate»Disable Error Handling**.
 - b. Enable execution highlighting.
 - c. Wire the error out cluster of a subVI to the error in cluster of another subVI.
 - d. Place a checkmark in the **Show Warnings** checkbox of the **Error List** dialog box.
2. Which of the following are the contents of the error cluster? (multiple answer)
 - a. Status: Boolean
 - b. Error: String
 - c. Code: 32-bit integer
 - d. Source: String

Self Review: Quiz Answers

1. How do you disable automatic error handling?
 - a. Select Operate»Disable Error Handling.
 - b. Enable execution highlighting.
 - c. Wire the error out cluster of a subVI to the error in cluster of another subVI.**
 - d. Place a checkmark in the Show Warnings checkbox of the Error List dialog box.
2. Which of the following are the contents of the error cluster? (multiple answer)
 - a. Status: Boolean**
 - b. Error: 32-bit integer
 - c. Code: 32-bit integer**
 - d. Source: String**

Notes

Developing Modular Applications

This lesson describes how to develop modular applications. The power of LabVIEW lies in the hierarchical nature of the VI. After you create a VI, you can use it on the block diagram of another VI. There is no limit on the number of layers in the hierarchy. Using modular programming helps you manage changes and debug the block diagram quickly.

Topics

- A. What is Modularity?
- B. Icon and Connector Pane
- C. Using SubVIs

A. What is Modularity?

Modularity defines the degree to which your VI is composed of discrete components such that a change to one component has minimal impact on other components. These components are called modules, or subVIs. Modularity increases the readability and reusability of your VIs.

A VI within another VI is called a subVI. A subVI corresponds to a subroutine in text-based programming languages. When you double-click a subVI, a front panel and block diagram appear, rather than a dialog box in which you can configure options. The front panel includes controls and indicators. The block diagram includes wires, front panel icons, functions, possibly subVIs, and other LabVIEW objects that also might look familiar.

The upper right corner of the front panel and block diagram displays the icon for the VI. This icon is the same as the icon that appears when you place the VI on the block diagram.

As you create VIs, you might find that you perform a certain operation frequently. Consider using subVIs or loops to perform that operation repetitively. For example, the following block diagram contains two identical operations.

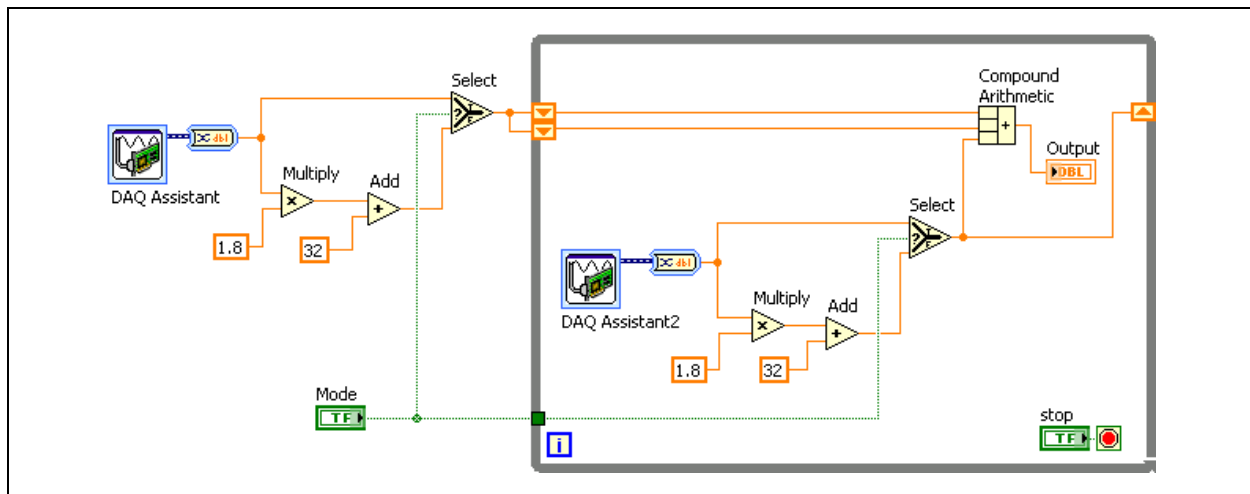


Figure 6-1. Block Diagram with Two Identical Operations

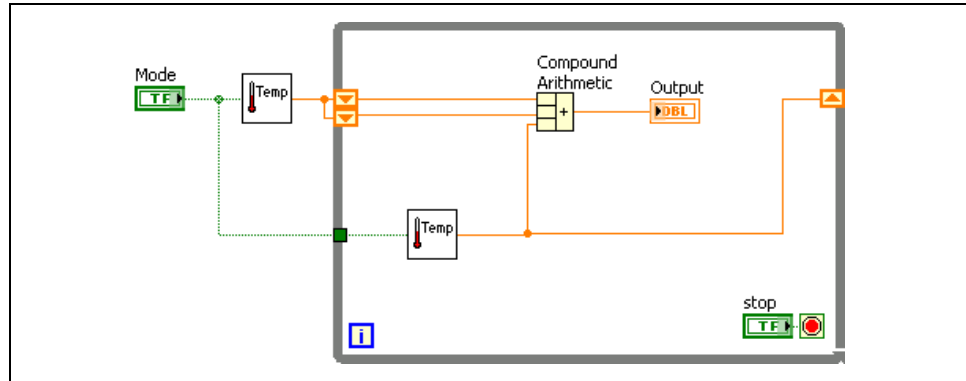
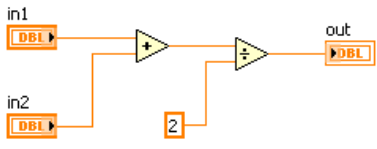
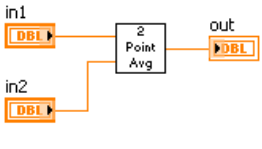


Figure 6-2. Block Diagram with SubVIs for Identical Operations

The example calls the Temperature VI as a subVI twice on its block diagram and functions the same as the previous block diagram. You also can reuse the subVI in other VIs.

The following pseudo-code and block diagrams demonstrate the analogy between subVIs and subroutines.

Function Code	Calling Program Code
<pre>function average (in1, in2, out) { out = (in1 + in2)/2.0; }</pre>	<pre>main { average (point1, point2, pointavg) }</pre>
SubVI Block Diagram	Calling VI Block Diagram
	

B. Icon and Connector Pane

After you build a VI front panel and block diagram, build the icon and the connector pane so you can use the VI as a subVI. The icon and connector pane correspond to the function prototype in text-based programming languages. Every VI displays an icon, such as the one shown as follows, in the upper right corner of the front panel and block diagram windows.



An icon is a graphical representation of a VI. It can contain text, images, or a combination of both. If you use a VI as a subVI, the icon identifies the subVI on the block diagram of the VI. You can double-click the icon to customize or edit it.

You also need to build a connector pane, shown as follows, to use the VI as a subVI.



The connector pane is a set of terminals that correspond to the controls and indicators of that VI, similar to the parameter list of a function call in text-based programming languages. The connector pane defines the inputs and outputs you can wire to the VI so you can use it as a subVI. A connector pane receives data at its input terminals and passes the data to the block diagram code through the front panel controls and receives the results at its output terminals from the front panel indicators.

Creating an Icon

The default icon contains a number that indicates how many new VIs you have opened since launching LabVIEW. Create custom icons to replace the default icon by right-clicking the icon in the upper right corner of the front panel or block diagram and selecting Edit Icon from the shortcut menu or by double-clicking the icon in the upper right corner of the front panel.

You also can drag a graphic from anywhere in your file system and drop it in the upper right corner of the front panel or block diagram. LabVIEW converts the graphic to a 32 × 32 pixel icon.

Refer to the Icon Art Glossary at ni.com for standard graphics to use in a VI icon.

Refer to the National Instruments Web site at ni.com/info and enter the info code `expnr7` for standard graphics to use in a VI icon.

Depending on the type of monitor you use, you can design a separate icon for monochrome, 16-color, and 256-color mode. LabVIEW uses the monochrome icon for printing unless you have a color printer.

Use the tools on the left side of the **Icon Editor** dialog box to create the icon design in the editing area. The normal size image of the icon appears in the appropriate box to the right of the editing area, as shown in Figure 6-3.



Figure 6-3. Icon Editor Window

Use the **Edit** menu to cut, copy, and paste images from and to the icon. When you select a portion of the icon and paste an image, LabVIEW resizes the image to fit into the selection area. You also can drag a graphic from anywhere in your file system and drop it in the upper right corner of the front panel or block diagram. LabVIEW converts the graphic to a 32 × 32 pixel icon.

Use the **Copy from** option on the right side of the **Icon Editor** dialog box to copy from a color icon to a black-and-white icon and vice versa. After you select a **Copy from** option, click the **OK** button to complete the change.



Note If you do not draw a complete border around a VI icon, the icon background appears transparent. When you select the icon on the block diagram, a selection marquee appears around each individual graphic element in the icon.

Use the tools on the left side of the **Icon Editor** dialog box to create the icon design in the editing area. The normal size image of the icon appears in the appropriate box to the right of the editing area. The following tasks can be performed with these tools:



Use the Pencil tool to draw and erase pixel by pixel.



Use the Line tool to draw straight lines. To draw horizontal, vertical, and diagonal lines, press the <Shift> key while you use this tool to drag the cursor.



Use the Color Copy tool to copy the foreground color from an element in the icon.



Use the Fill tool to fill an outlined area with the foreground color.



Use the Rectangle tool to draw a rectangular border in the foreground color. Double-click this tool to frame the icon in the foreground color.



Use the Filled Rectangle tool to draw a rectangle with a foreground color frame and filled with the background color. Double-click this tool to frame the icon in the foreground color and fill it with the background color.



Use the Select tool to select an area of the icon to cut, copy, move, or make other changes. Double-click this tool and press the <Delete> key to delete the entire icon.



Use the Text tool to enter text into the icon. Double-click this tool to select a different font. **(Windows)** The **Small Fonts** option works well in icons.



Use the Foreground/Background tool to display the current foreground and background colors. Click each rectangle to display a color palette from which you can select new colors.

Use the options on the right side of the editing area to perform the following tasks:

- **Show Terminals**—Displays the terminal pattern of the connector pane
- **OK**—Saves the drawing as the icon and returns to the front panel
- **Cancel**—Returns to the front panel without saving any changes

The menu bar in the **Icon Editor** dialog box contains more editing options such as **Undo**, **Redo**, **Cut**, **Copy**, **Paste**, and **Clear**.

Setting up the Connector Pane

Define connections by assigning a front panel control or indicator to each of the connector pane terminals. To define a connector pane, right-click the icon in the upper right corner of the front panel and select **Show Connector** from the shortcut menu to display the connector pane. The connector pane appears in place of the icon. When you view the connector pane for the first time, you see a connector pattern. You can select a different pattern by right-clicking the connector pane and selecting **Patterns** from the shortcut menu.

Each rectangle on the connector pane represents a terminal. Use the rectangles to assign inputs and outputs. The default connector pane pattern is $4 \times 2 \times 2 \times 4$. If you anticipate changes to the VI that would require a new input or output, keep the default connector pane pattern to leave extra terminals unassigned.

The following front panel has four controls and one indicator, so LabVIEW displays four input terminals and one output terminal on the connector pane.

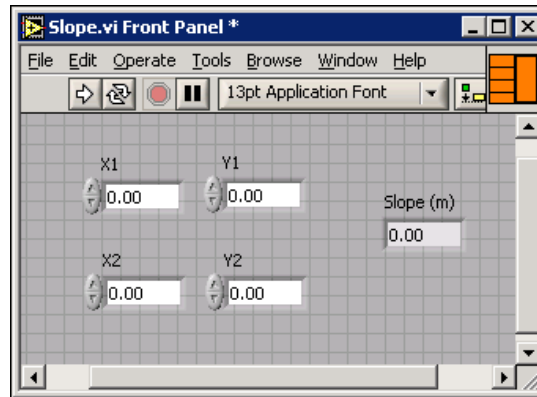


Figure 6-4. Slope VI Front Panel

Selecting and Modifying Terminal Patterns

Select a different terminal pattern for a VI by right-clicking the connector pane and selecting Patterns from the shortcut menu. For example, you can select a connector pane pattern with extra terminals. You can leave the extra terminals unconnected until you need them. This flexibility enables you to make changes with minimal effect on the hierarchy of the VIs.

You also can have more front panel controls or indicators than terminals.

A solid border highlights the pattern currently associated with the icon. You can assign up to 28 terminals to a connector pane.



The most commonly used pattern is shown at left. This pattern is used as a standard to assist in simplifying wiring.

Figure 6-5 shows an example of the standard layout used for terminal patterns. The top inputs and outputs are commonly used for passing references and the bottom inputs and outputs are used for error handling.

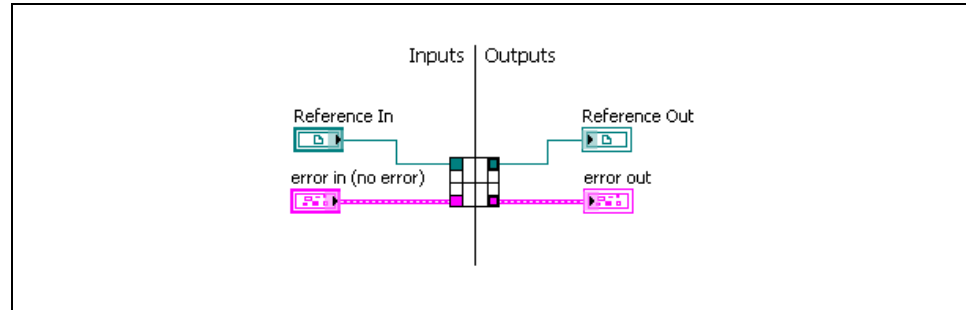


Figure 6-5. Example Terminal Pattern Layout



Note Assigning more than 16 terminals to a VI can reduce readability and usability.

Assigning Terminals to Controls and Indicators

After you select a pattern to use for the connector pane, you must define connections by assigning a front panel control or indicator to each of the connector pane terminals. When you link controls and indicators to the connector pane, place inputs on the left and outputs on the right to prevent complicated, unclear wiring patterns in your VIs.

To assign a terminal to a front panel control or indicator, click a terminal of the connector pane, then click the front panel control or indicator you want to assign to that terminal. Click an open space on the front panel. The terminal changes to the data type color of the control to indicate that you connected the terminal.

You also can select the control or indicator first and then select the terminal.



Note Although you use the Wiring tool to assign terminals on the connector pane to front panel controls and indicators, no wires are drawn between the connector pane and these controls and indicators.

C. Using SubVIs

To place a subVI on the block diagram, click the **Select a VI** button on the **Functions** palette. Navigate to the VI you want to use as a subVI and double-click to place it on the block diagram.

You also can place an open VI on the block diagram of another open VI. Use the Positioning tool to click the icon in the upper right corner of the front panel or block diagram of the VI you want to use as a subVI and drag the icon to the block diagram of the other VI.

Opening and Editing SubVIs

To display the front panel of a subVI from the calling VI, use the Operating or Positioning tool to double-click the subVI on the block diagram. You also can select **View»Browse Relationships»This VI's SubVIs**. To display the block diagram of a subVI from the calling VI, press the <Ctrl> key and use the Operating or Positioning tool to double-click the subVI on the block diagram.

You can edit a subVI by using the Operating or Positioning tool to double-click the subVI on the block diagram. When you save the subVI, the changes affect all calls to the subVI, not just the current instance.

Setting Required, Recommended, and Optional Inputs and Outputs

In the **Context Help** window, the labels of required terminals appear bold, recommended terminals appear as plain text, and optional terminals appear dimmed. The labels of optional terminals do not appear if you click the **Hide Optional Terminals and Full Path** button, shown as follows, in the **Context Help** window.



You can designate which inputs and outputs are required, recommended, and optional to prevent users from forgetting to wire subVI terminals.

Right-click a terminal in the connector pane and select **This Connection Is** from the shortcut menu. A checkmark indicates the terminal setting. Select **Required**, **Recommended**, or **Optional**.

For terminal inputs, required means that the block diagram on which you placed the subVI will be broken if you do not wire the required inputs. Required is not available for terminal outputs. For terminal inputs and outputs, recommended or optional means that the block diagram on which you placed the subVI can execute if you do not wire the recommended or

optional terminals. If you do not wire the terminals, the VI does not generate any warnings.

Inputs and outputs of VIs in `vi.lib` are already marked as **Required**, **Recommended**, or **Optional**. LabVIEW sets inputs and outputs of VIs you create to **Recommended** by default. Set a terminal setting to required only if the VI must have the input or output to run properly.

Creating a SubVI from an existing VI

You can simplify the block diagram of a VI by converting sections of the block diagram into subVIs. Convert a section of a VI into a subVI by using the Positioning tool to select the section of the block diagram you want to reuse and selecting **Edit>Create SubVI**. An icon for the new subVI replaces the selected section of the block diagram. LabVIEW creates controls and indicators for the new subVI, automatically configures the connector pane based on the number of control and indicator terminals you selected, and wires the subVI to the existing wires.

Figure 6-6 shows how to convert a selection into a subVI.

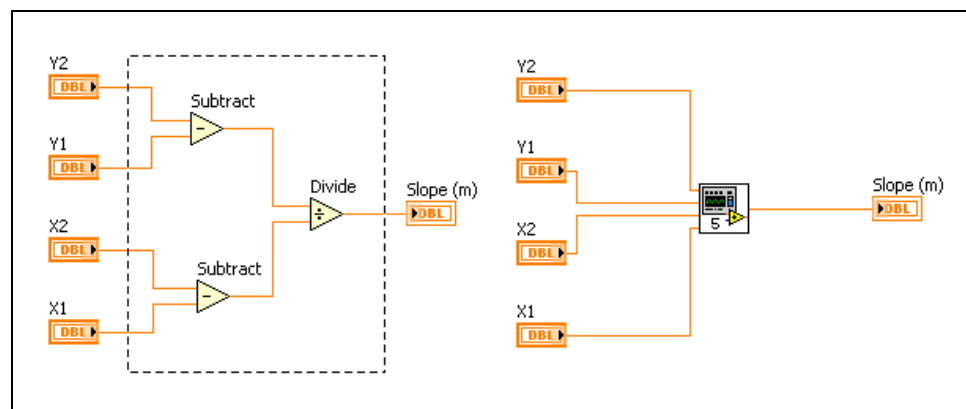


Figure 6-6. Creating a new SubVI

The new subVI uses a default pattern for the connector pane and a default icon. Double-click the subVI to edit the connector pane and icon, and to save the subVI.



Note Do not select more than 28 objects to create a subVI because 28 is the maximum number of connections on a connector pane. If your front panel contains more than 28 controls and indicators that you want to use programmatically, group some of them into a cluster and assign the cluster to a terminal on the connector pane.

Exercise 6-1 Project: Determine Warnings VI

Goal

Create the icon and connector pane for a VI so that the VI can be used as a subVI.

Scenario

You have created a VI that determines a warning string based on the inputs given. Create an icon and a connector pane so that this VI can be used as a subVI in the Weather Station project.

Design

The SubVI contains the following inputs and outputs:

Table 6-1. Determine Warnings SubVI inputs and outputs

Inputs	Outputs
Current Temp	Warning Text
Max Temp	Warning?
Min Temp	



Use the standard connector pane, shown at left, to assure room for future expansion. Add an error input and error output to the VI so that the code runs if there is no error, but does not run if there is an error.

Implementation

1. Open the Determine Warnings VI.
 - Select **File»Open Project**.
 - Open the `Weather Station.lvproj` in the `C:\Exercises\LabVIEW_Basics_I\Course Project` directory.
 - Double-click the `Determine Warnings.vi` in the **Project Explorer** window to open the VI.
2. Add an error input and an error output to the VI.
 - Place an `Error In 3D.ctl` on the front panel.
 - Place an `Error Out 3D.ctl` on the front panel.
3. Select a connector pane pattern for the VI.
 - Right-click the icon in the upper-right corner of the window and select **Show Connector** from the shortcut menu.
 - Right-click the icon in the upper-right corner of the window and select **Patterns**, from the shortcut menu and choose the pattern shown at left.
4. Connect the inputs and outputs to the connector as shown in Figure 6-7.

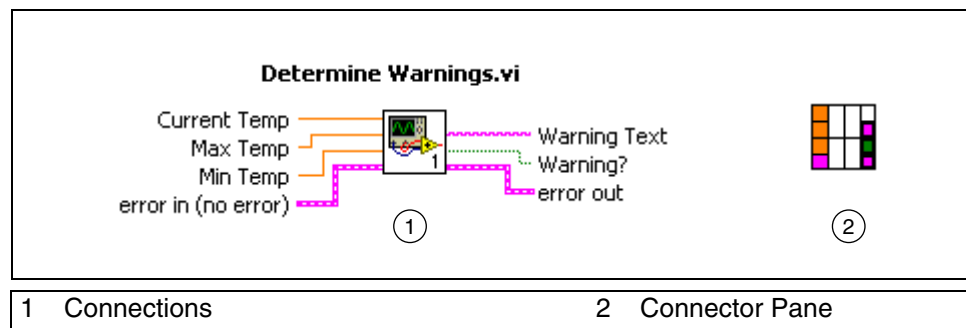


Figure 6-7. Connector Pane Connections for Determine Warnings VI

- Using the wiring tool, click the upper-left terminal of the connector pane.
- Click the corresponding front panel control, **Current Temp**.

Notice that the connector pane terminal fills in with a color to match the data type of the control connected to it.

- Click the next terminal in the connector pane.
 - Click the corresponding front panel control, **Max Temp**.
 - Continue wiring the connector pane until all controls and indicators are wired, and the **Context Help** window matches that shown in Figure 6-7.
5. Create an icon.
- Right-click the connector pane and select **Edit Icon**. The Icon Editor window opens.
 - Use the tools in the Icon Editor window to create an icon. Make the icon as simple or as complex as you want, however, it should be representative of the function of the VI. Figure 6-8 shows a simple example of an icon for this VI.



Figure 6-8. Sample Warning Icon

- Click **OK** when you are finished to close the Icon Editor window.



Tip Double-click the selection tool to select the existing graphic. Press <Delete> to delete the graphic.



Tip Double-click the text tool to modify fonts. You can select **Small Fonts** to choose fonts smaller than 9 points in size.

6. Right-click the connector pane and select **Show Icon** from the shortcut menu to return to Icon view.
7. Save the VI.
8. Switch to the block diagram.

- Set the VI to execute if no error occurs, and not execute if an error occurs.

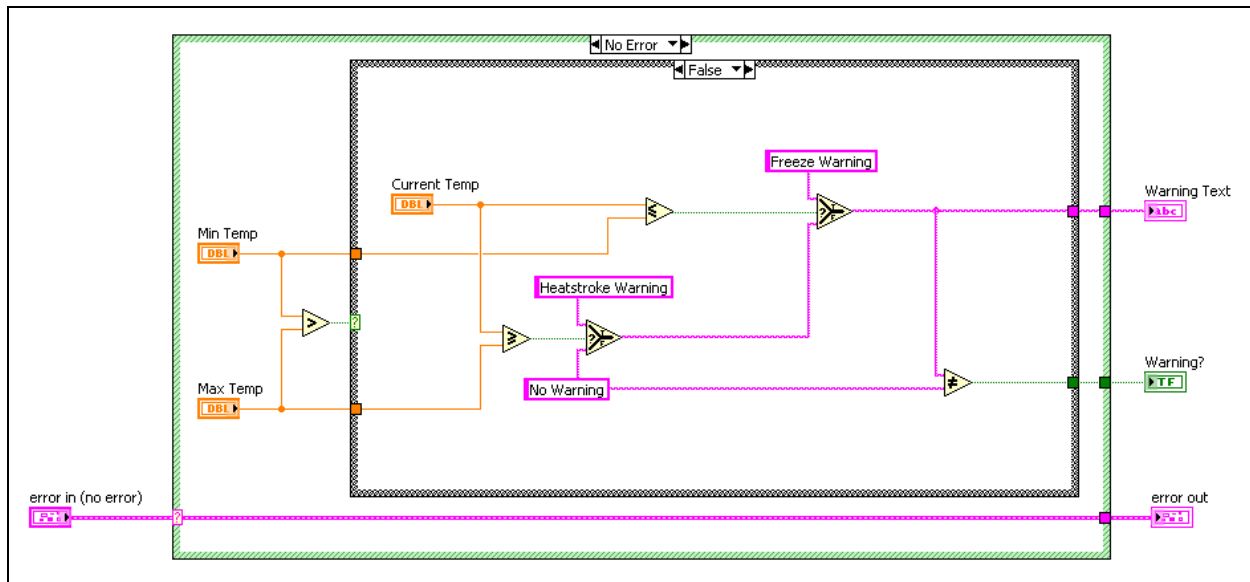


Figure 6-9. No Error Case of Determine Warnings VI

- Surround the block diagram code with a Case structure as shown in Figure 6-9. Leave the Warning Text and Warning? indicators outside of the Case structure.
- Place the **error in** terminal to the left of the Case structure.
- Place the **error out** terminal to the right of the Case structure.
- Wire the **error in** terminal to the case selector terminal.
- Confirm that the block diagram is in the No Error case. If it is not, switch to the case containing the code, right-click the Case structure and select **Make this Case No Error** from the shortcut menu.
- Wire the error cluster through the Case structure to the **error out** terminal as shown in Figure 6-9.
- Switch to the Error case.
- Wire the error cluster through the case to the **error out** tunnel.
- Right-click the Warning? tunnel and select **Create»Constant** from the shortcut menu.
- Use the Operating tool to change the constant to True.

- Right-click the Warning Text tunnel and select **Create»Constant** from the shortcut menu.
- Enter `Error` in the constant.
- Confirm that you have completed the Error case as shown in Figure 6-10.

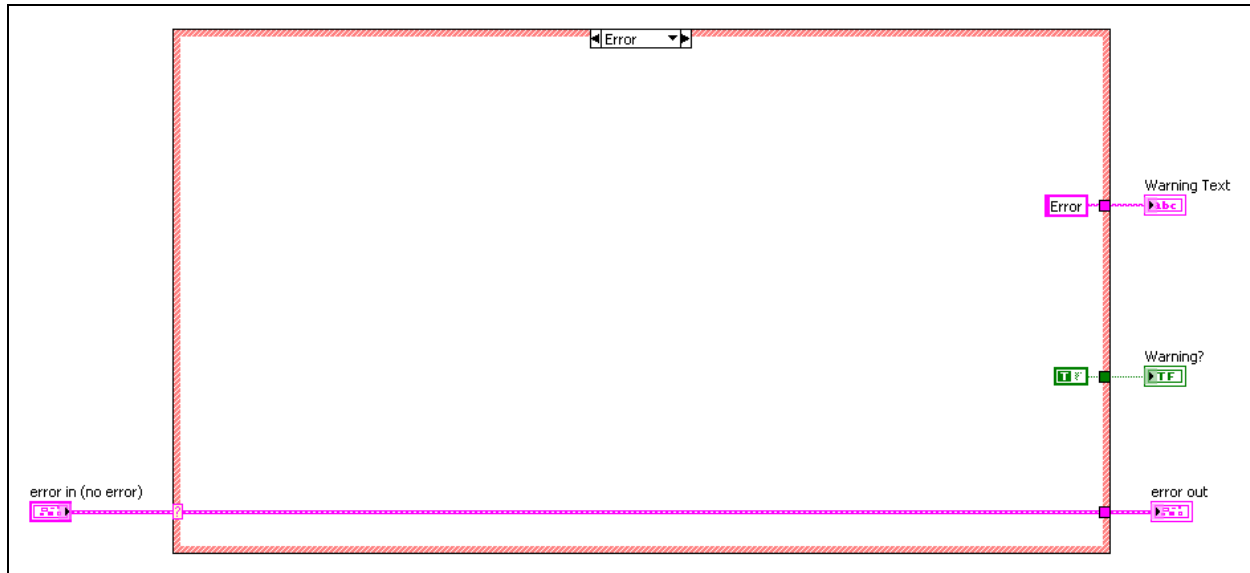


Figure 6-10. Error Case of Determine Warnings VI

If an error enters the VI, the VI outputs `Error` in `Warning Text`, and `True` in `Warning?` and passes out the error. If an error does not enter the VI, the VI operates as originally designed.

10. Save and close the VI.

Testing

Use a blank VI to test the subVI.

1. Open a blank VI.
2. Open the block diagram.
3. Place the subVI on the block diagram of the blank test VI.
 - Switch to the **Project Explorer** window.
 - Select the `Determine Warnings.vi` in the **Project Explorer** window.

- Drag and drop the `Determine Warnings.vi` to the block diagram of the blank test VI.
4. Create controls and indicators for each item in the subVI.
 - Right-click the **Current Temp** terminal and select **Create»Control** from the shortcut menu.
 - Right-click the **Max Temp** terminal and select **Create»Control** from the shortcut menu.
 - Right-click the **Min Temp** terminal and select **Create»Control** from the shortcut menu.
 - Right-click the **Warning Text** terminal and select **Create»Indicator** from the shortcut menu.
 - Right-click the **Warning?** terminal and select **Create»Indicator** from the shortcut menu.
 5. Switch to the front panel.
 6. Enter test values in Current Temp, Max Temp and Min Temp.
 7. Run the VI.
 8. After you have finished testing, close the test VI. You do not need to save the test VI.

End of Exercise 6-1

Self-Review: Quiz

1. On a subVI, which setting causes an error if the terminal is not wired?
 - a. Required
 - b. Recommended
 - c. Optional
2. You must create an icon to use a VI as a subVI.
 - a. True
 - b. False

Self-Review: Quiz Answers

1. On a subVI, which setting causes an error if the terminal is not wired?
 - a. **Required**
 - b. Recommended
 - c. Optional
2. You must create an icon to use a VI as a subVI.
 - a. True
 - b. **False: you should customize the icon, but the default icon is enough for functionality.**

Notes

Measurement Fundamentals

This lesson explains concepts that are critical to acquiring and generating signals effectively. These concepts focus on understanding the parts of your measurement system that exist outside of the computer. You will learn about transducers, signal sources, signal conditioning, grounding of your measurement system, and ways to increase the quality of your measurement acquisition. This lesson provides basic understanding of these concepts.

Topics

- A. Computer-Based Measurement Systems
- B. Measurement Concepts
- C. Increasing Measurement Quality

A. Computer-Based Measurement Systems

The fundamental task of all measurement systems is the measurement and/or generation of real-world physical signals. Measurement devices help you acquire, analyze, and present the measurements you take.

An example of a measurement system is shown in Figure 7-1. Before a computer-based measurement system can measure a physical signal, such as temperature, a sensor or transducer must convert the physical signal into an electrical one, such as voltage or current. You may need to condition the electrical signal before being measuring the signal. Signal conditioning may include filtering to remove noise or applying gain/attenuation to the signal to bring it into an acceptable measurement range. After conditioning the signal, measure the signal and then the measurement is communicated to the computer.

This course teaches two different methods of communicating the measured electrical signal to the computer: with a data acquisition (DAQ) board or with a stand-alone instrument (instrument control). Software controls the overall system, which includes acquiring the raw data, analyzing the data, and presenting the results. With these building blocks, you can obtain the physical phenomenon you want to measure into the computer for analysis and presentation.

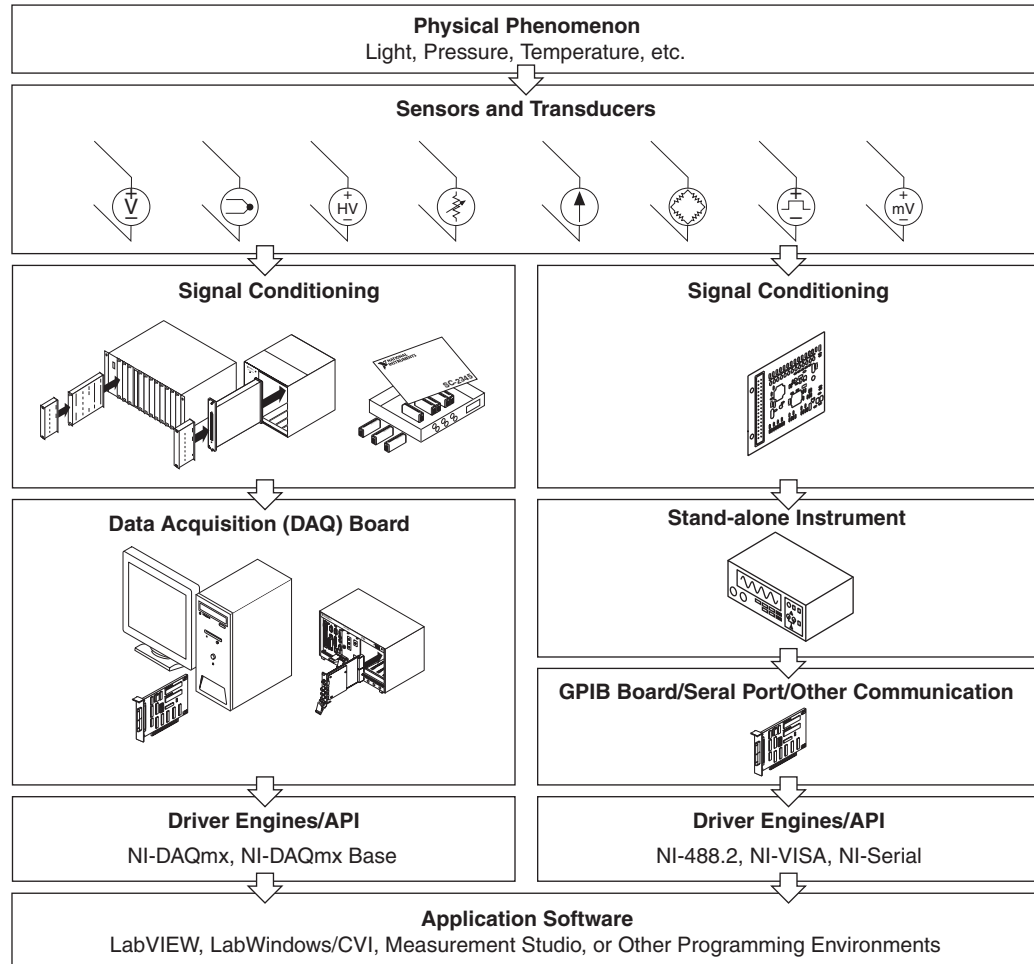


Figure 7-1. Measurement System Overview

B. Measurement Concepts

This section introduces you to concepts you should be familiar with before taking measurements with DAQ device and instruments.

Signal Acquisition

Signal acquisition is the process of converting physical phenomena into data the computer can use. A measurement starts with using a transducer to convert a physical phenomenon into an electrical signal. Transducers can generate electrical signals to measure such things as temperature, force, sound, or light. Table 7-1 lists some common transducers.

Table 7-1. Phenomena and Transducers

Phenomena	Transducer
Temperature	Thermocouples Resistance temperature detectors (RTDs) Thermistors Integrated circuit sensors
Light	Vacuum tube photosensors Photoconductive cells
Sound	Microphones
Force and pressure	Strain gages Piezoelectric transducers Load cells
Position (displacement)	Potentiometers Linear voltage differential transformers (LVDT) Optical encoders
Fluid flow	Head meters Rotational flowmeters Ultrasonic flowmeters
pH	pH electrodes

Signal Sources

Analog input acquisitions use grounded and floating signal sources.

Grounded Signal Sources

A grounded signal source is one in which the voltage signals are referenced to a system ground, such as the earth or a building ground, as shown in Figure 7-2. Because such sources use the system ground, they share a common ground with the measurement device. The most common examples

of grounded sources are devices that plug into a building ground through wall outlets, such as signal generators and power supplies.

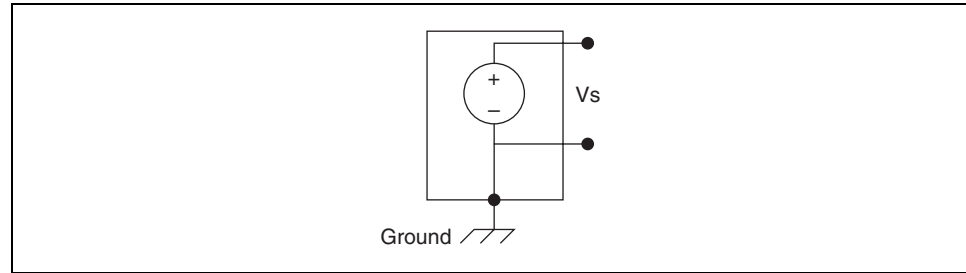


Figure 7-2. Grounded Signal Source



Note The grounds of two independently grounded signal sources generally are not at the same potential. The difference in ground potential between two instruments connected to the same building ground system is typically 10 mV to 200 mV. The difference can be higher if power distribution circuits are not properly connected. This causes a phenomenon known as a ground loop.

Floating Signal Sources

In a floating signal source, the voltage signal is not referenced to any common ground, such as the earth or a building ground, as shown in Figure 7-3. Some common examples of floating signal sources are batteries, thermocouples, transformers, and isolation amplifiers. Notice in Figure 7-3 that neither terminal of the source is connected to the electrical outlet ground as in Figure 7-2. Each terminal is independent of the system ground.

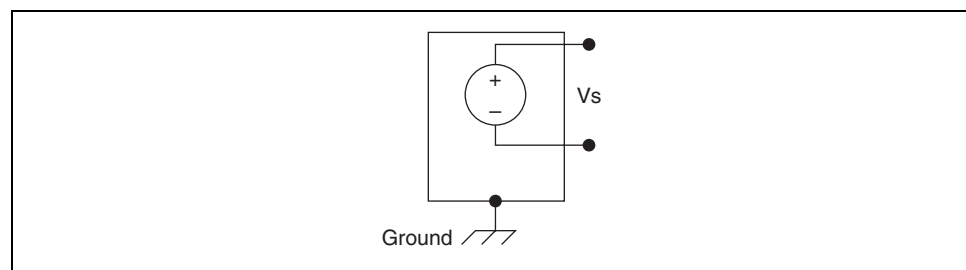


Figure 7-3. Floating Signal Source

Signal Conditioning

Signal conditioning is the process of measuring and manipulating signals to improve accuracy, isolation, filtering, and so on. Many stand-alone instruments and DAQ devices have built-in signal conditioning. Signal conditioning also can be applied externally, by designing a circuit to condition the signal or by using devices specifically made for signal conditioning. National Instruments has SCXI devices and other devices that are designed for this purpose. Throughout this section, different DAQ and SCXI devices illustrate signal conditioning topics.

To measure signals from transducers, you must convert them into a form a measurement device can accept. For example, the output voltage of most thermocouples is very small and susceptible to noise. Therefore, you might need to amplify the thermocouple output before you digitize it. This amplification is a form of signal conditioning. Common types of signal conditioning include amplification, linearization, transducer excitation, and isolation.

Figure 7-4 shows some common types of transducers and signals and the signal conditioning each requires.

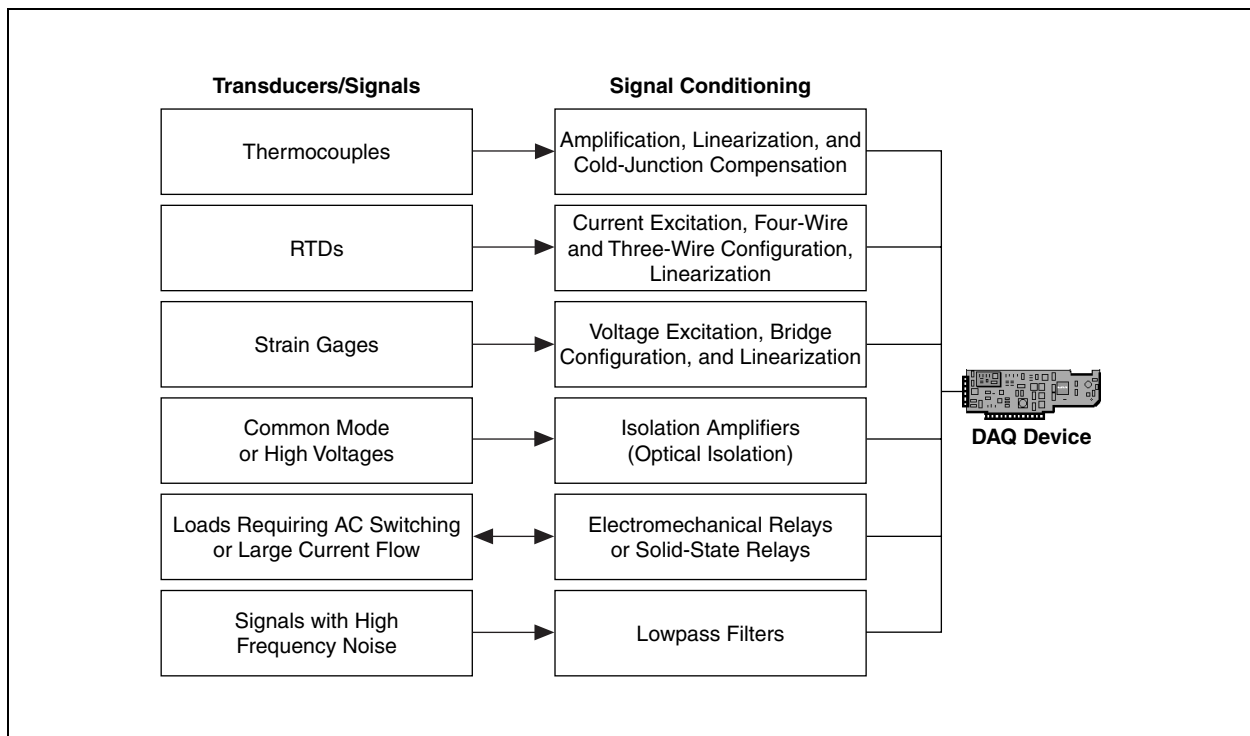


Figure 7-4. Common Transducers and Signal Conditioning Types

Amplification

Amplification is the most common type of signal conditioning. Amplifying electrical signals improves accuracy in the resulting digitized signal and reduces the effects of noise.

Signals should be amplified as close to the signal source as possible. By amplifying a signal near the device, any noise that attached to the signal is also amplified. Amplifying near the signal source result in the largest signal-to-noise ratio (SNR). For the highest possible accuracy, amplify the signal so the maximum voltage range equals the maximum input range of the analog-to-digital converter (ADC).

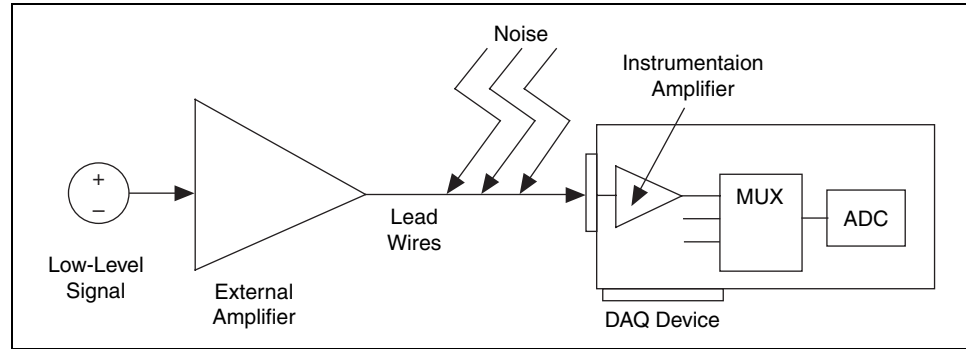


Figure 7-5. Signal Amplification

If you amplify the signal at the DAQ device while digitizing and measuring the signal, noise might have entered the lead wire, which decreases SNR. However, if you amplify the signal close to the signal source with a SCXI module, noise has a less destructive effect on the signal, and the digitized representation is a better reflection of the original low-level signal. Refer to the National Instruments Web site at ni.com/info and enter the info code `exd2hc` for more information about analog signals.

Linearization

Many transducers, such as thermocouples, have a nonlinear response to changes in the physical phenomena you measure. LabVIEW can linearize the voltage levels from transducers so you can scale the voltages to the measured phenomena. LabVIEW provides scaling functions to convert voltages from strain gages, RTDs, thermocouples, and thermistors.

Transducer Excitation

Signal conditioning systems can generate excitation, which some transducers require for operation. Strain gages and RTDs require external voltage and currents, respectively, to excite their circuitry into measuring physical phenomena. This type of excitation is similar to a radio that needs power to receive and decode audio signals.

Isolation

Another common way to use signal conditioning is to isolate the transducer signals from the computer for safety purposes.



Caution When the signal you monitor contains large voltage spikes that could damage the computer or harm the operator, do not connect the signal directly to a DAQ device without some type of isolation.

You also can use isolation to ensure that differences in ground potentials do not affect measurements from the DAQ device. When you do not reference the DAQ device and the signal to the same ground potential, a ground loop

can occur. Ground loops can cause an inaccurate representation of the measured signal. If the potential difference between the signal ground and the DAQ device ground is large, damage can occur to the measuring system. Isolating the signal eliminates the ground loop and ensures that the signals are accurately measured.

Measurement Systems

You configure a measurement system based on the hardware you use and the measurement you take.

Differential Measurement Systems

Differential measurement systems are similar to floating signal sources in that you make the measurement with respect to a floating ground that is different from the measurement system ground. Neither of the inputs of a differential measurement system are tied to a fixed reference, such as the earth or a building ground. Handheld, battery-powered instruments and DAQ devices with instrumentation amplifiers are examples of differential measurement systems.

A typical National Instruments device uses an implementation of an eight-channel differential measurement systems as shown in Figure 7-6. Using analog multiplexers in the signal path increases the number of measurement channels when only one instrumentation amplifier exists. In Figure 7-6, the AIGND (analog input ground) pin is the measurement system ground.

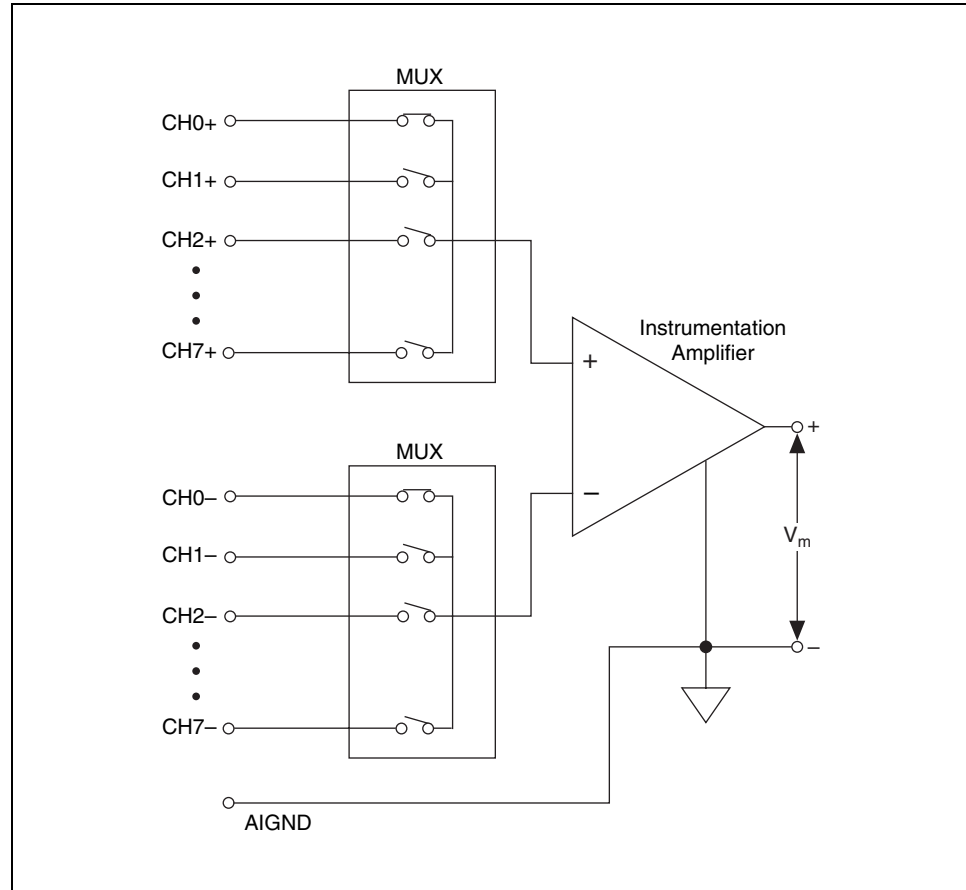


Figure 7-6. Typical Differential Measurement System

Referenced and Non-Referenced Single Ended

Referenced and non-referenced single-ended measurement systems are similar to grounded sources in that you make the measurement with respect to a ground. A referenced single-ended measurement system measures voltage with respect to the ground, AIGND, which is directly connected to the measurement system ground. Figure 7-7 shows a 16-channel referenced single-ended measurement system.

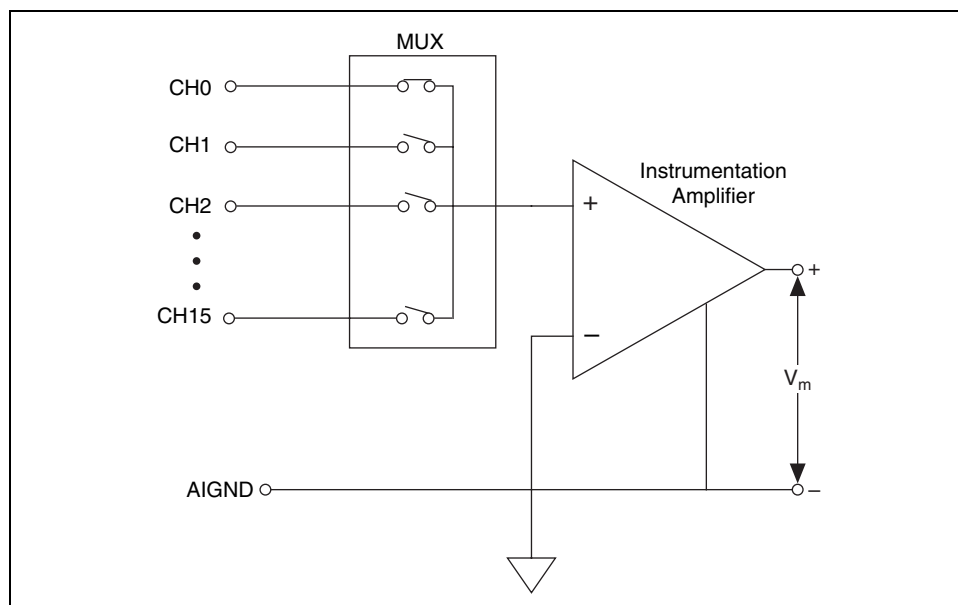


Figure 7-7. Typical Referenced Single-Ended (RSE) Measurement System

DAQ devices often use a non-referenced single-ended (NRSE) measurement technique, or pseudodifferential measurement, which is a variant of the referenced single-ended measurement technique. Figure 7-8 shows a NRSE system.

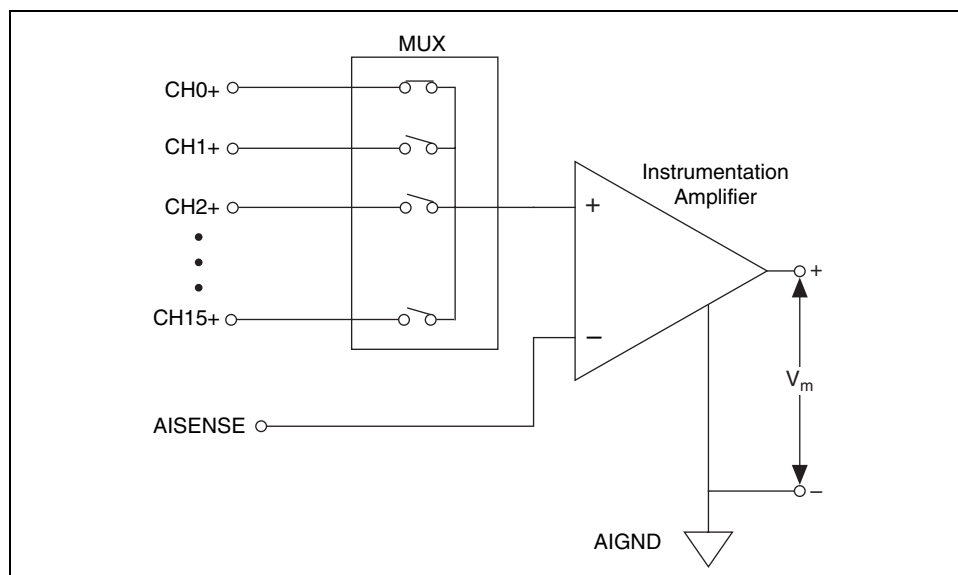


Figure 7-8. Typical Non-Referenced Single-Ended (NRSE) Measurement System

In a NRSE measurement system, all measurements are still made with respect to a single-node analog input sense (AISENSE on E Series devices), but the potential at this node can vary with respect to the measurement system ground (AIGND). A single-channel NRSE measurement system is the same as a single-channel differential measurement system.

Signal Sources and Measurement Systems Summary

Figure 7-9 summarizes ways to connect a signal source to a measurement system.

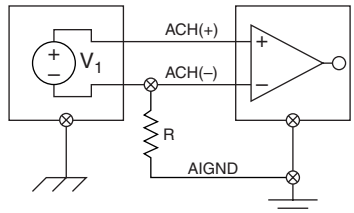
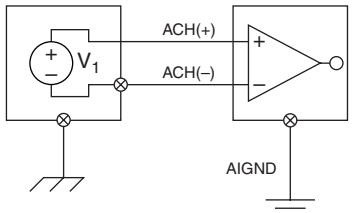
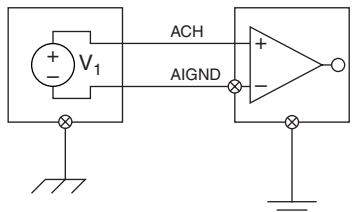
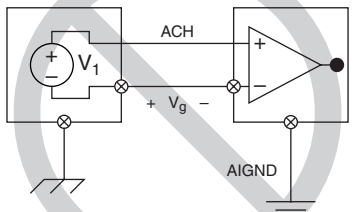
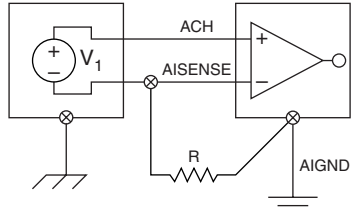
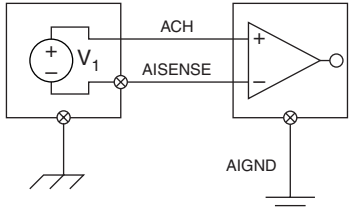
Input	Signal Source Type	
	Floating Signal Source (Not Connected to Building Ground)	Grounded Signal Source
	Examples <ul style="list-style-type: none"> • Ungrounded Thermocouples • Signal Conditioning with Isolated Outputs • Battery Devices 	Examples <ul style="list-style-type: none"> • Plug-in Instruments with Nonisolated Outputs
Differential (DIFF)	 <p>See text for information on bias resistors.</p>	
Single-Ended — Ground Referenced (RSE)		<p style="text-align: center;">NOT RECOMMENDED</p>  <p>Ground-loop losses, V_g, are added to measured signal.</p>
Single-Ended — Nonreferenced (NRSE)	 <p>See text for information on bias resistors.</p>	

Figure 7-9. Signal Source and Measurement Systems Summary

C. Increasing Measurement Quality

When you design a measurement system, you may find that the measurement quality does not meet your expectations. You might want to record the smallest possible change in a voltage level. Perhaps you cannot tell if a signal is a triangle wave or a sawtooth wave and would like to see a better representation of the shape of a signal. Often, you want to reduce the noise in the signal. Methods for achieving these three increases in quality are introduced in this section.

Achieving Smallest Detectable Change

The following reasons affect achieving the smallest detectable change in voltage:

- The resolution and range of the ADC
- The gain applied by the instrumentation amplifier
- The combination of the resolution, range, and gain to calculate a property called the code width value

Resolution

The number of bits used to represent an analog signal determines the resolution of the ADC. The resolution on a DAQ device is similar to the marks on a ruler. The more marks a ruler has, the more precise the measurements are. The higher the resolution is on a DAQ device, the higher the number of divisions into which a system can break down the ADC range, and therefore, the smaller the detectable change. A 3-bit ADC divides the range into 2^3 or eight divisions. A binary or digital code between 000 and 111 represents each division. The ADC translates each measurement of the analog signal to one of the digital divisions. The following illustration shows a 5 kHz sine wave digital image obtained by a 3-bit ADC. The digital signal does not represent the original signal adequately because the converter has too few digital divisions to represent the varying voltages of the analog signal. However, increasing the resolution to 16 bits to increase the ADC number of divisions from eight (2^3) to 65,536 (2^{16}) allows the 16-bit ADC to obtain an extremely accurate representation of the analog signal.

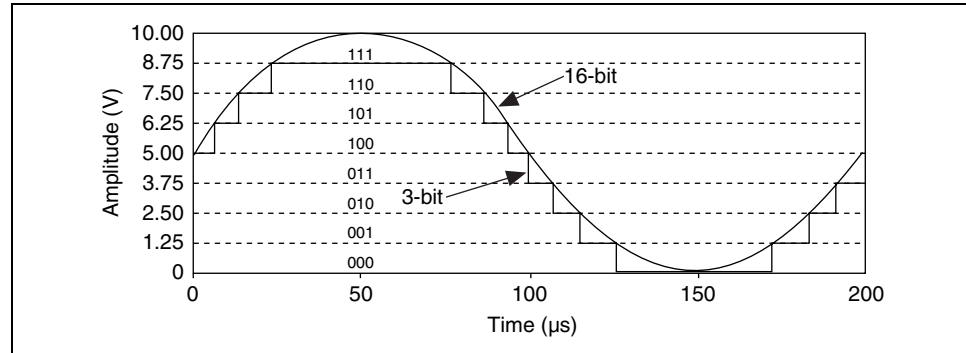


Figure 7-10. 3-Bit and 16-Bit Resolution Example

Device Range

Range refers to the minimum and maximum analog signal levels that the ADC can digitize. Many DAQ devices feature selectable ranges (typically 0 to 10 V or -10 to 10 V), so you can match the ADC range to that of the signal to take best advantage of the available resolution to accurately measure the signal. For example, in the following illustration, the 3-bit ADC in chart 1 has eight digital divisions in the range from 0 to 10 V, which is a unipolar range. If you select a range of -10 to 10 V, which is a bipolar range, as shown in chart 2, the same ADC separates a 20 V range into eight divisions. The smallest detectable voltage increases from 1.25 to 2.50 V, and the right chart is a much less accurate representation of the signal.

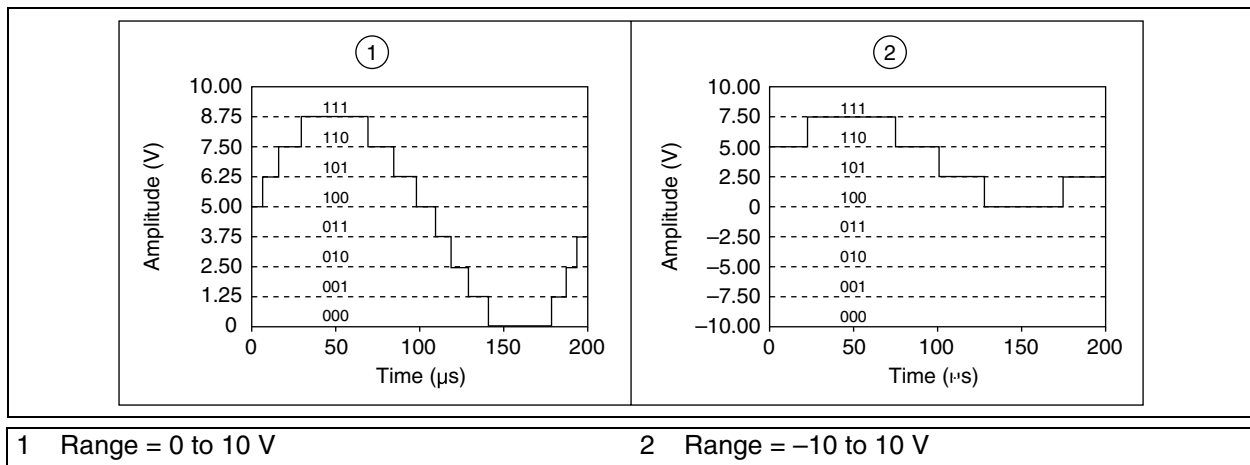


Figure 7-11. Range Example

Amplification

Amplification or attenuation of a signal can occur before the signal is digitized to improve the representation of the signal. By amplifying or attenuating a signal, you can effectively decrease the input range of an ADC and thus allow the ADC to use as many of the available digital divisions as possible to represent the signal.

For example, Figure 7-12 shows the effects of applying amplification to a signal that fluctuates between 0 and 5 V using a 3-bit ADC and a range of 0 to 10 V. With no amplification, or gain = 1, the ADC uses only four of the eight divisions in the conversion. By amplifying the signal by two before digitizing, the ADC uses all eight digital divisions, and the digital representation is much more accurate. Effectively, the device has an allowable input range of 0 to 5 V because any signal above 5 V when amplified by a factor of two makes the input to the ADC greater than 10 V.

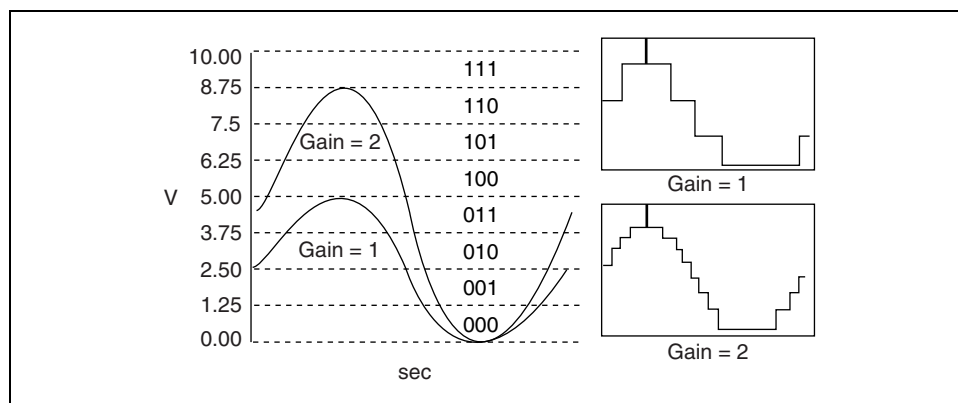


Figure 7-12. Amplification Example

The range, resolution, and amplification available on a DAQ device determine the smallest detectable change in the input voltage. This change in voltage represents one least significant bit (LSB) of the digital value and is also called the code width.

Code Width

Code width is the smallest change in a signal that a system can detect. Code width is calculated using the following formula where C is code width, D is device input range, and R is bits of resolution:

$$C = D \cdot \frac{1}{(2^R)}$$

Device input range is a combination of the gain applied to the signal and the input range of the ADC. For example, if the ADC input range is -10 to $+10$ V peak to peak and the gain is 2, the device input range is -5 to $+5$ V peak to peak, or 20 V.

The smaller the code width, the more accurately a device can represent the signal. The formula confirms what you have already learned in the discussion on resolution, range, and gain:

- Larger resolution = smaller code width = more accurate representation of the signal

- Larger amplification = smaller code width = more accurate representation of the signal
- Larger range = larger code width = less accurate representation of the signal

Determining the code width is important in selecting a DAQ device. For example, a 12-bit DAQ device with a 0 to 10 V input range and an amplification of one detects a 2.4 mV change, while the same device with a –10 to 10 V input range would detect a change of 4.8 mV.

$$C = D \cdot \frac{1}{(2^R)} = 10 \cdot \frac{1}{(2^{12})} = 2.4mV$$

$$C = D \cdot \frac{1}{(2^R)} = 20 \cdot \frac{1}{(2^{12})} = 4.8mV$$

Increasing Shape Recovery

The most effective way of increasing the shape recovery of a signal is to reduce your code width and increase your sampling frequency. To measure the frequency of your signal effectively, you must sample the signal at least the Nyquist frequency.

The following states the Nyquist Theorem:

$$f_{sampling} > 2 \cdot f_{signal}$$

Where $f_{sampling}$ is the sampling rate, and f_{signal} is the highest frequency component of interest in the measured signal.

The Nyquist Theorem states that you must sample a signal at a rate greater than twice the highest frequency component of interest in the signal to capture the highest frequency component of interest. Otherwise, the high-frequency content aliases at a frequency inside the spectrum of interest, called the pass-band.

To determine how fast to sample, refer to Figure 7-13 which shows the effects of various sampling rates. In case A, the sine wave of frequency f is sampled at the same frequency f . The reconstructed waveform appears as an alias at DC. However, if you increase the sampling rate to $2f$, the digitized

waveform has the correct frequency (same number of cycles) but appears as a triangle waveform. In this case f is equal to the Nyquist frequency. By increasing the sampling rate to well above f , for example $5f$, you can more accurately reproduce the waveform. In case C, the sampling rate is at

$$\frac{4f}{3}$$

The Nyquist frequency in this case is

$$\frac{(4f)/3}{2} = \frac{2f}{3}$$

Because f is larger than the Nyquist frequency, this sampling rate reproduces an alias waveform of incorrect frequency and shape.

The faster the sample, the better the shape recovery of the signal. However, available hardware often limits the sampling rate.

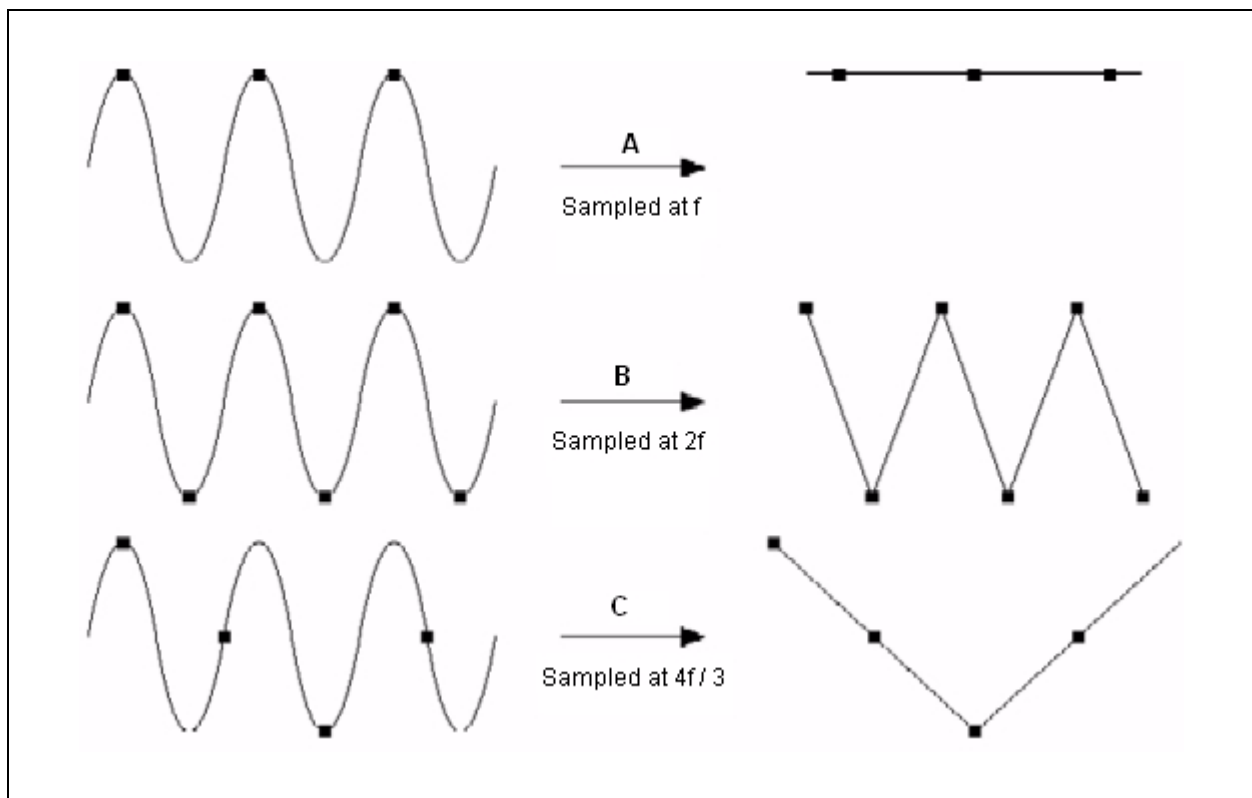


Figure 7-13. Effects of various sampling rates while sampling a signal

Decreasing Noise

To reduce noise take the following precautions:

- Use shielded cables or a twisted pair of cables.
- Minimize wire length to minimize noise the lead wires pick up.
- Keep signal wires away from AC power cables and monitors to reduce 50 or 60 Hz noise.
- Reduce the signal-to-noise (SNR) ratio by amplifying the signal close to the signal source.
- Acquire data at a higher frequency than necessary, then average the data to reduce the impact of the noise, as noise tends to average to zero.

Exercise 7-1 Concepts: Measurement Fundamentals

Goal

Understand how resolution, voltage range, gain, and aliasing affect a measured signal.

Description

1. Open `Resolution.vi` in the `C:\Exercises\LabVIEW Basics I\Measurement Fundamentals` directory.

This VI simulates the acquisition of a sine wave and the digitization that occurs with an analog to digital convertor (ADC). This VI contains the following controls and indicators:

- **Input Signal Voltage:** This input specifies the range of the signal being acquired. The default value of the control is ± 1 Volt. This means that the range of the signal is 2 V: voltage between the highest point of the signal and the lowest point of the signal.
- **Resolution (ADC):** This input specifies the resolution of the ADC of the data acquisition board used to acquire the signal. The default value of the control is 3 bits.
- **Device Input Range:** The input incorporates the input range of the ADC and the gain applied to the signal. The default value of the control is ± 1 Volts. This peak to peak voltage is equivalent to 2 volts. Since the input range of the ADC is ± 10 V, this means that there is a gain of 10 applied to the signal.
- **Code Width:** This output calculates the code width using the current values of the controls, where C is code width, D is device input range, and R is bits of resolution:

$$C = D \cdot \frac{1}{(2^R)} = 2 \cdot \frac{1}{(2^3)} = 0.25 V$$

2. Run the VI and experiment with the values of the controls.

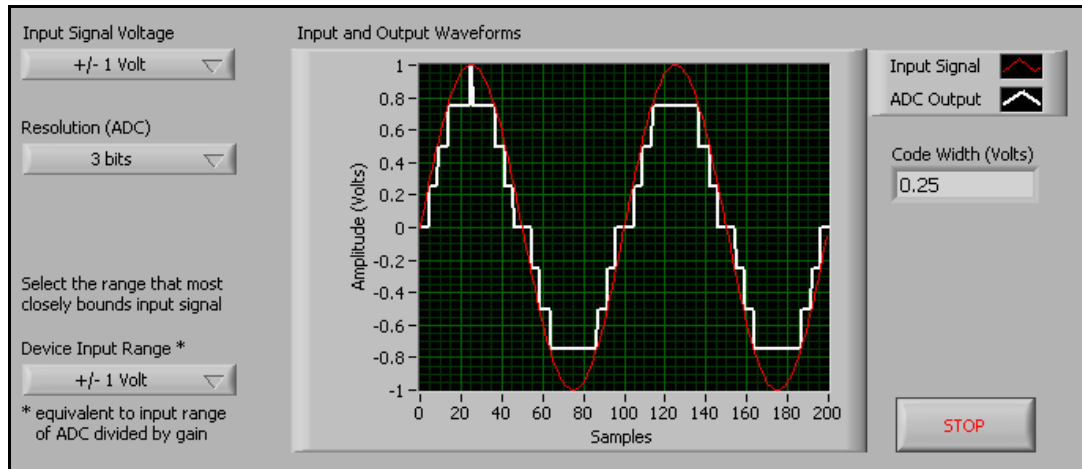


Figure 7-14. Resolution VI Front Panel

- Click the **Run** button to run the VI.
- Leave the default settings for the controls.

The red plot demonstrates the actual input sinewave. The white plot demonstrates the output of the ADC. Notice that the white plot is a poor representation of the signal. You can see the code width of .25 volts shown on the graph representing only 8 discrete levels.

- Change the Resolution (ADC).

Notice that the signal representation quality increases as a larger ADC resolution is chosen.

- Set the resolution to 3 bits.
- Change the Device Input Range.

Notice that when too large of an input range is chosen, the resolution is not efficiently divided among the signals range. When too small of an input range is chosen, part of the signal is cut off.

- Experiment further with different control values until you understand the importance of each input.

It is important to ensure that the input signal range is as close to the device input range as possible. In the next lesson, you learn to set the device input range in software.

- Using the Resolution VI, determine the code width of an input signal that varies between ± 0.8 V using a data acquisition board with a resolution of 16 bits. Assume that gain is efficiently applied.

Code Width:

- Determine the code width of an input signal that varies between ± 10 V using a data acquisition board with a resolution of 8 bits. The device input range is set to ± 10 V.

Code Width:

- If the device input range is ± 1 V, and the resolution is 12 bits, what is the largest input signal you can read without cutting off the input signal?

Input Signal Range:

- Stop and close the VI when you are finished.
- Open `Aliasing.vi` in the `C:\Exercises\LabVIEW Basics I\Measurement Fundamentals` directory.

This VI simulates the acquisition of a waveform at a specific sampling frequency. As you adjust the sampling frequency and the frequency of the acquired waveform, you can observe the Nyquist Theorem in effect. This VI contains the following controls:

- Original Signal:**
 - Frequency:** This input specifies the frequency of the signal being acquired. You can increase or decrease this frequency by turning the knob.
 - Sampled Waveform:** The input allows you to choose between a sine wave or a square wave. Use the sine wave input to experiment with the Nyquist Theorem, and the square wave to understand how the sampling frequency affects shape recovery.
- Sampled Signal:**
 - Sampling Rate (Hz):** This input specifies the rate at which the data acquisition board takes a sample of the acquired signal. According to the Nyquist Theorem, this rate should be at least twice the frequency of the sampled signal.

8. Run the VI and experiment with the values of the controls until the acquired frequency is wrong.

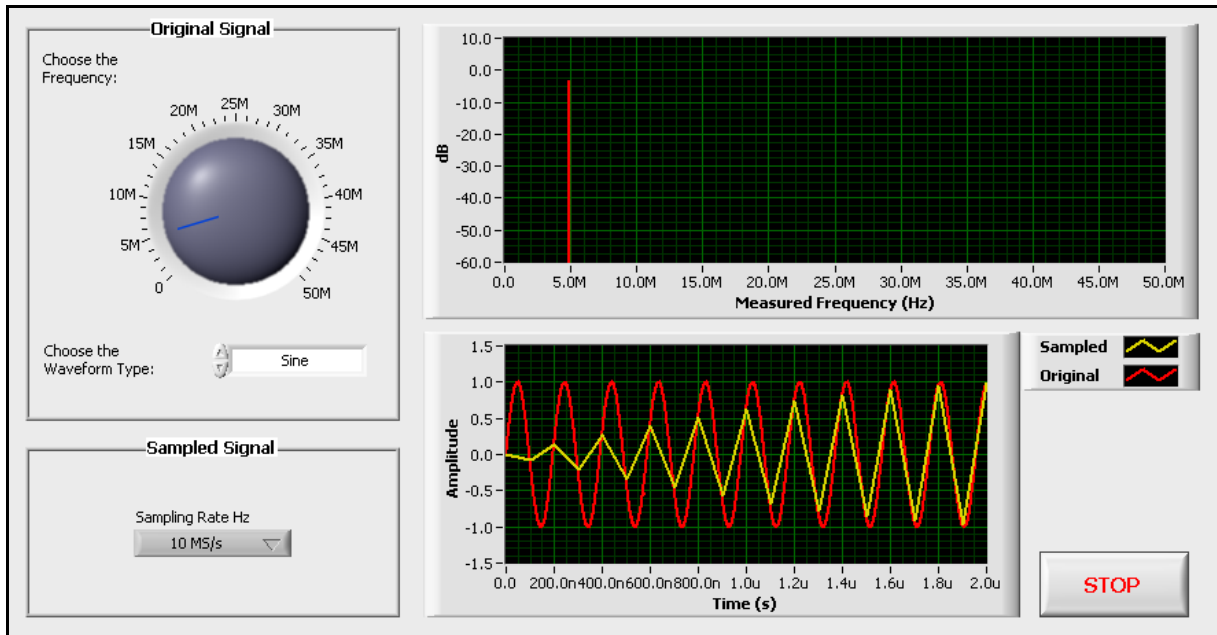


Figure 7-15. Aliasing VI Front Panel

- Set the Sampled Waveform to **Sine**.
 - Set the Sampling Rate Hz to **10MS/s** (megasamples per second).
 - Adjust the Frequency of the Sampled Signal, starting at the lowest frequency, and moving up until the frequency reported on the top chart is no longer correct. Notice how the Sampled plot becomes more distorted as you increase the Frequency of the Sampled Signal. After you have passed the Nyquist frequency (5 MHz in this case), the frequency recorded is no longer correct. This is an example of aliasing.
9. Try other values for the controls using a sine wave.
10. Set the Sampled Waveform to **Square**. Modify the controls to see how shape recovery is affected by the sampling frequency and the frequency of the signal.
11. Stop and close the VI when you are finished.

End of Exercise 7-1

Self-Review: Quiz

1. Calculate the code-width for signal acquired using a 16 bit data acquisition board with a device input range of 5 V.

2. You are acquiring a triangle wave with a frequency of 1100 Hz. Which sampling frequency should you use for best shape recovery of the signal?
 - a. 1 kHz
 - b. 10 kHz
 - c. 100 kHz
 - d. 1000 kHz

3. You are acquiring a triangle wave with a frequency of 1100 Hz. You can sample the signal at the following rates. Which is the minimum sampling frequency you should use to reliably acquire the frequency of the signal?
 - a. 1 kHz
 - b. 10 kHz
 - c. 100 kHz
 - d. 1000 kHz

Self-Review: Quiz Answers

1. Calculate the code-width for signal acquired using a 16 bit data acquisition board with a device input range of 5 V.

$$C = D \cdot \frac{1}{(2^R)} = \left(5 \cdot \frac{1}{(2^{16})}\right) = 76.29 \mu V$$

2. You are acquiring a triangle wave with a frequency of 1100 Hz. Which sampling frequency should you use for best shape recovery of the signal?
 - a. 1 kHz
 - b. 10 kHz
 - c. 100 kHz
 - d. **1000 kHz**
3. You are acquiring a triangle wave with a frequency of 1100 Hz. You can sample the signal at the following rates. Which is the minimum sampling frequency you should use to reliably acquire the frequency of the signal?
 - a. 1 kHz
 - b. **10 kHz**
 - c. 100 kHz
 - d. 1000 kHz

Notes

Data Acquisition

A data acquisition (DAQ) system uses a data acquisition board to pass a conditioned electrical signal to a computer for software analysis and data logging. You can choose a data acquisition board that uses a PCI bus, a PXI bus, or the computer USB or IEEE 1394 port. This lesson explains the hardware used in a data acquisition system, how to configure the devices and how to program analog input and output, counters, and digital input and output.

Topics

- A. Hardware
- B. Software Architecture
- C. Simulating a DAQ Device
- D. Analog Input
- E. Analog Output
- F. Counters
- G. Digital I/O

A. Hardware

A typical DAQ system has three basic types of hardware—a terminal block, a cable, and a DAQ device, as shown in Figure 8-1.

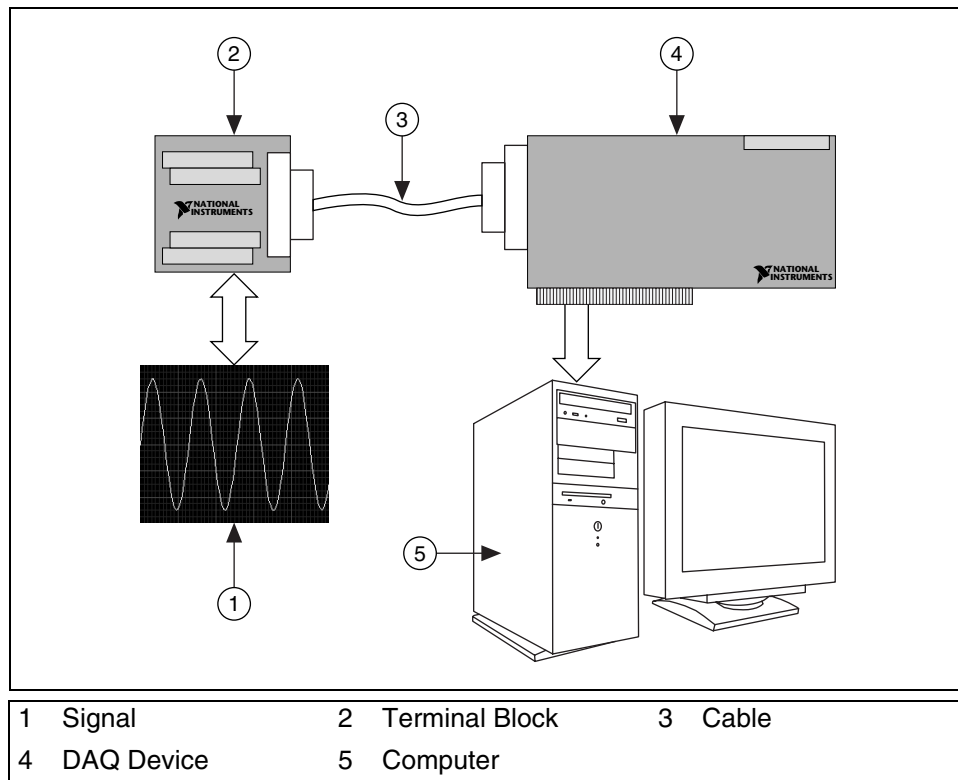


Figure 8-1. Typical DAQ System

After you have converted a physical phenomenon into a measurable signal with or without signal conditioning, you need to acquire that signal. To acquire a signal, you need a terminal block, a cable, a DAQ device, and a computer. This hardware combination can transform a standard computer into a measurement and automation system.

Terminal Block and Cable

A terminal block provides a place to connect signals. It consists of screw or spring terminals for connecting signals and a connector for attaching a cable to connect the terminal block to a DAQ device. Terminal blocks have 100, 68, or 50 terminals. The type of terminal block you should choose depends on two factors—the device and the number of signals you are measuring. A terminal block with 68 terminals offers more ground terminals to connect a signal to than a terminal block with 50 terminals. Having more ground terminals prevents the need to overlap wires to reach a ground terminal, which can cause interference between the signals.

Terminal blocks can be shielded or non-shielded. Shielded terminal blocks offer better protection against noise. Some terminal blocks contain extra features, such as cold-junction compensation, that are necessary to measure a thermocouple properly.

A cable transports the signal from the terminal block to the DAQ device. Cables come in 100-, 68-, and 50-pin configurations. Choose a configuration depending on the terminal block and the DAQ device you are using. Cables, like terminal blocks, are shielded or non-shielded.

Refer to the DAQ section of the National Instruments catalog or to ni.com/products for more information about specific types of terminal blocks and cables.

DAQ Signal Accessory

Figure 8-2 shows the terminal block you are using for this course, the DAQ Signal Accessory.

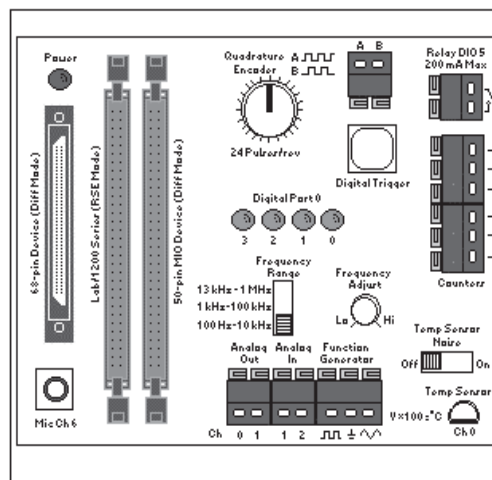


Figure 8-2. DAQ Signal Accessory

The DAQ Signal Accessory is a customized terminal block designed for learning purposes. It has three different cable connectors to accommodate many different DAQ devices and spring terminals to connect signals. You can access three analog input channels, one of which is connected to the temperature sensor, and two analog output channels.

The DAQ Signal Accessory includes a function generator with a switch to select the frequency range of the signal, and a frequency knob. The function generator can produce a sine wave or a square wave. A connection to ground is located between the sine wave and square wave terminal.

A digital trigger button produces a TTL pulse for triggering analog input or output. When you press the trigger button, the signal goes from +5 V to 0 V when pressed and returns to +5 V when you release the button. Four LEDs connect to the first four digital lines on the DAQ device. The LEDs use reverse logic, so when the digital line is high, the LED is off and vice versa.

The DAQ Signal Accessory has a quadrature encoder that produces two pulse trains when you turn the encoder knob. Terminals are provided for the input and output signals of two counters on the DAQ device. The DAQ Signal Accessory also has a relay, a thermocouple input, and a microphone jack.

DAQ Device

Most DAQ devices have four standard elements: analog input, analog output, digital I/O, and counters.

You can transfer the signal you measure with the DAQ device to the computer through a variety of different bus structures. For example, you can use a DAQ device that plugs into the PCI bus of a computer, a DAQ device connected to the PCMCIA socket of a laptop, or a DAQ device connected to the USB port of a computer. You also can use PXI/CompactPCI to create a portable, versatile, and rugged measurement system.

If you do not have a DAQ device, you can simulate one in Measurement and Automation Explorer to complete your software testing. You learn to simulate a device in the *Simulating a DAQ Device* section of this lesson.

Refer to the DAQ section of the NI catalog or to ni.com/products for more information about specific types of DAQ devices.

B. Software Architecture

National Instruments data acquisition boards have a driver engine that communicates between the board and the application software. There are two different driver engines to choose from: NI-DAQmx and Traditional NI-DAQ. You can use LabVIEW to communicate with these driver engines. You have already used the DAQ Assistant in LabVIEW to communicate with your data acquisition board. The DAQ Assistant is an Express VI that communicates with NI-DAQmx.

In addition, National Instruments has an application that is useful for configuring your data acquisition boards: Measurement & Automation Explorer (MAX). In this section, you learn about the driver engines and about using MAX to configure your data acquisition board.

NI-DAQ

NI-DAQ 7.0 contains two NI-DAQ drivers—Traditional NI-DAQ (Legacy) and NI-DAQmx—each with its own application programming interface (API), hardware configuration, and software configuration.

- Traditional NI-DAQ (Legacy) is an upgrade to NI-DAQ 6.9.x, the earlier version of NI-DAQ. Traditional NI-DAQ (Legacy) has the same VIs and functions and works the same way as NI-DAQ 6.9.x. You can use Traditional NI-DAQ (Legacy) on the same computer as NI-DAQmx, which you cannot do with NI-DAQ 6.9.x.
- NI-DAQmx is the latest NI-DAQ driver with new VIs, functions, and development tools for controlling measurement devices. The advantages of NI-DAQmx over previous versions of NI-DAQ include the DAQ Assistant for configuring channels and measurement tasks for a device; increased performance, including faster single-point analog I/O and multithreading; and a simpler API for creating DAQ applications using fewer functions and VIs than earlier versions of NI-DAQ.

Traditional NI-DAQ (Legacy) and NI-DAQmx support different sets of devices. Refer to the National Instruments Web site at ni.com/daq for the list of supported devices.

This lesson describes the NI-DAQmx API.

When programming an NI measurement device, you can use NI application software such as LabVIEW, LabWindows™/CVI™, and Measurement Studio, or open ADEs that support calling dynamic link libraries (DLLs) through ANSI C interfaces. Using NI application software greatly reduces development time for data acquisition and control applications regardless of which programming environment you use:

- LabVIEW supports data acquisition with the LabVIEW DAQ VIs, a series of VIs for programming with NI measurement devices.
- For C developers, LabWindows/CVI is a fully integrated ANSI C environment that provides the LabWindows/CVI Data Acquisition library for programming NI measurement devices.
- Measurement Studio development tools are for designing your test and measurement software in Microsoft Visual Studio .NET. Measurement Studio includes tools for Visual C#, Visual Basic .NET, and Visual C++ .NET.

DAQ Hardware Configuration

Before using a data acquisition board, you must confirm that the software can communicate with the board by configuring the devices. The devices are already configured for the computers in this class.

Windows

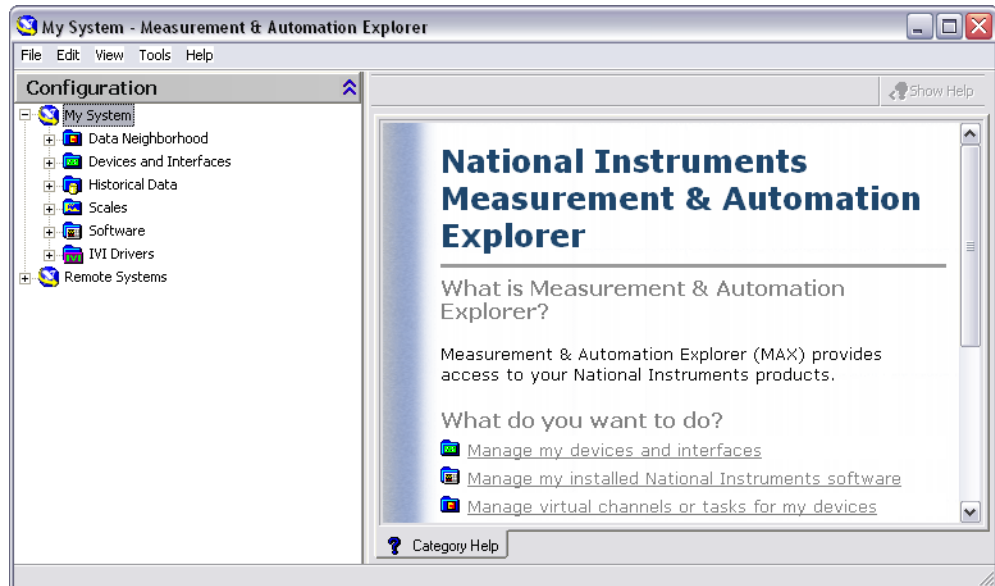
The Windows Configuration Manager keeps track of all the hardware installed in the computer, including National Instruments DAQ devices. If you have a Plug & Play (PnP) device, such as an E Series MIO device, the Windows Configuration Manager automatically detects and configures the device. If you have a non-PnP device, or legacy device, you must configure the device manually using the Add New Hardware option in the Control Panel.

You can verify the Windows Configuration by accessing the Device Manager. You can see **Data Acquisition Devices**, which lists all DAQ devices installed in the computer. Double-click a DAQ device to display a dialog box with tabbed pages. The **General** tab displays overall information regarding the device. The **Resources** tab specifies the system resources to the device such as interrupt levels, DMA, and base address for software-configurable devices. The **NI-DAQ Information** tab specifies the bus type of the DAQ device. The **Driver** tab specifies the driver version and location for the DAQ device.

Measurement & Automation Explorer

LabVIEW installs Measurement & Automation Explorer (MAX), which establishes all device and channel configuration parameters. After installing a DAQ device in the computer, you must run this configuration utility. MAX reads the information the Device Manager records in the Windows Registry and assigns a logical device number to each DAQ device. Use the device number to refer to the device in LabVIEW. Access MAX either by double-clicking the icon on the desktop or selecting **Tools»Measurement & Automation Explorer** in LabVIEW. The following window is the

primary MAX window. MAX is also the means for SCXI and SCC configuration.



The device parameters that you can set using the configuration utility depend on the device. MAX saves the logical device number and the configuration parameters in the Windows Registry.

The plug and play capability of Windows automatically detects and configures switchless DAQ devices, such as the PCI-6024E. When you install a device in the computer, the device is automatically detected.

Scales

You can configure custom scales for your measurements. This is very useful when working with sensors. It allows you to bring a scaled value into your application without having to work directly with the raw values. For example, in this course you use a temperature sensor that represents temperature with a voltage. The conversion equation for the temperature is: $\text{Voltage} \times 100 = \text{Celsius}$. After a scale is set, you can use it in your application program, providing the temperature value, rather than the voltage.

C. Simulating a DAQ Device

You can create NI-DAQmx simulated devices in NI-DAQmx 7.4 or later. Using NI-DAQmx simulated devices, you can try NI products in your application without the hardware. When you later acquire the hardware, you can import the NI-DAQmx simulated device configuration to the physical device using the MAX Portable Configuration Wizard. With NI-DAQmx simulated devices, you also can export a physical device configuration onto a system that does not have the physical device installed. Then, using the NI-DAQmx simulated device, you can work on your applications on a portable system and upon returning to the original system, you can easily import your application work.

Creating NI-DAQmx Simulated Devices

To create an NI-DAQmx simulated device, complete the following steps:

- Right-click **Devices and Interfaces** and select **Create»New**.
- A dialog box prompts you to select a device to add. Select **NI-DAQmx Simulated Device** and click **Finish**.
- In the **Choose Device** dialog box, select the family of devices for the device you want to simulate.
- Select the device and click **OK**. In the configuration tree in MAX, the icons for NI-DAQmx simulated devices are yellow. The icons for physical devices are green.
- If you select a PXI device, you are prompted to select a chassis number and PXI slot number.
- If you select an SCXI chassis, the SCXI configuration panels open.

Removing NI-DAQmx Simulated Devices

To remove an NI-DAQmx simulated device, complete the following steps:

- Expand **Devices and Interfaces»NI-DAQmx Devices**.
- Right-click the NI-DAQmx simulated device you want to delete.
- Click **Delete**.



Note In the configuration tree in MAX, the icons for NI-DAQmx simulated devices are yellow. The icons for physical devices are green.

Exercise 8-1 Concept: MAX

Goal

To use MAX to examine, configure, and test a device.

Description

Complete the following steps to examine the configuration for the DAQ device in the computer using MAX and use the test routines in MAX to confirm operation of the device. If you do not have a DAQ device, you can simulate a device using the instructions in *Part A. Creating a Simulated Device*.



Note Portions of this exercise that can only be completed with the use of a real device and a DAQ signal accessory are marked **Hardware**. Some of these steps have alternative instructions for simulated devices and are marked **No Hardware**.

1. Launch MAX by double-clicking the icon on the desktop or by selecting **Tools»Measurement & Automation Explorer** in LabVIEW. MAX searches the computer for installed National Instruments hardware and displays the information.

Part A. Creating a Simulated Device

2. Create an NI-DAQmx simulated device to allow you to complete the exercises in this chapter without hardware. If you have a DAQ device installed, you can skip this step.
 - Right-click **Devices and Interfaces** and select **Create»New**.
 - A dialog box prompts you to select a device to add. Select **NI-DAQmx Simulated Device** and click **Finish**.
 - In the **Choose Device** dialog box, select **M Series DAQ»NI PCI 6225**.
 - Click **OK**.

Part B. Examining the DAQ Device Settings

3. Expand the **Devices and Interfaces** section.
4. Expand the **NI-DAQmx Devices** section to view the installed National Instruments devices that use the NI-DAQmx driver.
5. Select the device listed in the **NI-DAQmx Devices** section. Figure 8-3 shows the PCI-MIO-16E-4 device.

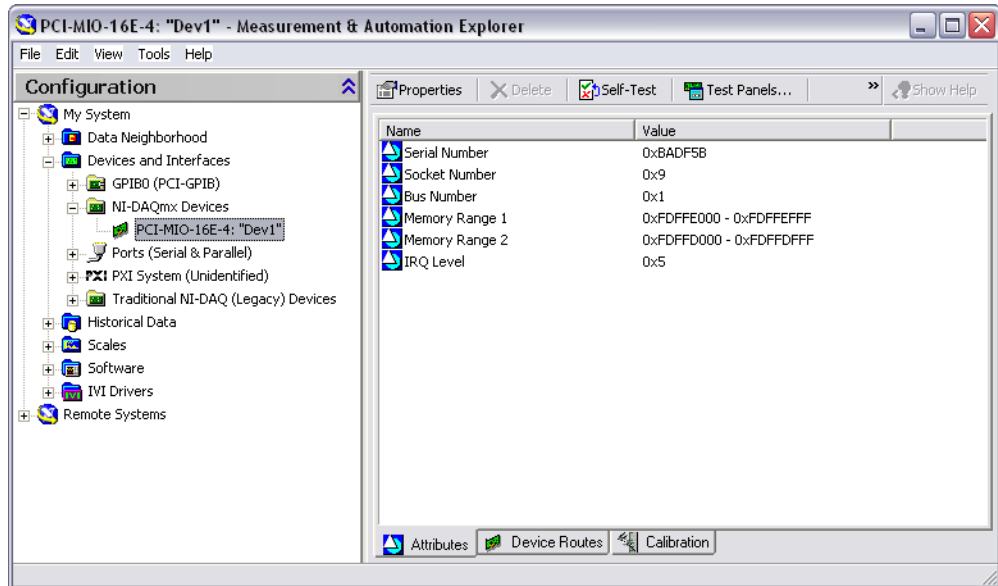


Figure 8-3. MAX with Device and Interfaces expanded

MAX displays the National Instruments hardware and software in the computer. The device number appears in quotes following the device name. The Data Acquisition VIs use this device number to determine which device performs DAQ operations. MAX also displays the attributes of the device such as the system resources that are being used by the device.



Note You might have a different device installed, and some of the options shown might be different. Click the **Show Help/Hide Help** button in the top right corner of MAX to hide the online help and show the DAQ device information.

6. Select the **Device Routes** tab to see detailed information about the internal signals that can be routed to other destinations on the device, as shown in Figure 8-4. This is a powerful resource that gives you a visual representation of the signals that are available to provide timing and synchronization with components that are on the device and other external devices.

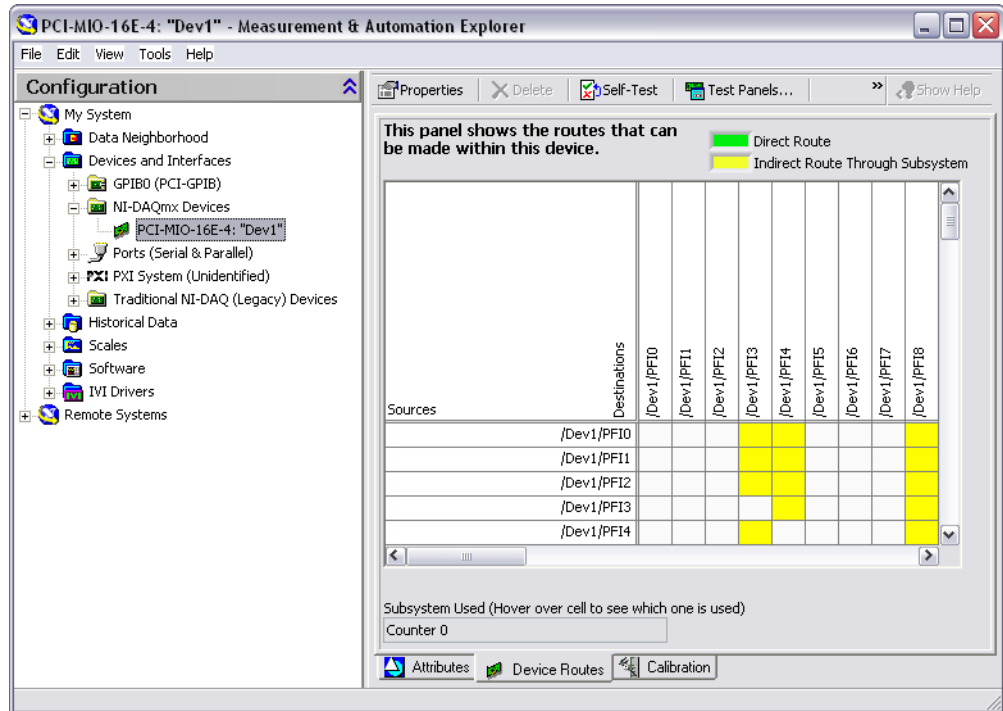


Figure 8-4. Device Routes

7. Select the **Calibration** tab, as shown in Figure 8-5, to see information about the last time the device was calibrated both internally and externally.

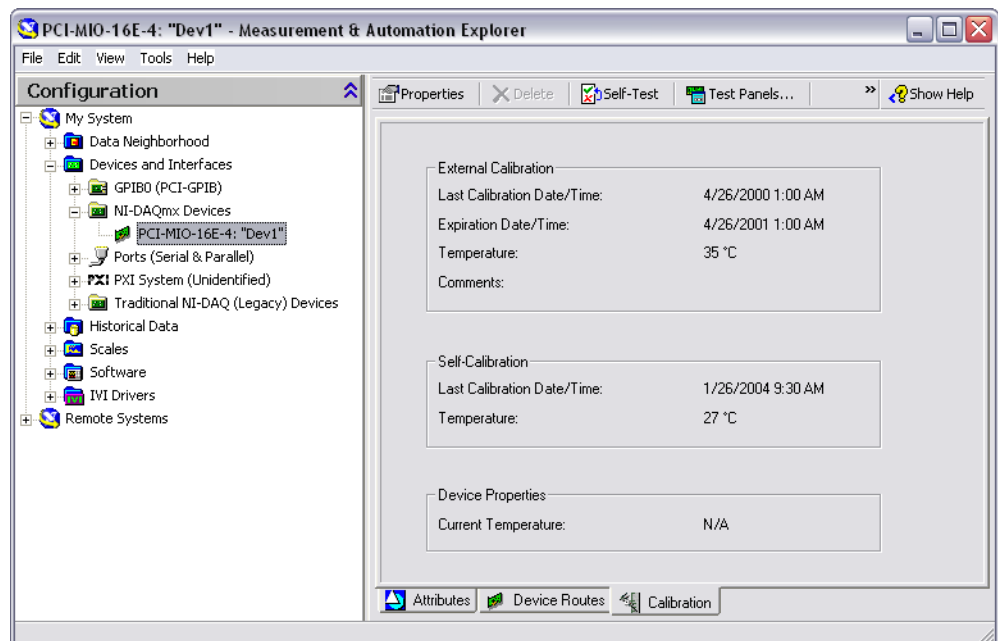


Figure 8-5. Calibration

8. Right-click the NI-DAQmx device in the configuration tree and select **Self-Calibrate** to calibrate the DAQ device using a precision voltage reference source and update the built-in calibration constants. When the device has been calibrated, the **Self Calibration** information updates in the **Calibration** tab. Skip this step if you are using a simulated device.

Part C. Testing the DAQ Device Components

9. Click the **Self-Test** button to test the device. This tests the system resources assigned to the device. The device should pass the test because it is already configured.
10. Click the **Test Panels** button to test the individual functions of the DAQ device, such as analog input and output. The **Test Panels** dialog box appears.
 - Use the **Analog Input** tab to test the various analog input channels on the DAQ device. Click the **Start** button to acquire data from analog input channel 0.
 - **Hardware:** If you have a DAQ Signal Accessory, channel Dev1/ai0 is connected to the temperature sensor. Place your finger on the sensor to see the voltage rise. You also can move the **Noise** switch to **On** on the DAQ Signal Accessory to see the signal change in this tab. When you are finished, click the **Stop** button.
 - **No Hardware:** If you are using a simulated device, a sine wave is shown on all input channels. Experiment with the setting on this tab. When you are finished, click the **Stop** button.
 - Click the **Analog Output** tab to set up a single voltage or sine wave on one of the DAQ device analog output channels.
 - Change the output **Mode** to **Sinewave Generation** and click the **Start** button. LabVIEW generates a continuous sine wave on analog output channel 0.
 - Hardware:** On the external DAQ Signal Accessory box, wire Analog Out Ch0 to Analog In Ch1.
 - Hardware:** Click the **Analog Input** tab and change the channel to Dev1/ai1. Click the **Start** button to acquire data from analog input channel 1. LabVIEW displays the sine wave from analog output channel 0.
 - Click the **Digital I/O** tab to test the digital lines on the DAQ device.

- Set lines 0 through 3 as output and toggle the **Logic Level** checkboxes. If you have a DAQ signal accessory, toggling the boxes turns the LEDs on or off. The LEDs use negative logic.
- Hardware:** Click the **Counter I/O** tab to determine if the DAQ device counter/timers are functioning properly. To verify counter/timer operation, change the counter **Mode** tab to **Edge Counting** and click the **Start** button. The **Counter Value** increments rapidly. Click **Stop** to stop the counter test.
- Click the **Close** button to close the **Test Panel** and return to MAX.

Part D. Setting a Custom Scale

11. **Hardware:** Create a custom scale for the temperature sensor on the DAQ Signal Accessory. The sensor conversion is linear, and the formula is $\text{Voltage} \times 100 = \text{Celsius}$.

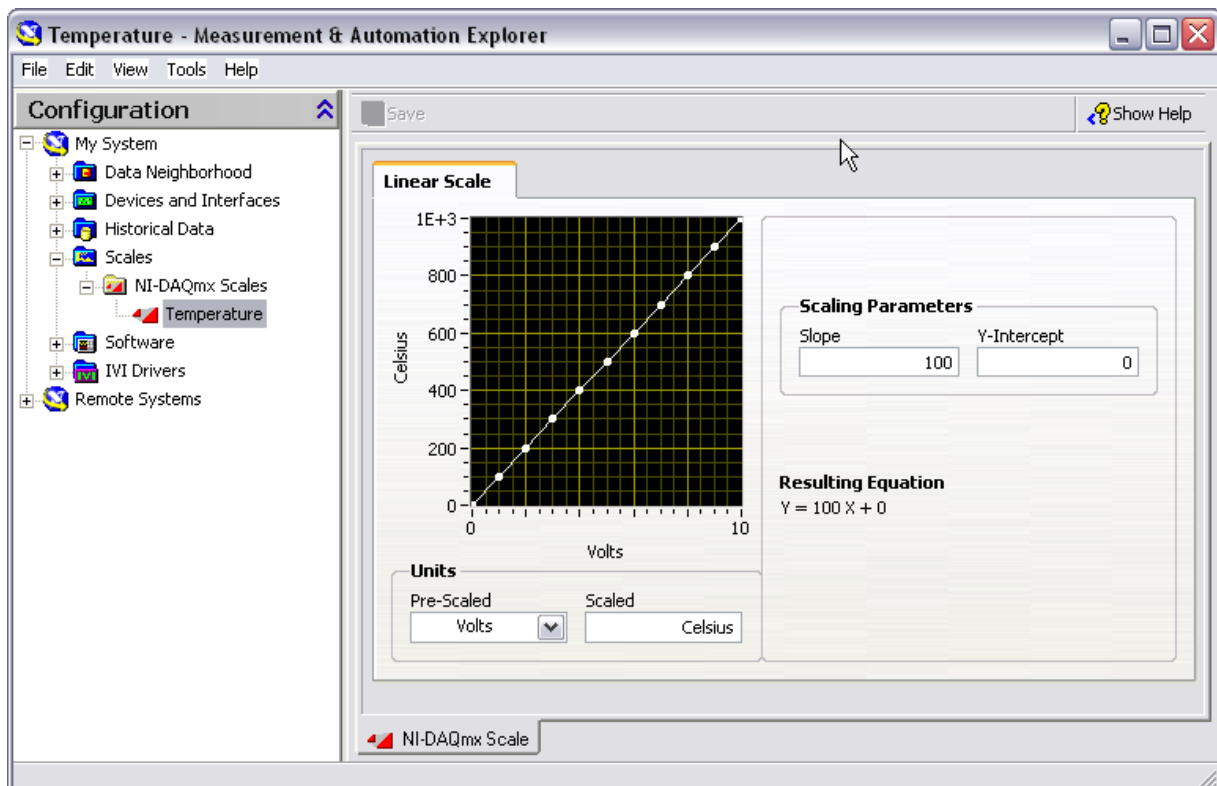


Figure 8-6. Temperature Scale

- Right-click the **Scales** section and select **Create New** from the shortcut menu.
- Select **NI-DAQmx Scale**.
- Click **Next**.

- Select **Linear**.
- Name the scale `Temperature`.
- Click **Finish**.
- Change the Scaling Parameters **Slope** to 100.
- Enter `Celsius` as the **Scaled Units**.
- Click the **Save** button on the toolbar to save the scale. You use this scale in later exercises.

12. Close MAX by selecting **File»Exit**.

End of Exercise 8-1

D. Analog Input

Analog input is the process of measuring an analog signal and transferring the measurement to a computer for analysis, display or storage. An analog signal is a signal that varies continuously. Analog input is most commonly used to measure voltage or current. You can use many types of devices to perform analog input, such as multifunction DAQ (MIO) devices, high-speed digitizers, digital multimeters (DMMs) and Dynamic Signal Acquisition (DSA) devices.

Analog-to-Digital Conversion

Acquiring an analog signal with a computer requires a process known as *analog-to-digital conversion*, which takes an electrical signal and translates it into digital data so that a computer can process it. *Analog-to-digital converters* (ADCs) are circuit components that convert a voltage level into a series of ones and zeroes.

ADCs sample the analog signal on each rising or falling edge of a sample clock. In each cycle, the ADC takes a snapshot of the analog signal, so that the signal can be measured and converted into a digital value. A *sample clock* controls the rate at which samples of the input signal are taken.

Because the incoming, or unknown signal is a real world signal with infinite precision, the ADC approximates the signal with fixed precision. After the ADC obtains this approximation, the approximation can be converted to a series of digital values. Some conversion methods do not require this step, because the conversion generates a digital value directly as the ADC reaches the approximation.

Task Timing

When performing analog input, the task can be timed to Acquire 1 Sample, Acquire n Samples, or Acquire Continuously.

Acquire 1 Sample

Acquiring a single sample is an on-demand operation. In other words, the driver acquires one value from an input channel and immediately returns the value. This operation does not require any buffering or hardware timing. For example, if you periodically monitor the fluid level in a tank, you would acquire single data points. You can connect the transducer that produces a voltage representing the fluid level to a single channel on the measurement device and initiate a single-channel, single-point acquisition when you want to know the fluid level.

Acquire *n* Samples

One way to acquire multiple samples for one or more channels is to acquire single samples in a repetitive manner. However, acquiring a single data

sample on one or more channels over and over is inefficient and time consuming. Moreover, you do not have accurate control over the time between each sample or channel. Instead you can use hardware timing, which uses a buffer in computer memory, to acquire data more efficiently. Programmatically, you need to include the timing function and specify the **sample rate** and the **sample mode (finite)**. As with other functions, you can acquire multiple samples for a single channel or multiple channels.

With NI-DAQmx, you also can gather data from multiple channels. For instance, you might want to monitor both the fluid level in the tank and the temperature. In such a case, you need two transducers connected to two channels on the device.

Acquire Continuously

If you want to view, process, or log a subset of the samples as they are acquired, you need to continually acquire samples. For these types of applications, set the sample mode to **continuous**.

Task Triggering

When a device controlled by NI-DAQmx does something, it performs an action. Two very common actions are producing a sample and starting a waveform acquisition. Every NI-DAQmx action needs a stimulus or cause. When the stimulus occurs, the action is performed. Causes for actions are called triggers. The start trigger starts the acquisition. The reference trigger establishes the reference point in a set of input samples. Data acquired up to the reference point is pretrigger data. Data acquired after the reference point is posttrigger data.

Exercise 8-2 Triggered Analog Input VI

Goal

To acquire an analog signal using a DAQ device and a digital trigger.

Scenario

Build a VI that measures the voltage signal on channel AI1 of the DAQ device. The VI begins measuring when a digital trigger is pressed and the Power switch on the front panel is on. The VI stops measuring when the Power switch on the front panel is off.

Design

User Interface Inputs and Outputs

Type	Name	Properties
Waveform Chart	Analog Input Data	X-Scale range: 1/100 second
Vertical Toggle Switch	Power	

External Inputs and Outputs

- Inputs: AI1 of the data acquisition board. Connect the sine function generator to channel analog input 1 on the DAQ Signal Accessory with a wire. You also can use a DAQ Simulated Device to acquire data.

Implementation

In the following steps, you build the front panel shown in Figure 8-7.

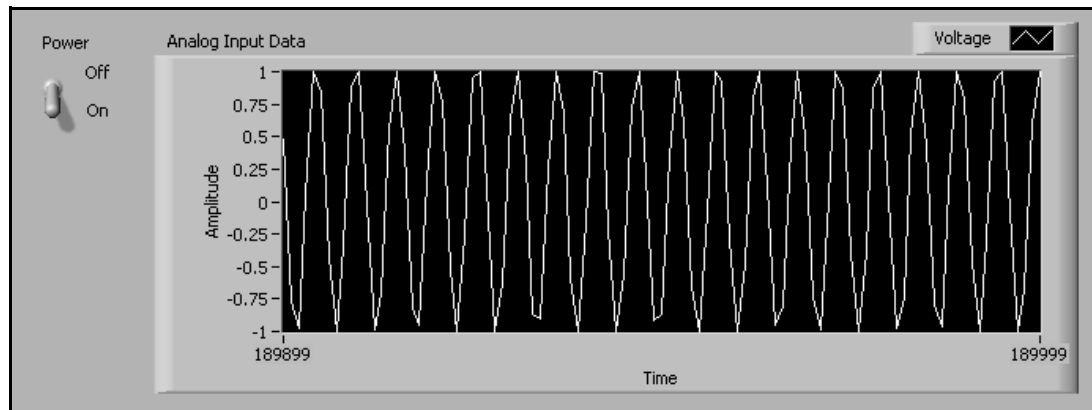


Figure 8-7. Triggered Analog Input front panel

1. Open a blank VI.
2. Create the Analog Input Data waveform chart.
 - Place a Waveform Chart on the front panel.
 - Rename the waveform chart `Analog Input Data`. You set up the scaling for the chart later in this exercise.
3. Create the Power vertical toggle switch.
 - Place a vertical toggle switch on the front panel.
 - Rename the switch `Power`.
 - Create two free labels, **Off** and **On**, using the Labeling tool.
 - Place the free labels as shown in Figure 8-7.
4. Save the VI as `Triggered Analog Input.vi` in the `C:\Exercises\LabVIEW Basics I\Triggered Analog Input` directory.

In the following steps, you build the block diagram shown in Figure 8-8.

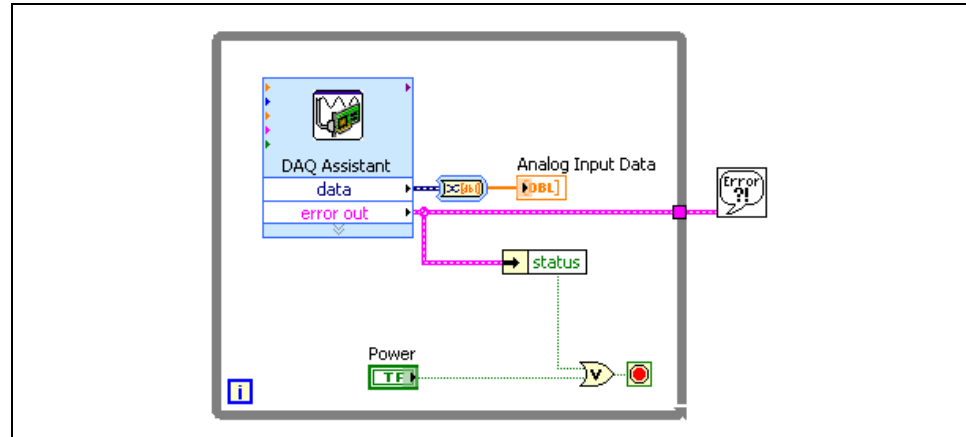


Figure 8-8. Triggered Analog Input Block Diagram

5. Set up the DAQ Assistant to acquire data on AI1 at 50KHz when the digital trigger is pressed. If you are using a simulated device, acquire the data without need of a trigger.



- Place the DAQ Assistant Express VI on the block diagram.
- Select **Analog Input»Voltage** for the measurement to make.
- Select **Dev1»ai1** for the physical channel.
- Click the **Finish** button.
- Set the **Signal Input Range** on the **Settings** tab to a range of -1 to 1 Volts.
- Set the **Acquisition Mode** on the **Task Timing** tab to **Continuous**.
- Set the **Samples to Read** in the **Clock Settings** section of the **Task Timing** tab to 5000 .
The number of samples defines the amount of data removed from the buffer at one time.
- Set the **Rate (Hz)** in the **Clock Settings** section of the **Task Timing** tab to $20k$.
- If you are using the DAQ Signal Accessory, switch to the **Task Triggering** tab. If you are using a NI-DAQmx Simulated Device, click the **OK** button and continue to step 6.
- Set the **Trigger Type** in the **Start Trigger** section of the **Task Triggering** tab to **Digital Edge**.
- Set the **Trigger Source** to **PFI0**.

- Set the **Edge** to **Rising**.
 - Click the **OK** button to close the **Analog Input Voltage Task Configuration** dialog box.
6. Click **OK** when the DAQ Assistant asks if you would like to have a While Loop automatically generated. Notice that it creates the While Loop, status Unbundle by Name, Or function and a Stop button for you.
 7. Delete the Stop button; you use the Power switch instead.
 8. Place the Power terminal in the While Loop.
 9. Convert the acquired data to an array of numerics to graph the data by sample number rather than time.



- Place a Convert from Dynamic Data Express VI in the While Loop.
- In the **Config** dialog box Select **1D array of scalars - automatic** in the **Resulting data type** listbox.
- Click **OK**.



10. Place a Simple Error Handler VI to the right of the While Loop.
11. Wire the block diagram as shown in Figure 8-8.
12. Switch to the front panel.
13. Adjust the X-Scale of the waveform chart.
 - Right-click the waveform chart and select **X-Scale»AutoScale X** to turn off autoscaling.
 - Use the Labeling tool to change the last value on the X Scale of the Waveform Chart to 100.
14. Save the VI.

Testing

1. If you are using the DAQ Signal Accessory, confirm that a wire connects the sine function generator to analog in channel 1.
2. Use the Operating Tool to put the Power switch in the On position.
3. Run the VI.

Hardware

- Press the digital trigger button on the DAQ Signal Accessory. The waveform chart should start displaying a sine wave.
- Change the frequency of the sine wave using the Frequency Adjust dial on the DAQ Signal Accessory.

No Hardware

- The waveform chart should start displaying a sine wave. You do not use a trigger because you cannot set up a digital trigger on a simulated device because there is no physical trigger to switch.
4. Switch the Power switch to the Off position when you are finished. The VI should stop.
 5. What happens if you start the VI with the switch in the Off position? Is this desired behavior?
 6. Modify the Power switch so that it returns to the On position after it is pressed, and the On position is the default value.
 - Use the Operating tool to place the Power switch in the On position.
 - Right-click the Power switch and select **Data Operations»Make Current Value Default** from the shortcut menu.
 - Right-click the Power switch and select **Mechanical Action»Latch When Pressed** from the shortcut menu.
 7. Run the VI. Does the Power switch behave as you expect?
 8. Stop and close the VI.

End of Exercise 8-2

E. Analog Output

Analog output is the process of generating electrical signals from your computer. Analog output is generated by performing digital-to-analog (D/A) conversions. The available analog output types for a task are voltage and current.

To perform a voltage or current task, a compatible device must be installed that can generate that form of signal.

Task Timing

When performing analog output, the task can be timed to Generate 1 Sample, Generate n Samples, or Generate Continuously.

Generate 1 Sample

Use single updates if the signal level is more important than the generation rate. For example, generate one sample at a time if you need to generate a constant, or DC, signal. You can use software timing to control when the device generates a signal.

This operation does not require any buffering or hardware timing. For example, if you need to generate a known voltage to stimulate a device, a single update would be an appropriate task.

Generate n Samples

One way to generate multiple samples for one or more channels is to generate single samples in a repetitive manner. However, generating a single data sample on one or more channels over and over is inefficient and time consuming. Moreover, you do not have accurate control over the time between each sample or channel. Instead, you can use hardware timing, which uses a buffer in computer memory to generate samples more efficiently.

You can use software timing or hardware timing to control when a signal is generated. With software timing, the rate at which the samples are generated is determined by the software and operating system instead of by the measurement device. With hardware timing, a TTL signal, such as a clock on the device, controls the rate of generation. A hardware clock can run much faster than a software loop. A hardware clock is also more accurate than a software loop.



Note Some devices do not support hardware timing. Consult the device documentation if you are unsure if the device supports hardware timing.

Programmatically, you need to include the timing function, specifying the **sample rate** and the **sample mode (finite)**. As with other functions, you can generate multiple samples for a single channel or multiple channels.

Use Generate n Samples if you want to generate a finite time-varying signal, such as an AC sine wave.

Generate Continuously

Continuous generation is similar to Generate n Samples, except that an event must occur to stop the generation. If you want to continuously generate signals, such as generating a non-finite AC sine wave, set the timing mode to **continuous**.

Task Triggering

When a device controlled by NI-DAQmx does something, it performs an action. Two very common actions are producing a sample and starting a generation. Every NI-DAQmx action needs a stimulus or cause. When the stimulus occurs, the action is performed. Causes for actions are called triggers. A start trigger starts the generation.

Digital-to-Analog Conversion

Digital-to-analog conversion is the opposite of analog-to-digital conversion. In digital-to-analog conversion, the data starts in the computer. The data might have been acquired earlier using analog input or may have been generated by software on the computer. A digital-to-analog converter (DAC) accepts this data and uses it to vary the voltage on an output pin over time. The DAC generates an analog signal that the DAC can send to other devices or circuits.

A DAC has an update clock that tells the DAC when to generate a new value. The function of the update clock is similar to the function of the sample clock for an analog-to-digital converter (ADC). At each cycle the clock, the DAC converts a digital value to an analog voltage and creates an output as a voltage on a pin. When used with a high speed clock, the DAC can create a signal that appears to vary constantly and smoothly.

F. Counters

A counter is a digital timing device. You typically use counters for event counting, frequency measurement, period measurement, position measurement, and pulse generation.

- **Count Register**—Stores the current count of the counter. You can query the count register with software.

- **Source**—An input signal that can change the current count stored in the count register. The counter looks for rising or falling edges on the source signal. Whether a rising or falling edge changes the count is software selectable. The type of edge selected is referred to as the active edge of the signal. When an active edge is received on the source signal, the count changes. Whether an active edge increments or decrements the current count is also software selectable.
- **Gate**—An input signal that determines if an active edge on the source changes the count. Counting can occur when the gate is high, low, or between various combinations of rising and falling edges. Gate settings are made in software.
- **Output**—An output signal that generates pulses or a series of pulses, otherwise known as a pulse train.

When you configure a counter for simple event counting, the counter increments when an active edge is received on the source. In order for the counter to increment on an active edge, the counter must be armed or started. A counter has a fixed number it can count to as determined by the resolution of the counter. For example, a 24-bit counter can count to:

$$2^{(\text{Counter Resolution})} - 1 = 2^{24} - 1 = 16,777,215$$

When a 24-bit counter reaches the value of 16,777,215, it has reached the terminal count. The next active edge forces the counter to roll over and start at 0.

Exercise 8-3 Count Events VI

Goal

Use the DAQ Assistant to input a counter value.

Scenario

You have been asked to build a VI that counts pulses from the quadrature encoder on the DAQ Signal Accessory.

Design

Quadrature Encoder

A 24-pulse per revolution mechanical quadrature encoder measure the position of a shaft as it rotates. The DAQ signal accessory quadrature encoder is a knob located in the upper central portion of the top panel. The quadrature encoder produces two pulse train outputs corresponding to the shaft position as you rotate the knob. Depending on the direction of rotation, phase A leads phase B by 90° or phase B leads phase A by 90°.

The DAQ Signal Accessory internally connects phase B of the quadrature encoder to the Up/Down line for Counter 0 (DIO6). Connect phase A of the quadrature encoder to the Source of Counter 0 (PFI8).

User Interface Inputs and Outputs

Type	Name	Properties
Numeric Indicator	Number of Events	Double
Stop Button	Stop	

External Inputs

- Counter 0 Source (PFI8): Phase A of quadrature encoder
- Counter 0 Up/Down (DIO6): Phase B of quadrature encoder

Implementation

1. Open a blank VI and build the front panel shown in Figure 8-9.



Figure 8-9. Count Events VI front panel

2. Save the VI as `Count Events.vi` in the `C:\Exercises\LabVIEW Basics I\Count Events` directory.
3. Switch to the block diagram.
4. Configure DAQ Assistant Express VI to use the counter to perform event counting.

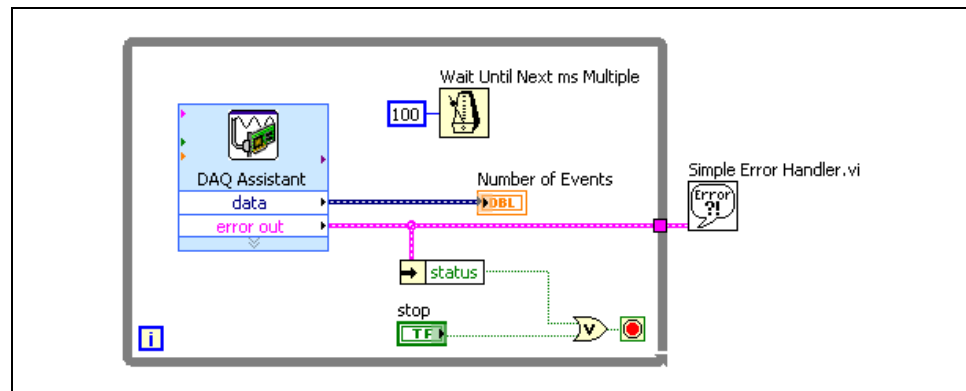


Figure 8-10. Count Events VI block diagram



- Place the DAQ Assistant Express VI on the block diagram.
 - Select **Counter Input**»**Edge Count** for the measurement to make.
 - Select **Dev1**»**ctr0** for the physical channel.
 - Click the **Finish** button.
 - Change the **Active Edge** pull-down menu to **Falling**.
 - Change the **Count Direction** pull-down menu to **Externally Controlled**.
 - Click the **OK** button to close the configuration dialog box.
5. Finish building the block diagram. Use Figure 8-10 as a guide to assist you.

6. Save the VI.

Testing

1. On the DAQ Signal Accessory, confirm that the A output of the quadrature encoder is wired to the SOURCE input of counter 0.
2. Run the VI.
3. Rotate the quadrature encoder knob on the DAQ Signal Accessory. Notice that the Number of Events indicator increments as you rotate the knob. The quadrature encoder knob produces pulses as you rotate the knob. The counter counts these pulses.

Rotate the quadrature encoder knob in the other direction. Notice that the Number of Events indicator decrements when you rotate the knob clockwise, and increments when you rotate the knob counterclockwise.

If the VI does not work as you expect, you may need to reset the DAQ Device in Measurement and Automation Explorer.
4. Stop the VI.
5. Save and close the VI.

End of Exercise 8-3

G. Digital I/O

Digital signals are electrical signals that transfer digital data over a wire. These signals typically have only two states: on and off, also known as high and low, or 1 and 0. When sending a digital signal across a wire, the sender applies a voltage to the wire and the receiver uses the voltage level to determine the value being sent. The voltage ranges for each digital value depend on the voltage level standard being used. Digital signals have many uses; the simplest application of a digital signal is controlling or measuring digital or finite state devices such as switches and LEDs. Digital signals also can transfer data; you can use them to program devices or communicate between devices. In addition, you can use digital signals as clocks or triggers to control or synchronize other measurements.

You can use the digital lines in a DAQ device to acquire a digital value. This acquisition is based on software timing. On some devices, you can configure the lines individually to either measure or generate digital samples. Each line corresponds to a channel in the task.

You can use the digital port(s) in a DAQ device to acquire a digital value from a collection of digital lines. This acquisition is based on software timing. You can configure the ports individually to either measure or generate digital samples. Each port corresponds to a channel in the task.

Exercise 8-4 Optional: Digital Count VI

Goal

Use the DAQ Assistant for digital I/O.

Scenario

Write a VI that converts the number of events generated by the quadrature encoder to a digital number to display on the LEDs on the DAQ Signal Accessory. Because there are only four LEDs, you are limited to a number between 0 and 15 (2^4). For numbers greater than 15 and less than 0, the LEDs should continue changing as though there were more LEDs available.

Design

Digital I/O

Each LED is wired to a digital line on the DAQ device. The lines are numbered 0, 1, 2, and 3, starting with the LED on the right. You can write to these lines individually or as a digital port. However, the digital port includes all 8 DIO lines. Because the quadrature encoder uses DIO6 for up/down counting, you cannot write to DIO6. Therefore, in this example, you should write a Boolean array to digital lines 0–4.

Flowchart

When a number is converted to a Boolean array, the number of elements in the array depends on the representation of the number used. If the number is 32-bytes, there are 32 elements in the Boolean array. However, because there are only four LEDs, you only need the first 4 elements of the array.

Modify the Count Events VI as shown in the following flowchart.

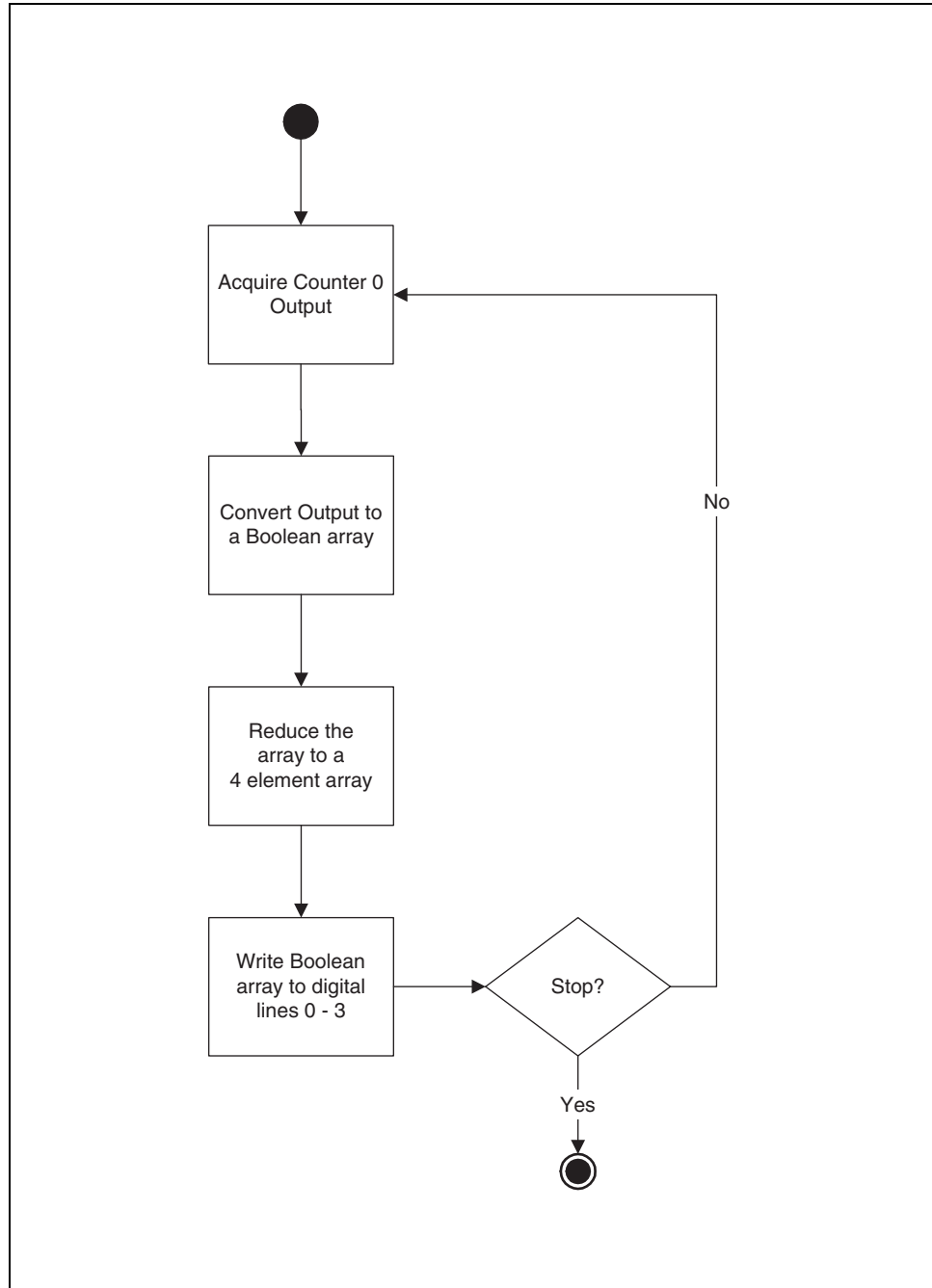


Figure 8-11. Digital Count Flowchart

Implementation

1. Open `Count Events.vi` in the `C:\Exercises\LabVIEW Basics I\Count Events` directory.
2. Save the VI as `Digital Count.vi`.
3. Switch to the block diagram of the VI.

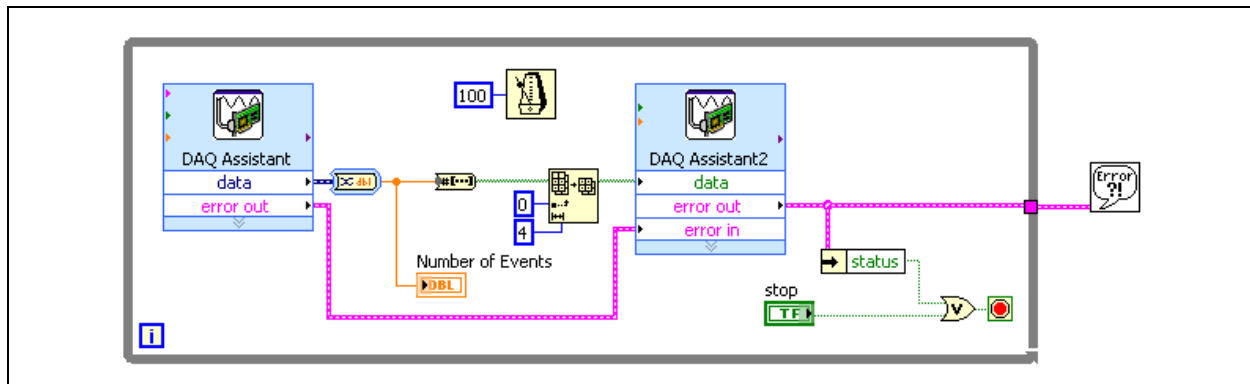


Figure 8-12. Digital Count VI Block Diagram

4. Delete the wire connected to the Number of Events terminal.
5. Delete the wire connected from the error terminal to the status element of the Unbundle by Name functions.
6. Enlarge the While Loop and increase the amount of space between the DAQ Assistant and the stop conditions.
7. Convert the count to a Boolean Array.



- Place a Convert From Dynamic Data Express VI to the right of the data output of the DAQ Assistant.
- Set the resulting data type to **Single Scalar**.
- Click **OK** to close the dialog box.



- Place a Number to Boolean Array function to the right of the Convert From Dynamic Data Express VI.
- Wire as shown in Figure 8-12.

8. Create a subarray containing the first four elements of the Boolean array.



- Place the Array Subset function to the right of the Number to Boolean Array function.

- Wire the output of the Number to Boolean Array function to the **array** terminal of the Array Subset function.
 - Right-click the **index** terminal and select **Create»Constant** from the shortcut menu.
 - Set the constant to 0.
 - Right-click the **length** terminal and select **Create»Constant** from the shortcut menu.
 - Set the constant to 4.
9. Configure digital lines 0–3 for edge counting.



- Place the DAQ Assistant Express VI in the While Loop.
- Select **Digital I/O»Line Output** for the measurement to make.
- Select **Dev1»line 0–line 3** for the physical channels and click the **Finish** button.
- For each line, select **Invert Line** because the LEDs use negative logic.
- Click the **OK** button to close the configuration dialog box.



Note In this exercise, you use individual lines rather than a port because DIO6 is used by Phase B of the Quadrature Encoder.

10. Wire the block diagram as shown in Figure 8-12.
11. Save the VI.

Testing

1. Display the front panel.
2. Run the VI.
3. Turn the quadrature encoder and observe the changes on the DAQ Signal Accessory.
4. Stop the VI.
5. Close the VI.

End of Exercise 8-4

Self-Review: Quiz

1. You are reading a signal at 50kHz. You want to acquire the signal until a stop trigger is pressed. Which task timing should you use?
 - a. Acquire 1 Sample
 - b. Acquire N Samples
 - c. Acquire Continuously

2. Your VI monitors a factory floor. Part of the VI controls an LED which is used to alert users to the status of the system. Which task timing should you use?
 - a. Generate 1 Sample
 - b. Generate N Samples
 - c. Generate Continuously

Self-Review: Quiz

1. You are reading a signal at 50kHz. You want to acquire the signal until a stop trigger is pressed. Which task timing should you use?
 - a. Acquire 1 Sample
 - b. Acquire N Samples
 - c. **Acquire Continuously**

2. Your VI monitors a factory floor. Part of the VI controls an LED which is used to alert users to the status of the system. Which task timing should you use?
 - a. **Generate 1 Sample**
 - b. Generate N Samples
 - c. Generate Continuously

Notes

Instrument Control

This lesson describes instrument control of stand-alone instruments using a GPIB or serial interface. Use LabVIEW to control and acquire data from instruments with the Instrument I/O Assistant, the VISA API, and instrument drivers.

Topics

- A. Instrument Control
- B. GPIB
- C. Serial Port Communication
- D. Using Other Interfaces
- E. Software Architecture
- F. Instrument I/O Assistant
- G. VISA
- H. Instrument Drivers

A. Instrument Control

When you use a PC to automate a test system, you are not limited to the type of instrument you can control. You can mix and match instruments from various categories. The most common categories of instrument interfaces are GPIB, serial, modular instruments, and PXI modular instruments. Additional types of instruments include image acquisition, motion control, USB, Ethernet, parallel port, NI-CAN, and other devices.

When you use PCs to control instruments, you need to understand properties of the instrument, such as the communication protocols to use. Refer to the instrument documentation for information about the properties of an instrument.

B. GPIB

The ANSI/IEEE Standard 488.1-1987, also known as General Purpose Interface Bus (GPIB), describes a standard interface for communication between instruments and controllers from various vendors. GPIB, or General Purpose Interface Bus, instruments offer test and manufacturing engineers the widest selection of vendors and instruments for general-purpose to specialized vertical market test applications. GPIB instruments are often used as stand-alone benchtop instruments where measurements are taken by hand. You can automate these measurements by using a PC to control the GPIB instruments.

IEEE 488.1 contains information about electrical, mechanical, and functional specifications. The ANSI/IEEE Standard 488.2-1992 extends IEEE 488.1 by defining a bus communication protocol, a common set of data codes and formats, and a generic set of common device commands.

GPIB is a digital, 8-bit parallel communication interface with data transfer rates of 1 Mbyte/s and higher, using a three-wire handshake. The bus supports one system controller, usually a computer, and up to 14 additional instruments. The GPIB protocol categorizes devices as controllers, talkers, or listeners to determine which device has active control of the bus. Each device has a unique GPIB primary address between 0 and 30. The Controller defines the communication links, responds to devices that request service, sends GPIB commands, and passes/receives control of the bus. Controllers instruct Talkers to talk and to place data on the GPIB. You can address only one device at a time to talk. The Controller addresses the Listener to listen and to read data from the GPIB. You can address several devices to listen.

Data Transfer Termination

Termination informs listeners that all data has been transferred. You can terminate a GPIB data transfer in the following three ways:

- The GPIB includes an End Or Identify (EOI) hardware line that can be asserted with the last data byte. This is the preferred method.
- Place a specific end-of-string (EOS) character at the end of the data string itself. Some instruments use this method instead of or in addition to the EOI line assertion.
- The listener counts the bytes transferred by handshaking and stops reading when the listener reaches a byte count limit. This method is often used as a default termination method because the transfer stops on the logical OR of EOI, EOS (if used) in conjunction with the byte count. Thus, you typically set the byte count to equal or exceed the expected number of bytes to be read.

Data Transfer Rate

To achieve the high data transfer rate that the GPIB was designed for, you must limit the number of devices on the bus and the physical distance between devices.

You can obtain faster data rates with HS488 devices and controllers. HS488 is an extension to GPIB that most NI controllers support.



Note Refer to the National Instruments GPIB support Web site at ni.com/support/gpibsupp.htm for more information about GPIB.

C. Serial Port Communication

Serial communication transmits data between a computer and a peripheral device, such as a programmable instrument or another computer. Serial communication uses a transmitter to send data one bit at a time over a single communication line to a receiver. Use this method when data transfer rates are low or you must transfer data over long distances. Most computers have one or more serial ports, so you do not need any extra hardware other than a cable to connect the instrument to the computer or to connect two computers to each other.

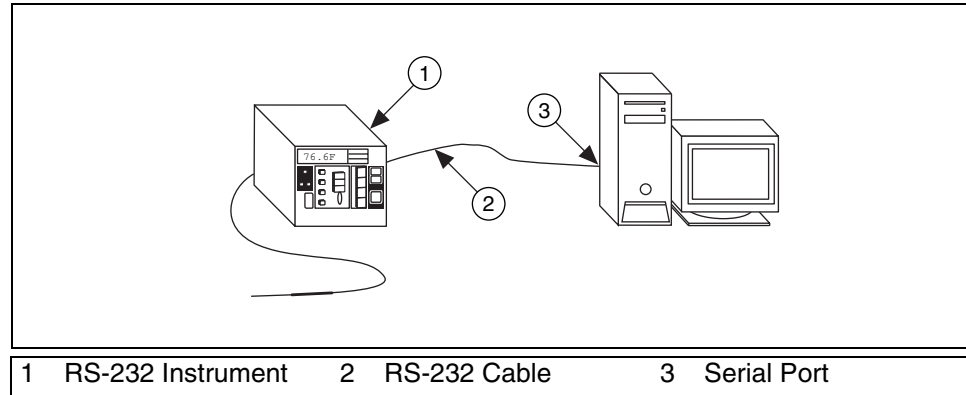


Figure 9-1. Serial Instrument Example

You must specify four parameters for serial communication: the baud rate of the transmission, the number of data bits that encode a character, the sense of the optional parity bit, and the number of stop bits. A character frame packages each transmitted character as a single start bit followed by the data bits.

Baud rate is a measure of how fast data moves between instruments that use serial communication.

Data bits are transmitted upside down and backwards, which means that inverted logic is used and the order of transmission is from least significant bit (LSB) to most significant bit (MSB). To interpret the data bits in a character frame, you must read from right to left and read 1 for negative voltage and 0 for positive voltage.

An optional parity bit follows the data bits in the character frame. The parity bit, if present, also follows inverted logic. This bit is included as a means of error checking. You specify ahead of time for the parity of the transmission to be even or odd. If you choose for the parity to be odd, the parity bit is set in such a way so the number of 1s add up to make an odd number among the data bits and the parity bit.

The last part of a character frame consists of 1, 1.5, or 2 stop bits that are always represented by a negative voltage. If no further characters are transmitted, the line stays in the negative (MARK) condition. The transmission of the next character frame, if any, begins with a start bit of positive (SPACE) voltage.

The following figure shows a typical character frame encoding the letter m.

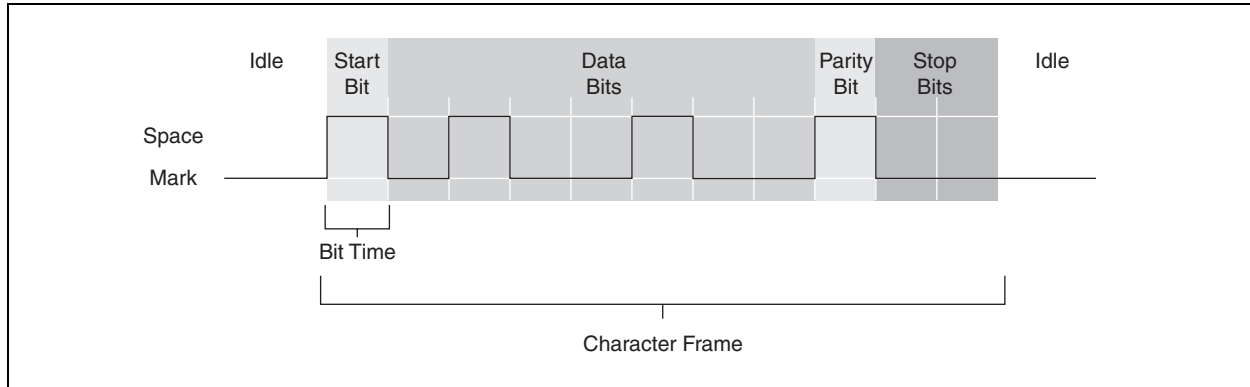


Figure 9-2. Character Frame for the Letter M

RS-232 uses only two voltage states, called MARK and SPACE. In such a two-state coding scheme, the baud rate is identical to the maximum number of bits of information, including control bits, that are transmitted per second.

MARK is a negative voltage, and SPACE is positive. The previous illustration shows how the idealized signal looks on an oscilloscope. The following is the truth table for RS-232:

Signal $> +3$ V = 0

Signal < -3 V = 1

The output signal level usually swings between +12 V and -12 V. The dead area between +3 V and -3 V is designed to absorb line noise.

A start bit signals the beginning of each character frame. It is a transition from negative (MARK) to positive (SPACE) voltage. Its duration in seconds is the reciprocal of the baud rate. If the instrument is transmitting at 9,600 baud, the duration of the start bit and each subsequent bit is about 0.104 ms. The entire character frame of eleven bits would be transmitted in about 1.146 ms.

Interpreting the data bits for the transmission yields 1101101 (binary) or 6D (hex). An ASCII conversion table shows that this is the letter m.

This transmission uses odd parity. There are five ones among the data bits, already an odd number, so the parity bit is set to 0.

Data Transfer Rate

You can calculate the maximum transmission rate in characters per second for a given communication setting by dividing the baud rate by the bits per character frame. In the previous example, there are a total of eleven bits per character frame. If the transmission rate is set at 9,600 baud, you get $9,600/11 = 872$ characters per second. Notice that this is the maximum character

transmission rate. The hardware on one end or the other of the serial link might not be able to reach these rates, for various reasons.

Serial Port Standards

The following examples are the most common recommended standards of serial port communication:

- RS-232 (ANSI/EIA-232 Standard) is used for many purposes, such as connecting a mouse, printer, or modem. It also is used with industrial instrumentation. Because of improvements in line drivers and cables, applications often increase the performance of RS-232 beyond the distance and speed in the standards list. RS-232 is limited to point-to-point connections between PC serial ports and devices.
- RS-422 (AIA RS-422A Standard) uses a differential electrical signal as opposed to the unbalanced (single-ended) signals referenced to ground with RS-232. Differential transmission, which uses two lines each to transmit and receive signals, results in greater noise immunity and longer transmission distances as compared to RS-232.
- RS-485 (EIA-485 Standard) is a variation of RS-422 that allows you to connect up to 32 devices to a single port and define the necessary electrical characteristics to ensure adequate signal voltages under maximum load. With this enhanced multidrop capability, you can create networks of devices connected to a single RS-485 serial port. The noise immunity and multidrop capability make RS-485 an attractive choice in industrial applications that require many distributed devices networked to a PC or other controller for data collection and other operations.

D. Using Other Interfaces

There are devices made to communicate with serial or GPIB instruments through the Ethernet, USB, or IEEE 1394 (FireWire) ports, which bypasses the need for a serial port or GPIB board on your computer. When using these devices, program them just as you would if they were using the serial port or a GPIB board.

USB and ethernet interfaces transform USB ports or ethernet ports into asynchronous serial ports for communication with serial instruments. You can install and use these interfaces as standard serial ports from your existing applications.

USB, ethernet, and IEEE 1394 controllers transform any computer with these ports into a full-function, Plug and Play, IEEE-488.2 Controller that can control up to 14 programmable GPIB instruments.

E. Software Architecture

The software architecture for instrument control using LabVIEW is similar to the architecture for DAQ. Instrument interfaces such as GPIB include a set of drivers. Use MAX to configure the interface. VISA, Virtual Instrument Software Architecture, is a common API to communicate with the interface drivers and is the preferred method used when programming for instrument control in LabVIEW, because VISA abstracts the type of interface used. Many LabVIEW VIs used for instrument control use the VISA API. For example, the Instrument I/O Assistant is a LabVIEW Express VI that can use VISA to communicate with message-based instruments and convert the response from raw data to an ASCII representation. Use the Instrument I/O Assistant when an instrument driver is not available. In LabVIEW, an instrument driver is a set of VIs specially written to communicate with an instrument.



Note GPIB drivers are available on the LabVIEW Installer CD-ROM and most GPIB drivers are available for download at ni.com/support/gpib/versions.htm. Always install the newest version of these drivers unless otherwise instructed in the release notes.

MAX (Windows; GPIB)

(Windows) Use MAX to configure and test the GPIB interface. MAX interacts with the various diagnostic and configuration tools installed with the driver and also with the Windows Registry and Device Manager. The driver-level software is in the form of a DLL and contains all the functions that directly communicate with the GPIB interface. The Instrument I/O VIs and functions directly call the driver software.



Note (Mac OS and UNIX) Refer to documentation supplied with your GPIB interface device for information about configuring and testing the interface.

Open MAX by double-clicking the icon on the desktop or by selecting **Tools»Measurement & Automation Explorer** in LabVIEW. The following example shows a GPIB interface in MAX after clicking the **Scan For Instruments** button on the toolbar.

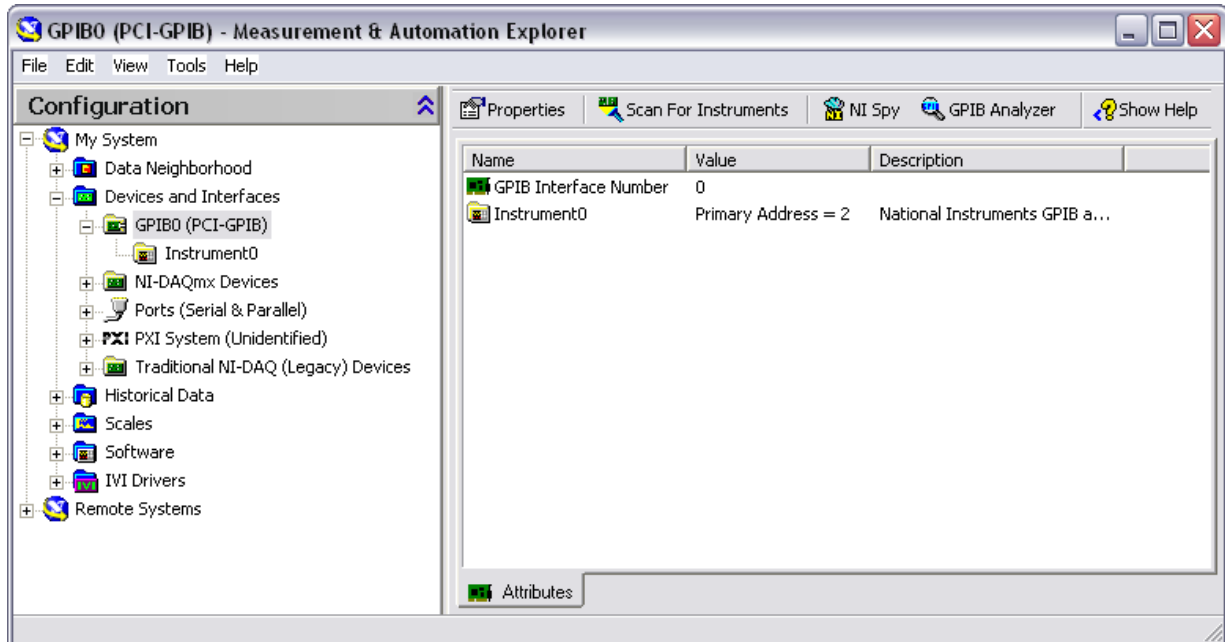


Figure 9-3. GPIB Interface in Measurement and Automation Explorer

Configure the objects listed in MAX by right-clicking each item and selecting an option from the shortcut menu. You learn to use MAX to configure and communicate with a GPIB instrument in the next exercise.

Exercise 9-1 Concept: GPIB Configuration with MAX

Goal

Learn to configure the NI Instrument Simulator and use MAX to examine the GPIB interface settings, detect instruments, and communicate with an instrument.

Description

1. Configure the NI Instrument Simulator.
 - Power off the NI Instrument Simulator.
 - Set the left bank of switches on the side of the box to match Figure 9-4.
 - Power on the NI Instrument Simulator.
 - Verify that both the Power and Ready LEDs are lit.

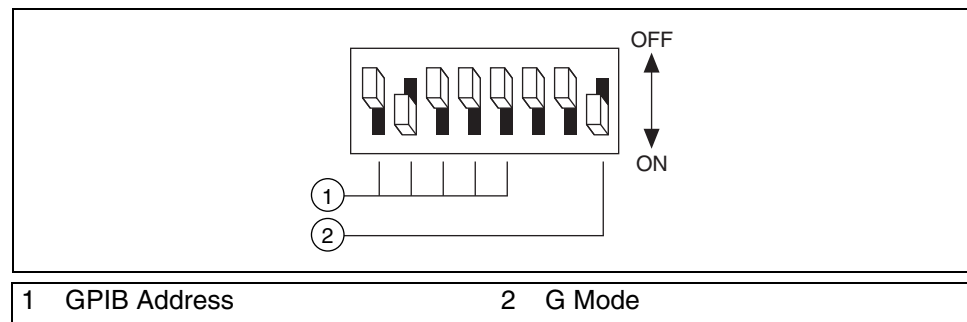


Figure 9-4. GPIB Configuration Settings for the NI Instrument Simulator

2. Launch MAX by either double-clicking the icon on the desktop or by selecting **Tools»Measurement & Automation Explorer** in LabVIEW.
3. View the settings for the GPIB interface.
 - Expand the **Devices and Interfaces** section to display the installed interfaces. If a GPIB interface is listed, the NI-488.2 software is correctly loaded on the computer.
 - Select the GPIB interface and click the **Properties** button on the toolbar to display the **Properties** dialog box.
 - Examine but do not change the settings for the GPIB interface.
 - Click the **OK** button to close the dialog box.

4. Communicate with the GPIB instrument.
 - Make sure the GPIB interface is still selected in the **Devices and Interfaces** section.
 - Click the **Scan for Instruments** button on the toolbar.
 - Expand the GPIB interface that is selected in the **Devices and Interfaces** section. One instrument named `Instrument0` appears.
 - Click **Instrument0** to display information about it in the right pane of MAX. Notice that the NI Instrument Simulator has a GPIB primary address (PAD) of 2.
 - Click the **Communicate with Instrument** button on the toolbar. An interactive window appears. You can use it to query, write to, and read from that instrument.
 - Enter `*IDN?` in **Send String** and click the **Query** button. The instrument returns its make and model number in **String Received** as shown in Figure 9-5. You can use this window to debug instrument problems or to verify that specific commands work as described in the instrument documentation.

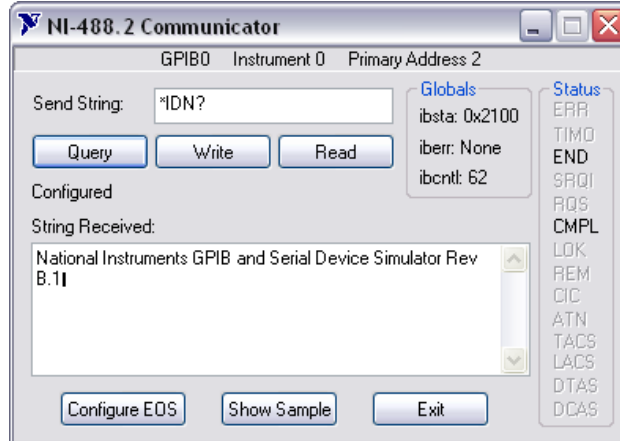


Figure 9-5. Communication with the GPIB instrument

- Enter `MEAS:DC?` in **Send String** and click the **Query** button. The NI Instrument Simulator returns a simulated voltage measurement.
- Click the **Query** button again to return a different value.
- Click the **Exit** button when done.

5. Set a VISA alias of `devsim` for the NI Instrument Simulator so you can use the alias instead of having to remember the primary address.
 - While **Instrument0** is selected in MAX, select the **VISA Properties** tab.
 - Enter `devsim` in the **VISA Alias on My System** field. You will use this alias throughout this lesson.
6. Select **File»Exit** to exit MAX.
7. Click **Yes** when prompted to save the instrument.

End of Exercise 9-1

F. Instrument I/O Assistant

The Instrument I/O Assistant is a LabVIEW Express VI which you can use to communicate with message-based instruments and convert the response from raw data to an ASCII representation. You can communicate with an instrument that uses a serial, Ethernet, or GPIB interface. Use the Instrument I/O Assistant when an instrument driver is not available.

The Instrument I/O Assistant organizes instrument communication into ordered steps. To use Instrument I/O Assistant, you place steps into a sequence. As you add steps to the sequence, they appear in the **Step Sequence** window. Use the view associated with a step to configure instrument I/O.

To launch the Instrument I/O Assistant, place the Instrument I/O Assistant Express VI on the block diagram in LabVIEW. The Instrument I/O Assistant Express VI is available in the Instrument I/O category of the Functions palette. The **Instrument I/O Assistant** configuration dialog box appears. If it does not appear, double-click the Instrument I/O Assistant icon. Complete the following steps to configure the Instrument I/O Assistant.

1. Select an instrument. Instruments that have been configured in MAX appear in the **Select an instrument** pull-down menu.
2. Choose a **Code generation type**. VISA code generation allows for more flexibility and modularity than GPIB code generation.
3. Select from the following communication steps using the **Add Step** button:
 - **Query and Parse**—Sends a query to the instrument, such as *IDN? and parses the returned string. This step combines the Write command and Read and Parse command.
 - **Write**—Sends a command to the instrument.
 - **Read and Parse**—Reads and parses data from the instrument
4. After adding the desired number of steps, click the **Run** button to test the sequence of communication that you have configured for the Express VI.
5. Click the **OK** button to exit the **Instrument I/O Assistant** configuration dialog box.

LabVIEW adds input and output terminals to the Instrument I/O Assistant Express VI on the block diagram that correspond to the data you receive from the instrument.

To view the code generated by the Instrument I/O Assistant, right-click the Instrument I/O Assistant icon and select **Open Front Panel** from the

shortcut menu. This converts the Express VI to a subVI. Switch to the block diagram to see the code generated.



Note After you convert an Express VI to a subVI, you cannot reconvert the Express VI.

Exercise 9-2 Concept: Instrument I/O Assistant

Goal

Configure a serial or GPIB instrument and communicate with the instrument using the Instrument I/O Assistant.

For serial, follow the instructions in part A of this exercise.

For GPIB, follow the instructions in part B of this exercise.

Part A: Serial Description

1. Configure the NI Instrument Simulator to communicate through the serial port.
 - Power off the NI Instrument Simulator.
 - Set the left bank of switches on the side of the box to match Figure 9-6.

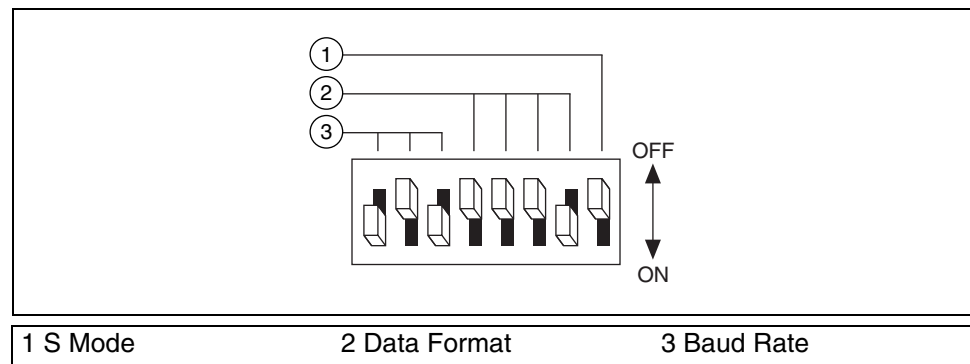


Figure 9-6. Serial Configuration Settings for the NI Instrument Simulator

- Make sure the NI Instrument Simulator is connected to a serial port on the computer with a serial cable.
- Make a note of the port number.
- Power on the NI Instrument Simulator.
- Verify that the Power, Ready, and Listen LEDs are lit to indicate that the device is in serial communication mode.

You build a block diagram similar to the one in Figure 9-7 in the following steps.

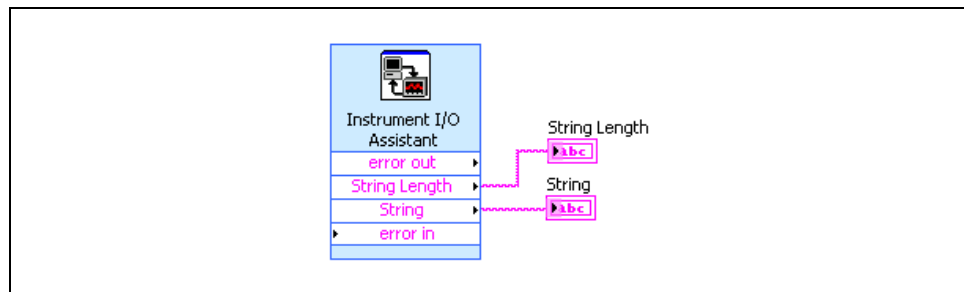


Figure 9-7. IIOASerial VI Block Diagram

2. Open a blank VI.
3. Save the VI as `Serial IIOA Read.vi` in the `C:\Exercises\LabVIEW Basics I\Instrument IO Assistant` directory.
4. Open the block diagram.
5. Configure the Instrument I/O Express VI to communicate with the NI Instrument Simulator.
6. Place the Instrument I/O Express VI on the block diagram. The **Instrument I/O Assistant** dialog box appears.



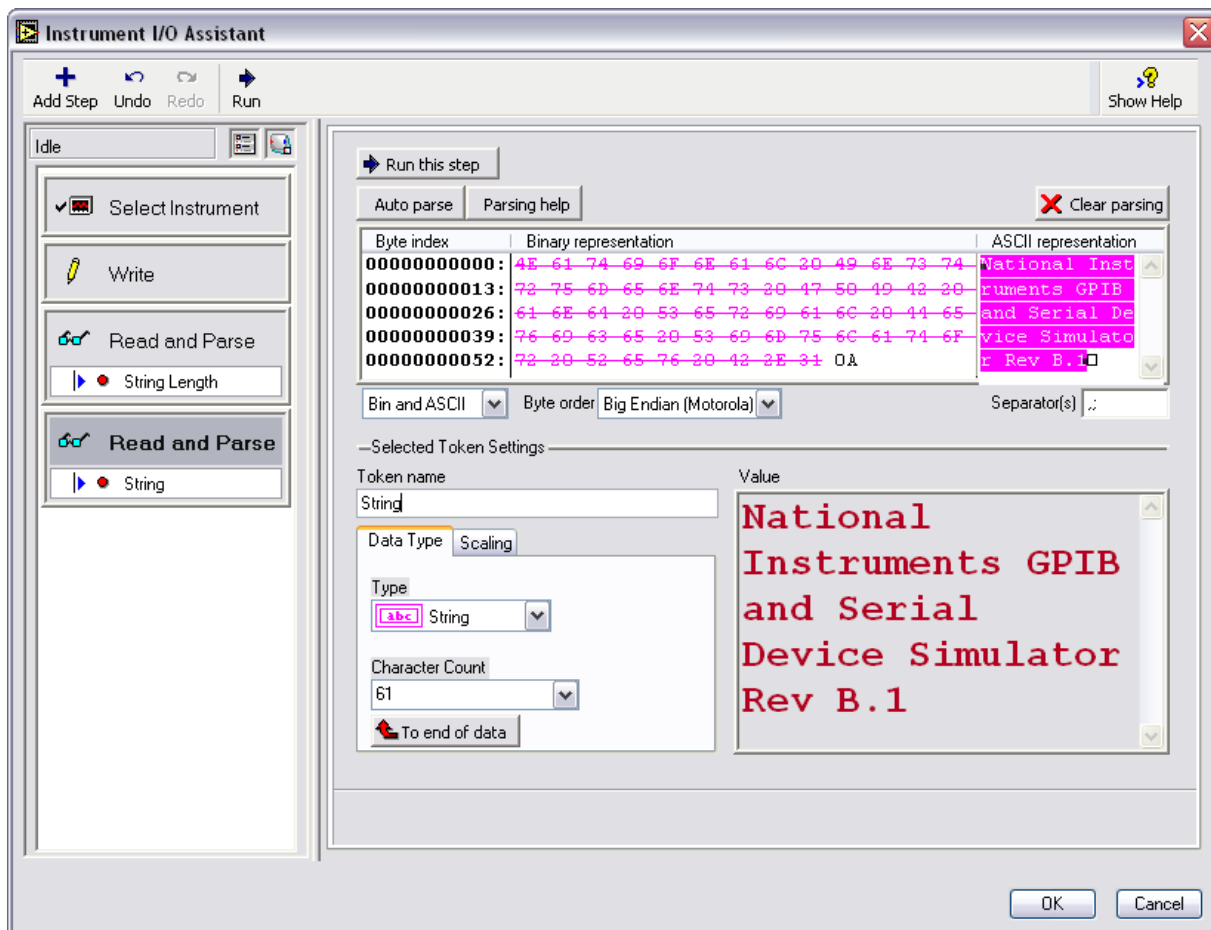


Figure 9-8. Serial Configuration of the Instrument I/O Assistant

- From the **Select an instrument** pull-down menu, choose **COM1** (or **COM2** depending on the connection port of the NI Instrument Simulator).
- Click the **Add Step** button.
- Click **Write**.
- In the command field, enter `*IDN?`.
- Click the **Add Step** button.
- Click **Read and Parse**.



Note The Instrument Simulator returns the byte size of the response, the termination character, the response, then another termination character. Therefore, after `*IDN?` is sent to the instrument, the response must be read twice; once to retrieve the size of the response, and once to retrieve the response.

- Click the **Add Step** button.
 - Click **Read and Parse** again.
 - Click the **Run** button (not the **Run this step** button). The **Run** button runs the entire sequence.
 - Return to the first **Read and Parse** step.
 - Click the **Auto parse** button. The value returned is the size in bytes of the query response.
 - Rename `Token` to `String Length` in the **Token name** text box.
 - Select the second **Read and Parse** step.
 - Click the **Auto parse** button. The value returned is the identification string of the NI Instrument Simulator.
 - Rename `Token` to `String` in the **Token name** text box. The configuration window should be similar to Figure 9-8.
 - Select **OK** to return to the block diagram.
7. Create an indicator for the response from the instrument.
 - Right-click the **String** terminal.
 - Select **Create»Indicator** from the shortcut menu.
 8. Create an indicator for the response length from the instrument.
 - Right-click the **String Length** terminal.
 - Select **Create»Indicator** from the shortcut menu.



Tip To allow LabVIEW to handle errors automatically, do not connect a Simple Error Handler VI to **error out**.

9. Display the front panel. It should be similar to Figure 9-9.

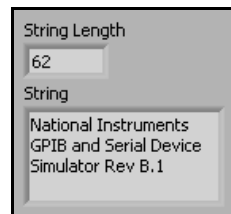


Figure 9-9. IIOASerial VI Front Panel

10. Save the VI.
11. Run the VI.
12. Examine the code generated by the I/O Assistant.
 - Right-click the I/O Assistant and select **Open Front Panel**.
 - Click the **Convert** button when asked if you want to convert to a subVI.
 - View the code generated by the I/O Assistant. Where is the command `*IDN?` written to the Instrument Simulator? Where is the voltage being read?
 - Select **File»Exit** to exit the subVI. Do not save changes.
13. Close the VI when finished.

Part B: GPIB Description

1. Configure the NI Instrument Simulator to communicate through the GPIB interface.
 - Power off the NI Instrument Simulator.
 - Set the left bank of switches on the side of the box to match Figure 9-10.

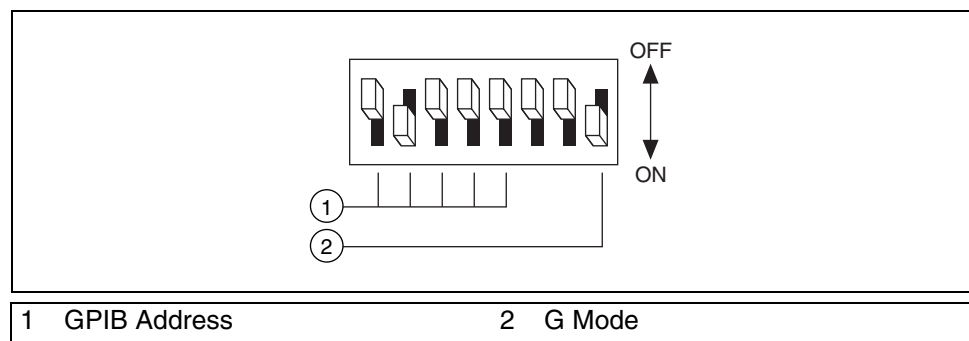


Figure 9-10. GPIB Configuration Settings for the NI Instrument Simulator

- Make sure the NI Instrument Simulator is connected to the GPIB board.
- Power on the NI Instrument Simulator.
- Verify that both the Power and Ready LEDs are lit.

2. Open a blank VI.
3. Save the VI as GPIB IIOA Read.vi in the C:\Exercises\LabVIEW Basics I\Instrument IO Assistant directory.

You build a block diagram similar to the one in Figure 9-11 in the following steps.

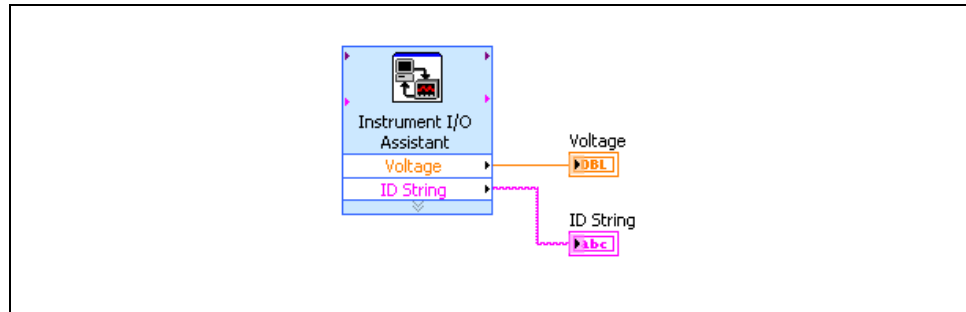


Figure 9-11. IIOAGPIB VI Block Diagram

4. Open the block diagram.
5. Configure the Instrument I/O Express VI to communicate with the NI Instrument Simulator.
6. Place the Instrument I/O Express VI on the block diagram. The **Instrument I/O Assistant** dialog box appears.



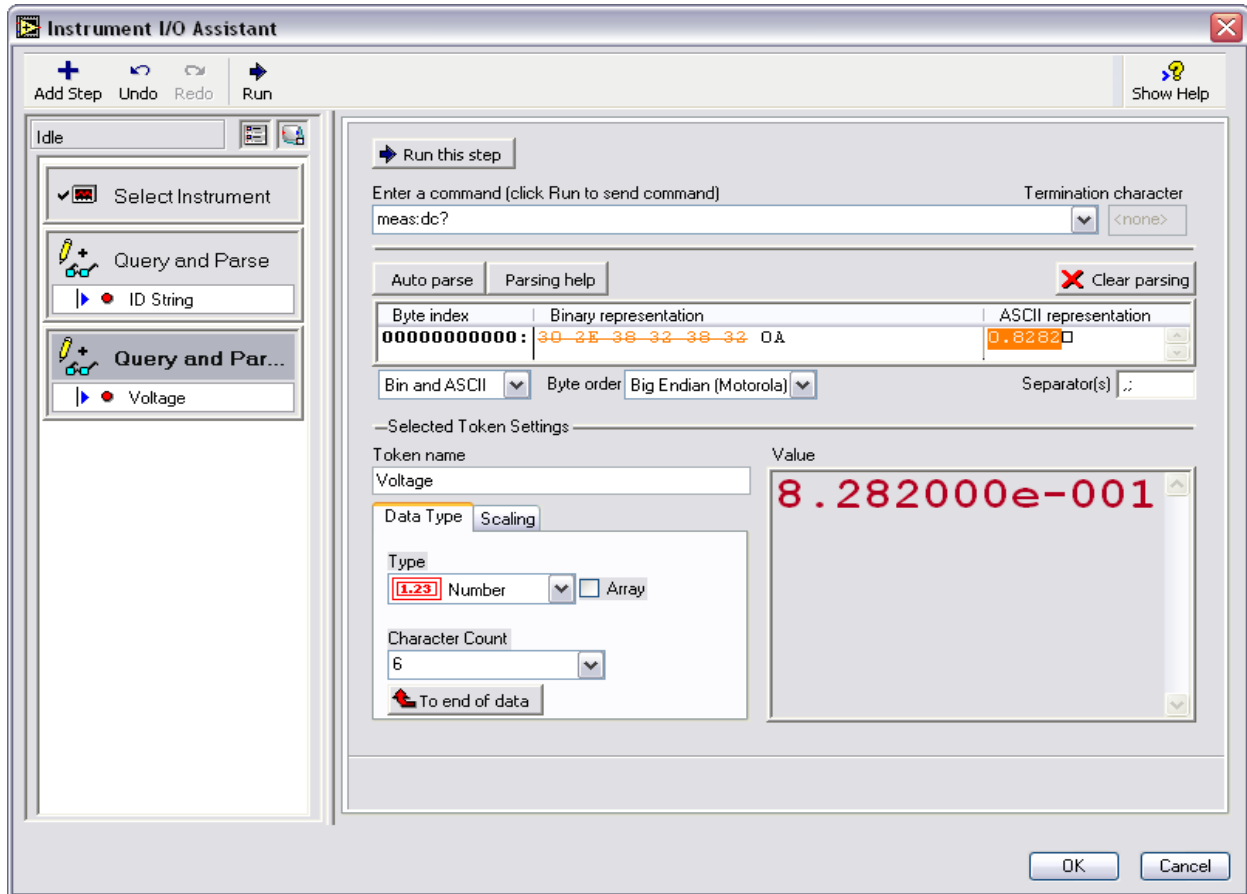


Figure 9-12. GPIB Configuration of the Instrument I/O Assistant

- Select **devsim** from the **Select an instrument** pull-down menu.
- Select **VISA Code Generation** from the **Code generation type** pull-down menu.
- Click the **Add Step** button.
- Click **Query and Parse** to write and read from the Instrument Simulator.
- Enter `*IDN?` as the command.
- Click the **Run this step** button. If no error warning appears in the lower half of the dialog box, this step has successfully completed.
- To parse the data received, click the **Auto parse** button.
- Rename Token by typing ID String in the **Token name** text box.
- Click the **Add Step** button.

- Click **Query and Parse**.
 - Enter `MEAS:DC?` as the command.
 - Click the **Run this step** button.
 - To parse the data received, click the **Auto parse** button. The data returned is a random numeric value.
 - Rename `Token` by typing `Voltage` in the **Token name** text box. The configuration window should be similar to Figure 9-12.
 - Click the **OK** button to exit the I/O Assistant and return to the block diagram.
7. Create an indicator for the ID String.
 - Right-click the **ID String** terminal and select **Create»Indicator** from the shortcut menu.
 8. Create an indicator for the voltage.
 - Right-click the **Voltage** terminal and select **Create»Indicator** from the shortcut menu.



Tip To allow LabVIEW to handle errors automatically, do not connect a Simple Error Handler VI to **error out**.

9. Display the front panel. The front panel should be similar to the front panel in Figure 9-13.

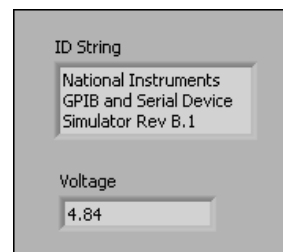


Figure 9-13. IIOAGPIB VI Front Panel

10. Save the VI.
11. Run the VI. Resize the string indicator if necessary.
12. Examine the code generated by the I/O Assistant.
 - Right-click the I/O Assistant and select **Open Front Panel**.

- Click the **Convert** button when prompted to convert to a subVI.
- View the code generated by the I/O Assistant. Where is the command `*IDN?` written to the Instrument Simulator? Where is the voltage being read?
- Select **File»Exit** to exit the subVI. Do not save changes.

13. Close the VI when finished.

End of Exercise 9-2

G. VISA

Virtual Instrument Software Architecture (VISA) is the lower layer of functions in the LabVIEW instrument driver VIs that communicates with the driver software. VISA by itself does not provide instrumentation programming capability. VISA is a high-level API that calls low-level drivers. VISA can control VXI, GPIB, serial, or computer-based instruments and makes the appropriate driver calls depending on the type of instrument used. When debugging VISA problems, remember that an apparent VISA problem could be an installation problem with one of the drivers that VISA calls.

In LabVIEW, VISA is a single library of functions you use to communicate with GPIB, serial, VXI, and computer-based instruments. You do not need to use separate I/O palettes to program an instrument. For example, some instruments give you a choice for the type of interface. If the LabVIEW instrument driver were written with functions on the **Functions»All Functions»Instrument I/O»GPIB** palette, those instrument driver VIs would not work for the instrument with the serial port interface. VISA solves this problem by providing a single set of functions that work for any type of interface. Therefore, many LabVIEW instrument drivers use VISA as the I/O language.

VISA Programming Terminology

The following terminology is similar to that used for instrument driver VIs:

- **Resource**—Any instrument in the system, including serial and parallel ports.
- **Session**—You must open a VISA session to a resource to communicate with it, similar to a communication channel. When you open a session to a resource, LabVIEW returns a VISA session number, which is a unique refnum to that instrument. You must use the session number in all subsequent VISA functions.
- **Instrument Descriptor**—Exact name of a resource. The descriptor specifies the interface type (GPIB, VXI, ASRL), the address of the device (logical address or primary address), and the VISA session type (INSTR or Event).

The instrument descriptor is similar to a telephone number, the resource is similar to the person with whom you want to speak, and the session is similar to the telephone line. Each call uses its own line, and crossing these lines results in an error. Table 9-1 shows the proper syntax for the instrument descriptor.

Table 9-1. Syntax for Various Instrument Interfaces

Interface	Syntax
Asynchronous serial	ASRL [board] [: : INSTR]
GPIB	GPIB [board] : : primary address [: : secondary address] [: : INSTR]
VXI instrument through embedded or MXIbus controller	VXI [board] : : VXI logical address [: : INSTR]
GPIB-VXI controller	GPIB-VXI [board] [: : GPIB-VXI primary address] : : VXI logical address [: : INSTR]

You can use an alias you assign in MAX instead of the instrument descriptor. **(Mac OS)** Edit the `visaconf.ini` file to assign a VISA alias. **(UNIX)** Use the `visaconf` utility.

If you choose not to use the Instrument I/O Assistant to automatically generate code for you, you can still write a VI to communicate with the instrument. The most commonly used VISA communication functions are the VISA Write and VISA Read functions. Most instruments require you to send information in the form of a command or query before you can read information back from the instrument. Therefore, the VISA Write function is usually followed by a VISA Read function. The VISA Write and VISA Read functions work with any type of instrument communication and are the same whether you are doing GPIB or serial communication. However, because serial communication requires you to configure extra parameters, you must start the serial port communication with the VISA Configure Serial Port VI.

VISA and Serial

The VISA Configure Serial Port VI initializes the port identified by **VISA resource name** to the specified settings. **Timeout** sets the timeout value for the serial communication. **Baud rate**, **data bits**, **parity**, and **flow control** specify those specific serial port parameters. The **error in** and **error out** clusters maintain the error conditions for this VI.

Figure 9-14 shows how to send the identification query command `*IDN?` to the instrument connected to the COM2 serial port. The VISA Configure Serial Port VI opens communication with COM2 and sets it to 9,600 baud, eight data bits, odd parity, one stop bit, and XON/XOFF software handshaking. Then, the VISA Write function sends the command. The VISA Read function reads back up to 200 bytes into the read buffer, and the Simple Error Handler VI checks the error condition.

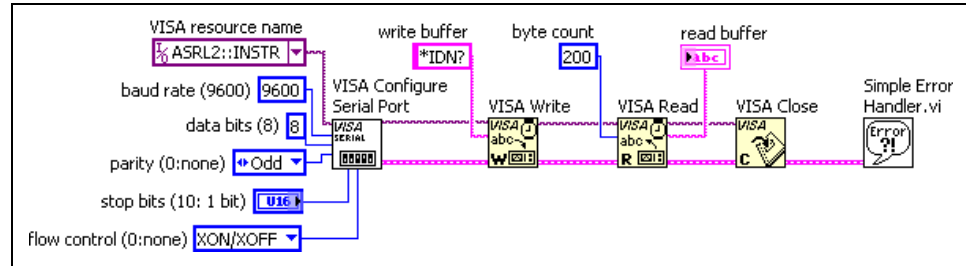


Figure 9-14. Configuring serial for VISA example



Note The VIs and functions located on the **Functions»All Functions»Instrument I/O»Serial** palette are also used for parallel port communication. You specify the VISA resource name as being one of the LPT ports. For example, you can use MAX to determine that LPT1 has a VISA resource name of ASRL10::INSTR.

Exercise 9-3 VISA Write & Read VI

Goal

Communicate with a serial or GPIB interface to an instrument using VISA functions.

Description

This VI uses VISA to communicate with either a serial or a GPIB interface to an instrument. The VI can send one buffer of data to the instrument, and read one buffer back. If using GPIB, the user specifies how many bytes to read from the bus. If using serial, the VI determines how many bytes are available, and reads them all.

1. Open the `VISA Write & Read.vi` in the `C:\Exercises\LabVIEW Basics I\VISA Write & Read` directory.

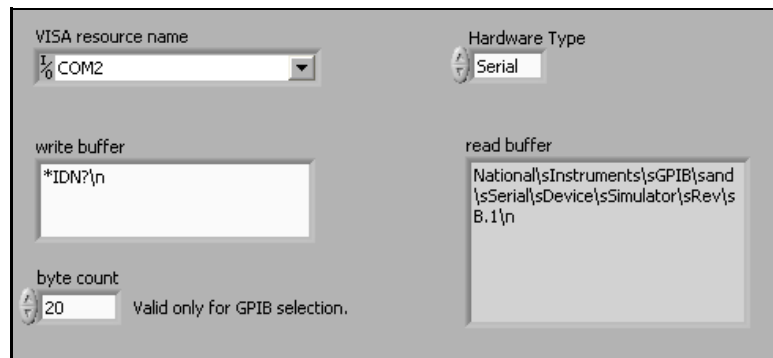


Figure 9-15. VISA Write & Read VI Front Panel

2. Open the block diagram of the VI and examine the code. The GPIB portion is shown in Figure 9-16.

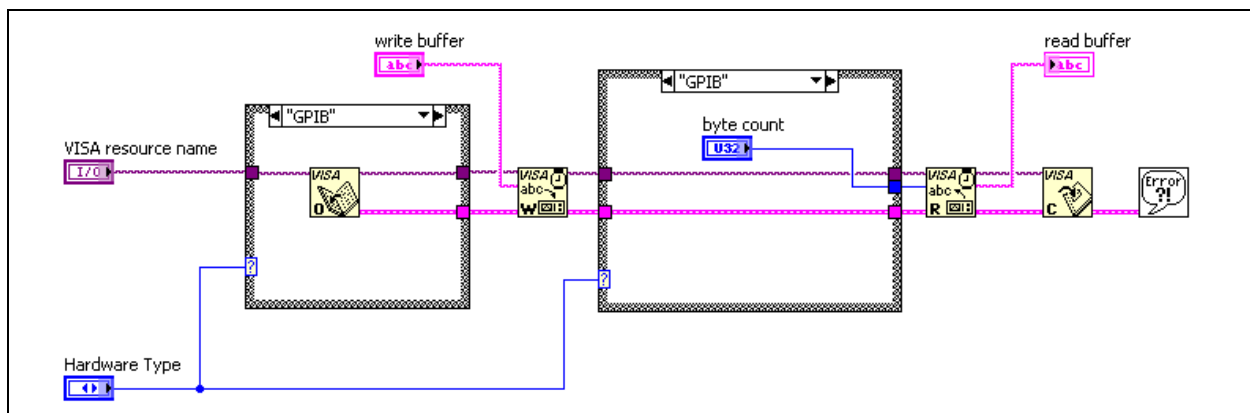


Figure 9-16. GPIB portion of the VISA Write & Read VI Block Diagram

Follow the instructions in the *Test A: Serial* section to communicate through the serial port. Follow the instructions in the *Test B: GPIB* section to communicate through the GPIB port.

Test A: Serial

1. Configure the NI Instrument Simulator to communicate through the serial port. It may still be set up from the last exercise.
 - Power off the NI Instrument Simulator.
 - Set the left bank of switches on the side of the box to match Figure 9-17.

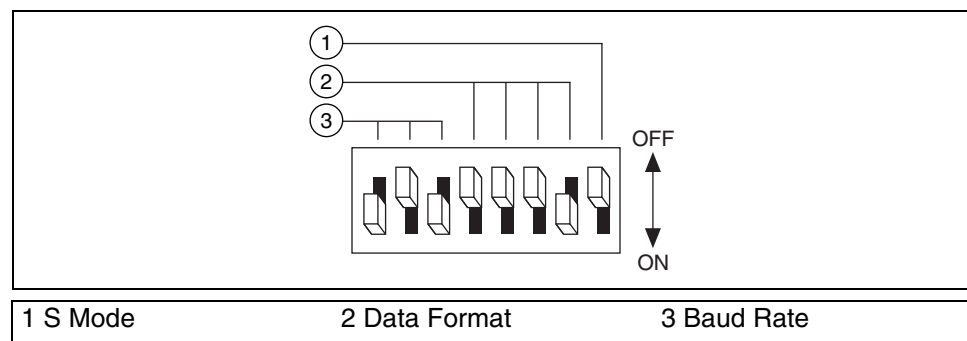


Figure 9-17. Serial Configuration Settings for the NI Instrument Simulator

- Make sure the NI Instrument Simulator is connected to a serial port.
 - Power on the NI Instrument Simulator.
 - Verify that the Power, Ready and Listen LEDs are lit.
2. Enter values into the controls in preparation for communicating with the instrument. You do not need to enter a value in the byte count, as this control is only used for GPIB communication.
 - Select the serial port in the **VISA resource name** control.
 - Select Serial from the Hardware Type enumerated control.
 - Enter *IDN? in the **write buffer**.
 3. Run the VI.
 4. The top of the instrument simulator lists other commands that are recognized by this instrument. Try other commands in this VI.
 5. Close the VI when finished.

Test B: GPIB

1. Configure the NI Instrument Simulator to communicate through the GPIB interface.
 - Power off the NI Instrument Simulator.
 - Set the left bank of switches on the side of the box to match Figure 9-18.

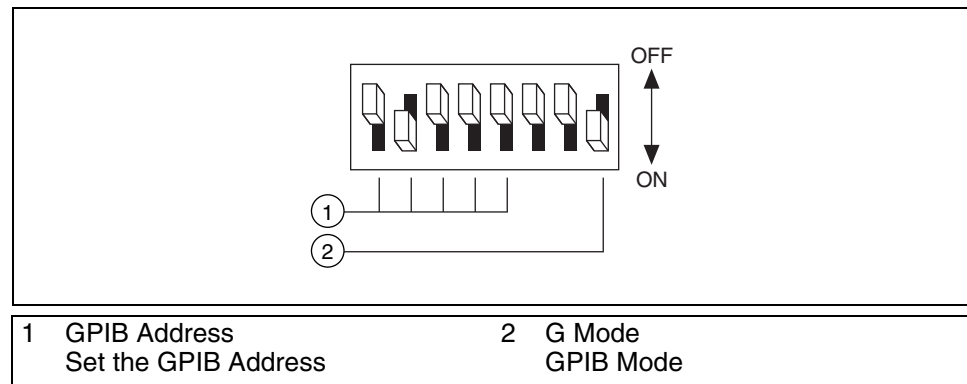


Figure 9-18. GPIB Configuration Settings for the NI Instrument Simulator

- Make sure the NI Instrument Simulator is connected to the GPIB board.
 - Power on the NI Instrument Simulator.
 - Verify that both the Power and Ready LEDs are lit.
2. Enter values into the controls in preparation for communicating with the instrument.
 - Select devsim in the **VISA resource name** control.
 - Select GPIB from the Hardware Type enumerated control.
 - Enter *IDN? in the **write buffer**.
 3. Run the VI.
 4. The top of the instrument simulator lists other commands that are recognized by this instrument. Try other commands in this VI.
 5. Close the VI when finished.

End of Exercise 9-3

H. Instrument Drivers

Imagine the following scenario. You wrote a LabVIEW VI that communicates with a specific oscilloscope in your lab. Unfortunately, the oscilloscope no longer works, and you must replace it. However, this particular oscilloscope is no longer made. You found a different brand of oscilloscope that you want to purchase, but your VI no longer works with the new oscilloscope. You must rewrite your VI.

When you use an instrument driver, the driver contains the code specific to the instrument. Therefore, if you change instruments, you must replace only the instrument driver VIs with the instrument driver VIs for the new instrument, which greatly reduces your redevelopment time. Instrument drivers help make test applications easier to maintain because the drivers contain all the I/O for an instrument in one library, separate from other code. When you upgrade hardware, upgrading the application is easier because the instrument driver contains all the code specific to that instrument.

What Is an Instrument Driver?

A LabVIEW Plug and Play instrument driver is a set of VIs that control a programmable instrument. Each VI corresponds to an instrument operation, such as configuring, triggering, and reading measurements from the instrument. Instrument drivers help users get started using instruments from a PC and saves them development time and cost because users do not need to learn the programming protocol for each instrument. With open-source, well documented instrument drivers, end users can customize their operation for better performance. A modular design makes the driver easier to customize.

Where Do I Find Instrument Drivers?

You can locate most LabVIEW Plug and Play instrument driver in the Instrument Driver Finder. You can access the Instrument Driver Finder within LabVIEW by selecting **Tools»Instrumentation»Find Instrument Drivers** or **Help»Find Instrument Drivers**. The Instrument Driver Finder connects you with ni.com to find instrument drivers. When you install an instrument driver, an example program using the driver is added to the NI Example Finder.

Example Instrument Driver VI

The block diagram in Figure 9-19 initializes the Agilent 34401 digital multimeter (DMM), uses a configuration VI to choose the resolution and range, select the function, and enable or disable auto range, uses a data VIs to read a single measurement, closes the instrument, and checks the error status. Every application that uses an instrument driver has a similar sequence of events: Initialize, Configure, Data, and Close.

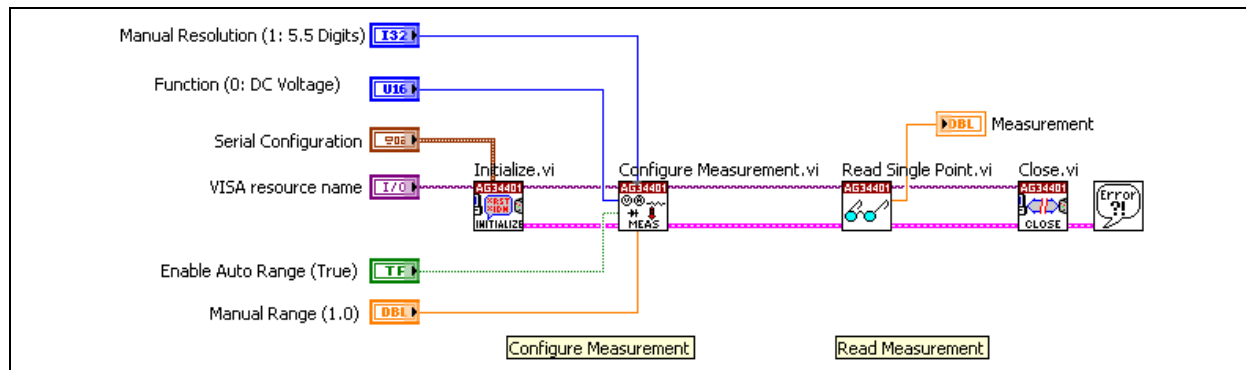


Figure 9-19. Agilent 34401 DMM Instrument Driver Example

This is an example program that is available in the NI Example Finder when you install the LabVIEW Plug and Play instrument driver for the Agilent 34401 DMM.

How Do Instrument Drivers Work?

Many programmable instruments have a large number of functions and modes. With this complexity, it is necessary to provide a consistent design model that aids both instrument driver developers as well as end users who develop instrument control applications. The LabVIEW Plug and Play instrument driver model contains both external structure and internal structure guidelines. The external structure shows how the instrument driver interfaces with the user and to other software components in the system. The internal structure shows the internal organization of the instrument driver software module.

For the external structure of the instrument driver, the user interacts with the instrument driver using an API or an interactive interface. Usually, the interactive interface is used for testing or for end-users. The API is accessed through LabVIEW. The instrument driver communicates with the instrument using VISA.

Internally, the VIs in an instrument driver are organized into six categories. These categories are summarized in the following table.

Category	Description
Initialize	The initialize VI establishes communication with the instrument and is the first instrument driver VI called.

Configure	This collection of VIs are software routines that configure the instrument to perform specific operations. After calling these VIs, the instrument is ready to take measurements or stimulate a system.
Action/Status	This collection of VIs command the instrument to carry out an action (i.e. arming a trigger) or obtain the current status of the instrument or pending operations.
Data	The data VIs transfer data to or from the instrument.
Utility	THis collection of VIs perform a variety of auxiliary operations, such as reset and self-test.
Close	The close VI terminates the software connection to the instrument. This is the last instrument driver VI called.

Exercise 9-4 Concept: Instrument Driver

Goal

Install an instrument driver and explore the example programs that accompany the instrument driver.

Description

In this exercise, you install the instrument driver the NI Instrument Simulator. After installation, you explore the VIs that the instrument driver provides and the example programs that are added to the NI Example Finder.

Install Instrument Driver

1. Exit LabVIEW.
2. Navigate to the `C:\Exercises\LabVIEW Basics I\Instrument Driver` directory. This folder contains the LabVIEW Plug and Play instrument driver for the Instrument Simulator.
3. Double-click the NI Instrument Simulator Zip folder to extract the contents.
4. Copy the folder that was extracted.
5. Navigate to the `C:\Program Files\National Instruments\LabVIEW 8.0\instr.lib` directory.
6. Paste the copied folder in this directory.

Explore Instrument Driver

7. Start LabVIEW.
8. Open a blank VI.
9. Switch to the block diagram.
10. Navigate to the **Instrument I/O»Instrument Drivers»NI Instrument Simulator** category of the **Functions** palette.
11. Explore the palette using the **Context Help** window to familiarize yourself with the functionality.

Use Example Programs

12. Select **Help»Find Examples** to start the NI Example Finder.

13. Confirm that you are browsing according to task.
14. Navigate to **Hardware Input and Output»Instrument Drivers»LabVIEW Plug and Play** in the task structure.
15. Double-click **NI Instrument Simulator Read DMM Measurement.vi** to open the example program. This VI reads a single measurement from the Instrument Simulator.
16. Prepare the Instrument Simulator. This VI can communicate with the instrument through serial or GPIB.
 - To communicate through serial, set the Instrument Simulator switches as shown in Figure 9-20.

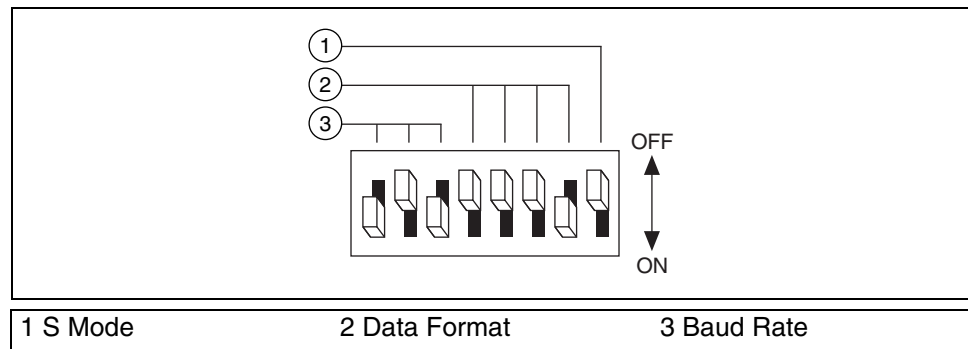


Figure 9-20. Serial Configuration Settings for the NI Instrument Simulator

- To communicate through GPIB, set the Instrument Simulator switches as shown in Figure 9-21.

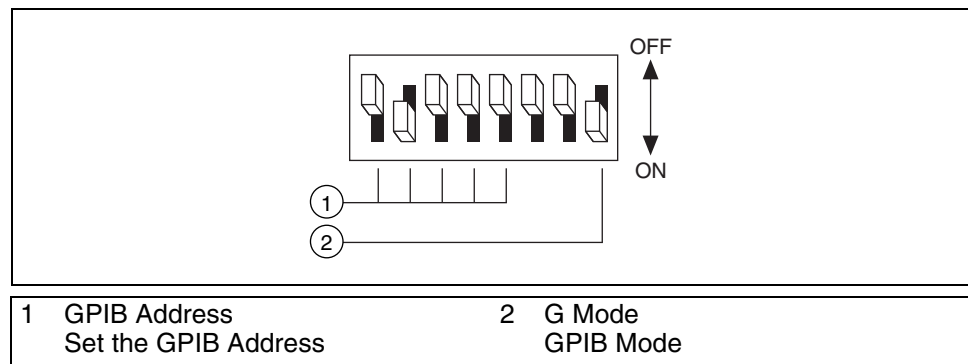


Figure 9-21. GPIB Configuration Settings for the NI Instrument Simulator

17. Select the communication type on the VISA Resource Name control.
 - If you are using serial, select the resource (COM1 or COM2) that the serial cable is connected to.

- If you are using GPIB, select the devsim VISA alias.
- 18. Run the VI.
- 19. Explore the block diagram of the VI. Do not save changes.
- 20. Close the VI.
- 21. Return to the NI Example Finder.
- 22. Double-click **NI Instrument Simulator Read Oscilloscope Waveform.vi** to open the next example program. This VI read a single waveform from the Instrument Simulator.
- 23. Select the same VISA Resource Name you selected in Step 17.
- 24. Run the VI.
- 25. Choose a different Waveform Function.
- 26. Run the VI again.
- 27. Explore the block diagram of the VI.
- 28. Close the VI and the NI Example Finder when you are finished. Do not save changes.

End of Exercise 9-4

Self Review: Quiz

1. Which instrument interface does not use the VISA API?
 - a. Serial
 - b. Parallel
 - c. GPIB
 - d. Ethernet
2. What API does the Instrument I/O Assistant use?
 - a. C
 - b. Visual Basic
 - c. VISA
 - d. NI-DAQmx
3. Which of the following is a way to inform listeners that all data has been transferred?
 - a. Asserting the End or Identify (EOI) line.
 - b. Placing a end-of-string (EOS) character at the beginning of the data being transferred.
 - c. Using the VISA Close function.
 - d. Turning off the power to the controller.

Self Review: Quiz Answers

1. Which instrument interface does not use the VISA API?
 - a. Serial
 - b. Parallel**
 - c. GPIB
 - d. Ethernet
2. What API does the Instrument I/O Assistant use?
 - a. C
 - b. Visual Basic
 - c. VISA**
 - d. NIDAQmx
3. Which of the following is a way to inform listeners that all data has been transferred?
 - a. Asserting the End or Identify (EOI) line.**
 - b. Placing a end-of-string (EOS) character at the beginning of the data being transferred.
 - c. Using the VISA Close function.
 - d. Turning off the power to the controller.

Notes

Analyzing and Storing Measurement Data

You have learned how to acquire data and how to display it, but analysis and storage of your data is usually a very important part of any project. In this lesson, you learn about manipulating strings, analyzing numerical data, processing signals, and storing ASCII data.

Topics

- A. Analyzing and Processing Numeric Data
- B. Reading and Writing Data to File

A. Analyzing and Processing Numeric Data

Users generally start their work by acquiring data into an application or program, because their tasks typically require interaction with physical processes. In order to extract valuable information from that data, make decisions on the process, and obtain results, the data needs to be manipulated and analyzed.

As an engineering-focused tool, LabVIEW includes hundreds of analysis functions. You can build these functions right into your applications to make intelligent measurements and obtain results faster.

Choosing the Correct Method for Analysis

Users incorporate analysis into their applications and programs in different ways. There are certain considerations that help determine the way in which analysis should be performed.

Inline vs. Offline Analysis

Inline analysis implies that you analyze the data within the same application where you acquired it. This is generally the case when dealing with applications where decisions have to be made during run time and the results have direct consequences on the process, typically through the changing of parameters or executing of actions. This is typically the case in control applications. When dealing with inline analysis, it is important to consider the amount of data acquired and the particular analysis routines that are performed on that data. A proper balance must be found because they could easily become computationally intensive and have an adverse effect on the performance of the application.

Other examples for inline analysis are applications where the parameters of the measurement need to be adapted to the characteristics of the measured signal. One case is where one or more signals need to be logged, but these change very slowly except for sudden bursts of high-speed activity. In order to reduce the amount of data logged, the application would have to quickly recognize the need for a higher sampling rate, and reduce it when the burst is over. By measuring and analyzing certain aspects of the signal, the application can adapt to the circumstances and enable the appropriate execution parameters. Although this is only one example, there are thousands of applications where a certain degree of intelligence – the ability to make decisions based on various conditions – and adaptability are required, which can only be provided by adding analysis algorithms to the application.

Decisions based on acquired data are not always made in an automated manner. Very frequently, those involved with the process need to monitor the execution and determine whether it is performing as expected or if one

or more variables need to be adjusted. Although it is not uncommon for users to log data, extract it from a file or database and then analyze it offline to modify the process, many times the changes need to happen during run time. In these cases, the application must handle the data coming from the process, and then manipulate, simplify, format, and present the data in a way that it is most useful to the user.

LabVIEW offers analysis routines for point-by-point execution; these routines are designed specifically to meet the needs of inline analysis in real-time applications. Point-by-point analysis is essential when dealing with control processes where high-speed, deterministic, point-by-point data acquisition is present. Any time resources are dedicated to real-time data acquisition, point-by-point analysis becomes a necessity as acquisition rates and control loops are increased by orders of magnitude. The point-by-point approach simplifies the design, implementation, and testing process, because the flow of the application closely matches the natural flow of the real-world processes that the application is monitoring and controlling. Point-by-point analysis is streamlined and stable, because it ties directly into the acquisition and analysis process.

Use offline analysis when the results do not need to be obtained in real-time fashion in order to make decisions on the process. Offline analysis applications require only that sufficient computational resources are available. The main intent of such applications is to identify cause and effect of variables affecting a process by correlating multiple data sets. These applications generally require importing data from custom binary or ASCII files and commercial databases such as Oracle, Access, and other SQL/ODBC-enabled databases. After the data is imported into LabVIEW, users perform several or hundreds of available analysis routines, manipulate the data, and arrange it in a specific format for reporting purposes.

Programmatic vs. Interactive Analysis

Acquiring data and processing it for the sake of online visualization is not enough. Users typically store hundreds or thousands of megabytes of data in hard drives and data bases. After anywhere from one to hundreds of runs of the application, users proceed to extract information in order to make decisions, compare results, and make appropriate changes to the process, until the desired results are achieved. It is relatively easy to acquire amounts of data so large that it rapidly becomes unmanageable. In fact, with a fast DAQ board and enough channels, it may only take a few milliseconds to compile thousands of values. It is not a trivial task to make sense out of all that data. Engineers and scientists are typically expected to present reports, create graphs, and ultimately corroborate any assessments and conclusions with empirical data.

In order to simplify the process of analyzing measurements, you can create applications that provide dialogs and interfaces that others can use so that depending on their input, specific analysis routines are performed on any given data set. By building this type of application, users build a certain degree of interactivity into their applications. For this to be efficient, you must have extensive knowledge about the information and the types of analysis in which the user is interested.

You can also perform significant data reduction and formatting before storing it to disk, so that when the stored data is retrieved for further analysis, it is easier to handle.

Analysis Categories

LabVIEW offers hundreds of built-in analysis functions that cover different areas and methods of extracting information from acquired data. You can use these functions as is, or modify, customize, and extend them to suit a particular need. These functions are categorized in the following groups: Measurement, Signal Processing, Mathematics, Image Processing, Control, Simulation, and Application Areas.

- Measurement
 - Amplitude and Level
 - Frequency (Spectral) Analysis
 - Noise and Distortion
 - Pulse and Transition
 - Signal and Waveform Generation
 - Time Domain Analysis
 - Tone Measurements
- Signal Processing
 - Digital Filters
 - Convolution and Correlation
 - Frequency Domain
 - Joint Time-Frequency Analysis (Signal Processing Toolset)
 - Sampling/Resampling
 - Signal Generation
 - Super-Resolution Spectral Analysis (Signal Processing Toolset)
 - Transforms
 - Time Domain
 - Wavelet and Filter Bank Design (Signal Processing Toolset)
 - Windowing

- Mathematics
 - Basic Math
 - Curve Fitting and Data Modeling
 - Differential Equations
 - Interpolation and Extrapolation
 - Linear Algebra
 - Nonlinear Systems
 - Optimization
 - Root Finding
 - Special Functions
 - Statistics and Random Processes
- Image Processing
 - Blob Analysis and Morphology
 - Color Pattern Matching
 - Filters
 - High-Level Machine Vision Tools
 - High-Speed Grayscale Pattern Matching
 - Image Analysis
 - Image and Pixel Manipulation
 - Image Processing
 - Optical Character Recognition
 - Region-of-Interest Tools
- Control
 - PID and Fuzzy Control
 - Simulation
 - Simulation Interface (Simulation Interface Toolkit)
- Application Areas
 - Machine Condition Monitoring (Order Analysis Toolset)
 - Machine Vision (IMAQ, Vision Builder)
 - Motion Control
 - Sound and Vibration (Sound and Vibration Analysis Toolset)

For a complete list of LabVIEW analysis functions refer to ni.com/analysis.

Exercise 10-1 Concept: Analysis Types

Goal

Choose when to use inline, offline, programmatic or interactive analysis for an application.

Description

For each scenario, circle which forms of analysis to use. Most scenarios use more than one form.

Scenario 1

The failure rate of your manufacturing line is related directly to the speed of production. Monitor the failure rate programmatically. If the failure rate is greater than 3 %, decrease the speed of the line. If the failure rate is less than 2 %, increase the speed of the line.

Inline Offline Programmatic Interactive

Scenario 2

You are listening to a radio station. The frequency components of the radio station signal are determined and recorded to file. If you have difficulty hearing the radio station, you tell the VI to pass the signal through a filter before recording the data.

Inline Offline Programmatic Interactive

Scenario 3

You are recording temperature and pressure data. Once a week, you prepare a report for your manager correlating the temperature and pressure trends during thunderstorms.

Inline Offline Programmatic Interactive

Scenario 4

You are performing stress analysis on a bridge. During rush hour, you must also record vibration data on the bridge. It is considered rush hour when more than 100 cars use the bridge in 5 minutes. A sensor records the number of cars crossing the bridge.

Inline Offline Programmatic Interactive

Refer to the following page for answers to these scenarios.

Scenario 1

- Inline Analysis
- Programmatic Analysis

Inline analysis occurs to determine the speed of the line and the failure rate. Programmatic analysis occurs to determine when to change the speed of the line.

Scenario 2

- Inline Analysis
- Interactive Analysis

The user tells the VI when to apply the filter, which means the analysis is interactive. However, because the filtering happens immediately when the user specifies, the analysis is inline.

Scenario 3

- Offline Analysis

The data can be correlated at any point; does not need to occur as the data is acquired. When it is analyzed, it is most likely be analyzed programmatically. However, without more information, you cannot determine whether programmatic or interactive analysis is appropriate.

Scenario 4

- Programmatic Analysis

The VI uses the sensor to determine when rush hour is occurring and immediately begins recording the additional data. Since no information is given on how the data is analyzed, cannot determine whether inline or offline analysis is appropriate.

End of Exercise 10-1

B. Reading and Writing Data to File

File I/O records or reads data in a file.

A typical file I/O operation involves the following process.

1. Create or open a file. Indicate where an existing file resides or where you want to create a new file by specifying a path or responding to a dialog box to direct LabVIEW to the file location. After the file opens, a refnum represents the file.

Refer to the *References to Objects or Applications* section of Chapter 4, *Building the Front Panel*, for more information about refnums.

2. Read from or write to the file.
3. Close the file.

File Formats

LabVIEW can use or create the following file formats: Binary, ASCII, LVM, and TDM.

- **Binary**— Binary files are the underlying file format of all other file formats.
- **ASCII**—An ASCII file is a specific type of binary file that is a standard used by most programs. It consists of a series of ASCII codes. ASCII files are also called text files.
- **LVM**— The LabVIEW measurement data file (`.lvm`) is a tab-delimited text file you can open with a spreadsheet application or a text-editing application. The `.lvm` file includes information about the data, such as the date and time the data was generated. This file format is a specific type of ASCII file created for LabVIEW.
- **TDM**—This file format is a specific type of binary file created for National Instruments products. It actually consists of two separate files: an XML section contains the data attributes, and a binary file for the waveform.

In this course, you learn about creating text (ASCII) files. Use text files when you want to access the file from another application, if disk space and file I/O speed are not crucial, if you do not need to perform random access read or writes, and if numeric precision is not important.

You used an LVM file in Lesson 2. To learn more about binary and TDM files, refer to the LabVIEW Help or the LabVIEW Basics II course.

LabVIEW Data Directory

You can use the default LabVIEW Data directory to store the data files LabVIEW generates, such as `.lvm` or `.txt` files. LabVIEW installs the LabVIEW Data directory in the default file directory for your operating



system to help you organize and locate the data files LabVIEW generates. By default, the Write LabVIEW Measurement File Express VI stores the .lvnm files it generates in this directory, and the Read LabVIEW Measurement File Express VI reads from this directory. The Default Data Directory constant, shown at left, and the Default Data Directory property also return the LabVIEW Data directory by default.

Select **Tools»Options** and select **Paths** from the top pull-down menu to specify a different default data directory. The default data directory differs from the default directory, which is the directory you specify for new VIs, custom controls, VI templates, or other LabVIEW documents you create.

File I/O VIs

File I/O VIs and some File I/O functions, such as the Read from Text File and Write to Text File functions, can perform all three steps for common file I/O operations. The VIs and functions designed for multiple operations might not be as efficient as the functions configured or designed for individual operations. If you are writing to a file in a loop, use low level file I/O VIs. If you are writing to a file in a single operation, use the high-level file I/O VIs if you prefer.

Disk Streaming with Low-Level Functions

You also can use File I/O functions for disk streaming operations, which save memory resources by reducing the number of times the function interacts with the operating system to open and close the file. Disk streaming is a technique for keeping files open while you perform multiple write operations, for example, within a loop. Wiring a path control or a constant to the Write to Text File function, the Write to Binary File function, or the Write to Spreadsheet File VI adds the overhead of opening and closing the file each time the function or VI executes. VIs can be more efficient if you avoid opening and closing the same files frequently.

To avoid opening and closing the same file, you need to pass a refnum to the file into the loop. When you open a file, device, or network connection, LabVIEW creates a refnum associated with that file, device, or network connection. All operations you perform on open files, devices, or network connections use the refnums to identify each object.

The examples in Figure 10-2 and Figure 10-1 show the advantages of using disk streaming. In the first example, the VI must open and close the file during each iteration of the loop. The second example uses disk streaming to reduce the number of times the VI must interact with the operating system to open and close the file. By opening the file once before the loop begins and closing it after the loop completes, you save two file operations on each iteration of the loop.

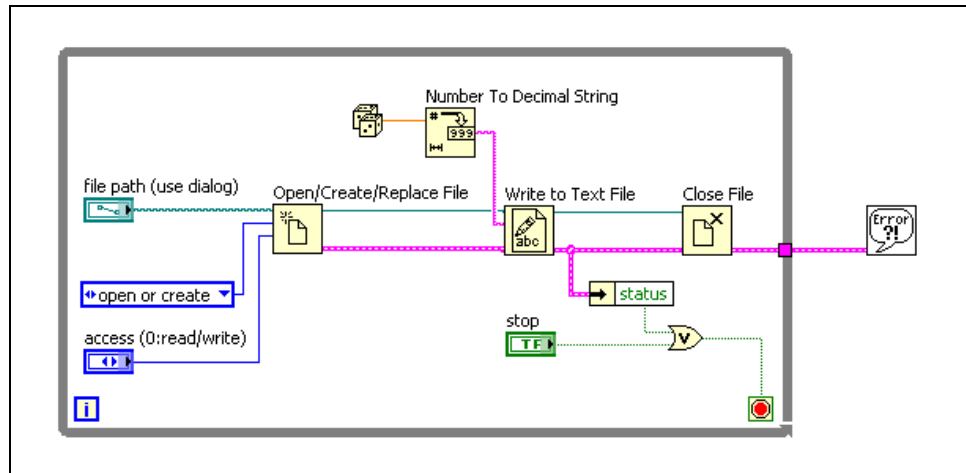


Figure 10-1. Non-Disk Streaming Example

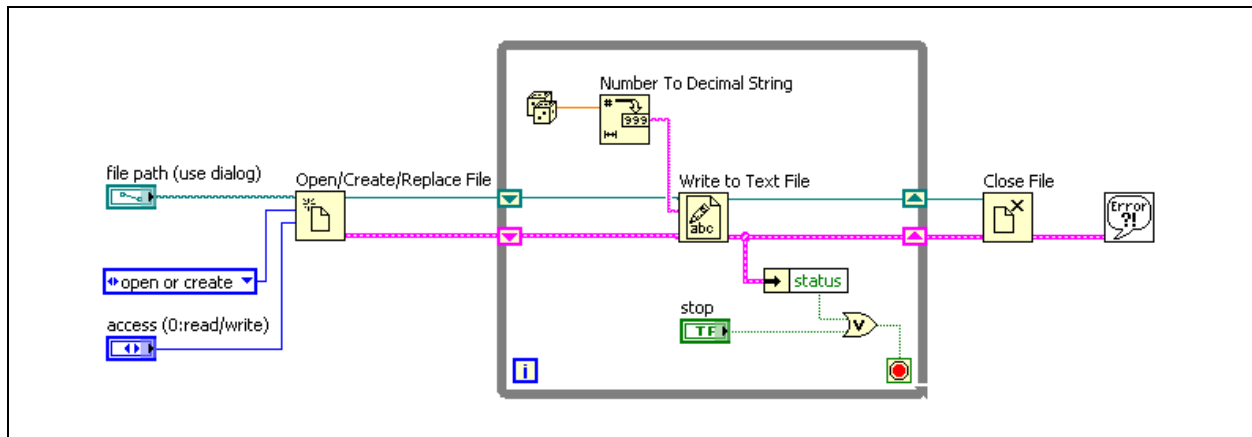


Figure 10-2. Disk Streaming Example

High Level File I/O

High-level File I/O VIs include the following:

- **Write to Spreadsheet File**—Converts a 2D or 1D array of single-precision numbers to a text string and writes the string to a new ASCII file or appends the string to an existing file. You also can transpose the data. The VI opens or creates the file before writing to it and closes it afterwards. You can use this VI to create a text file readable by most spreadsheet applications.
- **Read From Spreadsheet File**—Reads a specified number of lines or rows from a numeric text file beginning at a specified character offset and converts the data to a 2D single-precision array of numbers. The VI opens the file before reading from it and closes it afterwards. You can use this VI to read a spreadsheet file saved in text format.

- **Write to Measurement File**—An Express VI that writes data to a text-based measurement file (.lvm) or a binary measurement file (.tdm) format. You can specify the save method, file format (.lvm or .tdm), header type, and delimiter.
- **Read from Measurement File**—An Express VI that writes data to a text-based measurement file (.lvm) or a binary measurement file (.tdm) format. You can specify the file name, file format and segment size.

Avoid placing the high-level VIs in loops, because the VIs perform open and close operations each time they run.

Exercise 10-2 Optional: Read VCard VI

Goal

Read an ASCII file into LabVIEW.

Scenario

The business card contacts for your company are stored in the Windows Address Book. You must extract specific data from an individual business card into a LabVIEW text display.

Design

Inputs and Outputs

In this VI, the output appears in a dialog box, and the inputs are from a file. Therefore, there are no inputs or outputs necessary on the front panel of this VI.

Flowchart

To understand how to design this program, you must first view the text file created by the Address Book.

1. Open WordPad from **Start»Programs»Accessories»WordPad**.
2. Select **File»Open**.
3. Navigate to the `C:\Exercises\LabVIEW Basics I\Read VCard` directory.
4. Change the file type to **All Documents**.
5. Select one of the business card files in this directory.

This is an example of the text file created.

Start of VCard text file

```
BEGIN:VCARD
VERSION:2.1
N:McGillicutney;Heather;Louise;Ms.
FN:Heather Louise McGillicutney
NICKNAME:Lou
ORG:National Instruments;Internal Affairs
TITLE:President
NOTE:I am an imaginary person.
```

```
TEL;WORK;VOICE:512-555-1212¶
TEL;HOME;VOICE:512-555-1212¶
TEL;CELL;VOICE:512-555-1212¶
TEL;PAGER;VOICE:512-555-1212¶
TEL;WORK;FAX:512-555-1212¶
TEL;HOME;FAX:512-555-1212¶
ADR;WORK:;Corner;11500 N. Mopac Expressway;Austin;Texas;78759;USA¶
LABEL;WORK;ENCODING=QUOTED-PRINTABLE:Corner=0D=0A11500 N. Mopac
Expressway=0D=0AAustin, Texas 78759=0D=0AUSA¶
ADR;HOME:;;111 Easy Street;Austin;Texas;78759;USA¶
LABEL;HOME;ENCODING=QUOTED-PRINTABLE:111 Easy Street=0D=0AAustin, Texas
78759=0D=0AUSA¶
EMAIL;PREF;INTERNET:heather@ni.com¶
REV:20050818T150422Z¶
END:VCARD
```

End of VCard text file

Notice that the file contains beginning and end tags. You can use the end tag to determine when to stop reading the file. The file also had a end of line at the end of each tag. There is also a colon between the tag and the corresponding data. There is a semicolon separating different parts of a each data element. All of this information is useful when writing a VI meant to parse data.

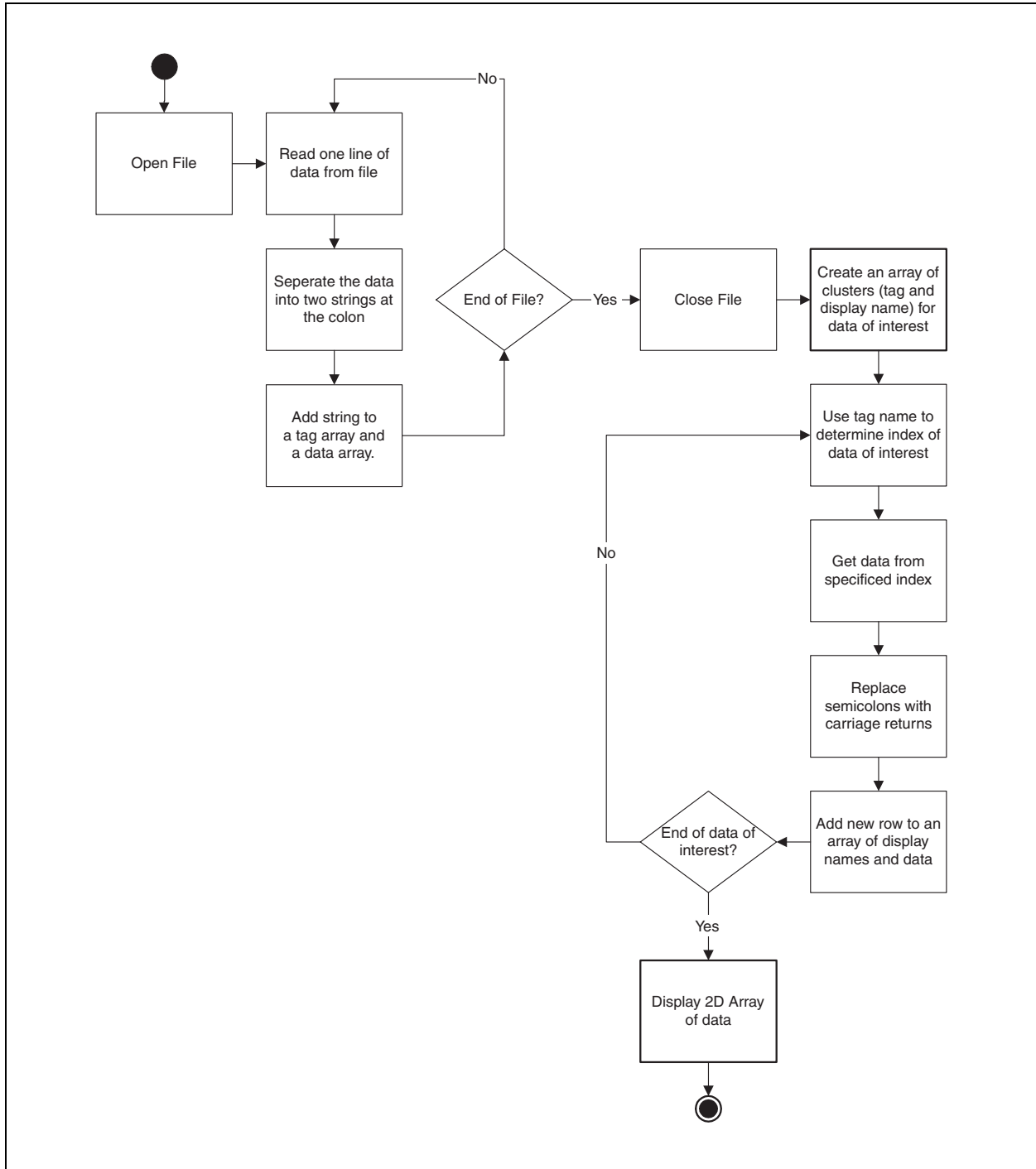


Figure 10-3. Read VCard VI Flowchart

This program consists of two loops. One loop reads the data from the business card file, line by line. The second loop chooses which pieces of data to display, replaces each tag name with a more meaningful name, and replaces the semicolons with end of line characters. The flowchart elements in Figure 10-3 with a thicker border represent VIs that have already been built for you for this exercise.

Implementation

1. Create a business card entry in the Windows Address Book.
 - Open the Address Book from **Start»Programs»Accessories»Address Book**.
 - Select **File»New Contact**.
 - Fill in some or all of the fields with your information or an imaginary contact.
 - Click **OK** when you have finished.
 - Select **File»Export»Business Card (vCard)**.
 - Navigate to the C:\Exercises\LabVIEW Basics I\Read VCard directory.
 - Click **Save**.
 - Select **File»Exit** to close the Windows Address Book.
2. Open a blank VI.
3. Save the VI as Read VCard.vi in the C:\Exercises\LabVIEW Basics I\Read VCard directory.
4. Open the block diagram.

In the following steps, you create a block diagram similar to Figure 10-4. In this block diagram, you read the vCard you just created as a 2D array of strings. The first dimension of the array contains the tags; the second dimension contains the data.

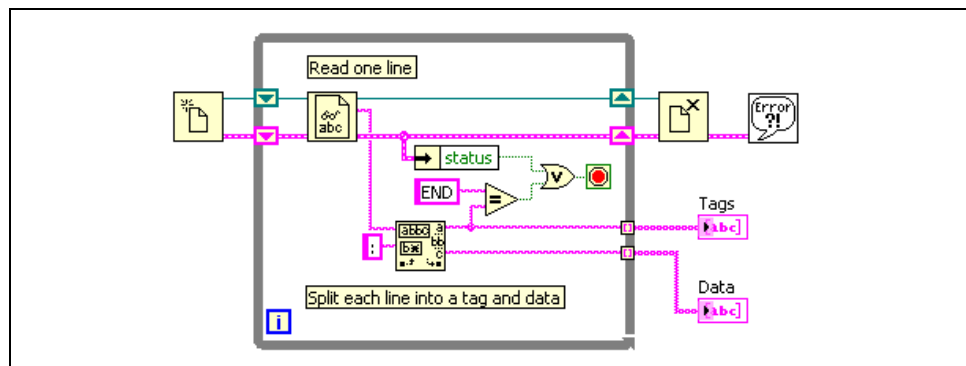


Figure 10-4. Read VCard VI Block Diagram

5. Open the text file.

- Place an Open/Create/Replace File function on the block diagram. You do not need to wire any of the inputs of this function to use the default settings.
6. Read the data from the text file.
- Place a While Loop from the **Structures** palette on the block diagram to the right of the Open/Create/Replace File function.
 - Place a Read From Text File function inside the While Loop.
 - Wire the **refnum out** terminal from the Open/Create/Replace File function to the **file (use dialog)** terminal of the Read From Text File function.
 - Wire the **error out** terminal from the Open/Create/Replace File function to the **error in** terminal of the Read From Text File function.
 - Place a Match Pattern function after the Read Text function.
 - Wire the **text** terminal of the Read Text function to the **string** terminal of the Match Pattern function.
 - Right-click the **regular expression** terminal of the Match Pattern function and select **Create»Constant** from the shortcut menu.
 - Enter a colon (:) into the string constant.
 - Wire the **before substring** terminal of the Match Pattern function to create an output tunnel from the While Loop.
 - Right-click the output tunnel and select **Enable Indexing** from the shortcut menu.
 - Wire the **after substring** terminal of the Match Pattern function to create an output tunnel from the While Loop.
 - Right-click the output tunnel and select **Enable Indexing** from the shortcut menu.
7. Stop the While Loop if an error occurs or when the end of the file has been reached.
- Wire the **error out** terminal of the Read From Text File function to create an output tunnel on the While Loop.

- Right-click the tunnel and select **Replace with Shift Register** from the shortcut menu. Your cursor should change into a shift register, indicating that you should choose the input side of the shift register.
-
- Click the error input tunnel on the right side of the While Loop to change the input tunnel to a shift register.
- Place an Unbundle By Name function inside the While Loop.
- Wire the error out from the Read from Text File function to the Unbundle By Name function.
- Place an Or function in the While Loop.
- Wire the **status** element of the error cluster to the **x** input of the Or function.
- Place an Equal function in the While Loop.
- Wire the **before substring** terminal of the Match Pattern function to the **y** terminal of the Equal function.
- Right-click the **x** terminal of the Equal function.
- Select **Create»Constant**.
- Enter END into the String Constant.
- Wire the output of the Equal function to the **y** input of the Or function.
- Wire the output of the Or function to the conditional terminal of the While Loop.



8. Close the text file.
 - Wire the **refnum out** terminal of the Read From Text File function to create an output tunnel on the While Loop.
 - Right-click the tunnel and select **Replace with Shift Register**. Your cursor should change into a shift register, indicating that you should choose the input side of the shift register.
 - Click the left refnum input tunnel of the While Loop to replace the tunnel with a shift register.
 - Place a Close File function to the right of the While Loop.

- Wire the refnum output tunnel to the **refnum input** terminal of the Close File function.
 - Wire the error output tunnel to the **error in** terminal of the Close File function.
9. Display the arrays generated on the output of the While Loop.
- Right-click the before the substring indexed output tunnel and select **Create»Indicator** from the shortcut menu.
 - Name the array indicators `Tags`.
 - Right-click the bottom array output tunnel and Select **Create»Indicator** from the shortcut menu.
 - Name the array indicators `Data`.
10. Check for errors.
- Place a Simple Error Handler to the right of the Close File function.
 - Wire the **error out** terminal from the Close Function to the **error in** terminal of the Simple Error Handler.
11. Save the VI.
12. Open the front panel of the VI.
13. Expand the indicators to show multiple elements of the arrays.
14. Run the VI.

Figure 10-5 shows an example of the front panel after running this VI. Notice that it is very similar to opening the text file. The names used for each category are not very clear. In the rest of this exercise, you modify the VI so that it parses the data for you, making it more legible to a user.



Figure 10-5. Read VCard VI Front Panel without Data Parsing

In the following steps, you add to the block diagram to parse the data in the arrays. To simplify this process, two VIs have already been built for you. One of these VIs creates an array where each array element is a cluster containing a tag and a replacement name for the tag. The second VI opens a dialog box that displays the final data in a table.

15. Switch to the block diagram.
16. Delete the tag and data array indicators.
17. Delete the wire connecting the Close File function to the Simple Error Handler.

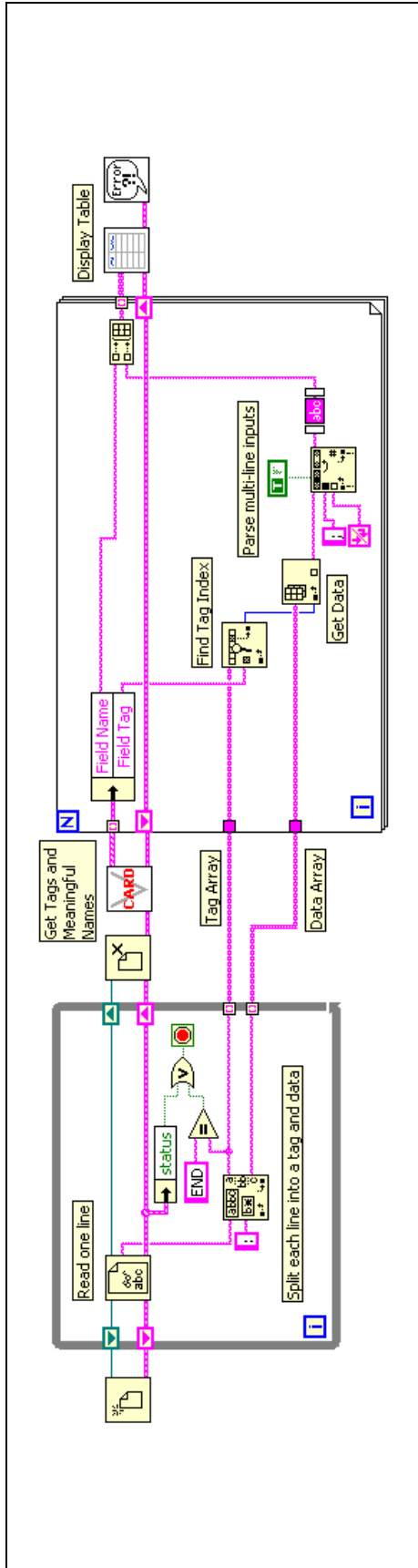


Figure 10-6. Read VCard VI Block Diagram

18. Move the Simple Error Handler out of the way. You use this VI later in this exercise.

19. Access the array of tags and replacement names.

- Place the `Vcard Tags.vi` to the right of the Close File function. This VI is located in the `C:\Exercises\LabVIEW Basics I\Read VCard` directory.



Tip Use the **Select a VI** category of the Functions palette to locate a VI that is not part of the **Functions** palette. After you have placed the VI on the block diagram, you can double-click the VI to open it and examine its block diagram.

- Wire the **error out** terminal from the Close File function to the **error in** terminal of the Vcard Tags VI.

20. Create a 2D array containing the replacement names and the corresponding data.

- Place a For Loop to the right of the Vcard Tags VI.
- Wire the **Array** terminal from the Vcard Tags VI to create an input tunnel on the For Loop. Notice that indexing has automatically been enabled.
- Place an Unbundle by Name function inside the For Loop.
- Wire the indexed input tunnel to the Unbundle by Name function.
- Expand the Unbundle by Name function to show two elements.
- Place a Search 1D Array function to the right of the Unbundle by Name function.
- Wire the **Field Tag** element of the Unbundle by Name function to the **element** terminal of the Search 1D Array function.
- Wire the tag array to the **1D array** terminal of the Search 1D Array function. The tag array is the top indexed output tunnel of the While Loop. Disable indexing on the For Loop tag away tunnel.
- Place an Index Array function to the right of the Search 1D Array function.
- Wire the data array to the **array** terminal of the Index Array function. The data array is the bottom indexed output tunnel of the While Loop. Disable indexing on the For Loop data away tunnel.
- Wire the **index of element** terminal from the Search 1D Array function to the **index** terminal of the Index Array function.

- Place a Search and Replace String function to the right of the Index Array function.
- Wire the **element** terminal of the Index Array to the **input string** terminal of the Search and Replace String.
- Right-click the **search string** terminal of the Search and Replace String function and select **Create»Constant**.
- Enter a semi-colon (;) in the string constant.
- Place an End of Line constant below the string constant.
- Wire the End of Line Constant to the **replace string** terminal of the Search and Replace String function.
- Right-click the **replace all?(F)** terminal of the Search and Replace String function and select **Create»Constant** from the shortcut menu.
- Use the Operating tool to change the False Boolean to a True Boolean.
- Place a Trim Whitespace VI to the right of the Search and Replace String function.
- Wire the **result string** terminal for the Search and Replace String function to the input of the Trim Whitespace VI.
- Place a Build String Array function to the right of the Trim Whitespace VI.
- Resize the Build Array to have two nodes.
- Wire the **Field Name** element of the Unbundle by Name function to the top node of the Build Array function.
- Wire the **trimmed string** output of the Trim Whitespace VI to the bottom node of the Build Array function. You now have a two dimensional array with meaningful descriptions in the first dimensional and corresponding data in the second dimension.



21. Display the generated array.

- Wire the output of the Build Array function to create an output tunnel on the For Loop.
- Confirm that the output tunnel is auto-indexed.

- Place the Table Dialog VI, located in the `C:\Exercises\LabVIEW Basics I\Read VCard` directory, to the right of the For Loop.
- Wire the indexed output tunnel to the **Contact Information** terminal of the Table Dialog VI.
- Wire the error cluster from the Vcard Tags VI to the Table Dialog VI.
- Replace the error cluster tunnels with shift registers.
- Move the Simple Error Handler VI to the right of the Table Dialog VI.
- Wire the **error out** terminal of the Table Dialog VI to the **error in** terminal of the Simple Error Handler VI.

22. Switch to the front panel.

23. Save the VI.

Testing

1. Run the VI.
2. When prompted, navigate to the business card file you created earlier in this exercise.
3. Close the Table Dialog window to stop the VI.
4. Close the VI when you have finished.

End of Exercise 10-2

Self-Review: Quiz

1. You are writing a testing application to test the audio characteristics of microphones as they are manufactured. For each microphone you test, you must acquire 1 second of sound data and compare the data to an ideal signal. You then need to log the acquired data and a pass/fail status to a file. What type of analysis should you use?
 - a. Inline programmatic analysis
 - b. Offline programmatic analysis
 - c. Interactive analysis
2. For the situation described in the previous question, your continuously running test program logs to a single file the results of all tests that occur in one hour. If you are concerned about the execution speed of your program, should you use low-level or high-level File I/O VIs?
 - a. Low-level file I/O VIs
 - b. High-level file I/O VIs
3. If you want to view data in a text editor such as Notepad, what file format should you use to save the data?
 - a. ASCII
 - b. TDM

Self-Review: Quiz Answers

1. You are writing a testing application to test the audio characteristics of microphones as they are manufactured. For each microphone you test, you must acquire 1 second of sound data and compare the data to an ideal signal. You then need to log the acquired data and a pass/fail status to a file. What type of analysis should you use?
 - a. **Inline programmatic analysis**
 - b. Offline programmatic analysis
 - c. Interactive analysis
2. For the situation described in the previous question, your continuously running test program logs to a single file the results of all tests that occur in one hour. If you are concerned about the execution speed of your program, should you use low-level or high-level File I/O VIs?
 - a. **Low-level file I/O VIs**
 - b. High-level file I/O VIs
3. If you want to view data in a text editor such as Notepad, what file format should you use to save the data?
 - a. **ASCII**
 - b. TDM

Notes

Common Design Techniques and Patterns

The first step in developing a LabVIEW project is to explore the architectures that exist within LabVIEW. Architectures are essential for creating a successful software design. The most common architectures are usually grouped into design patterns.

As a design pattern gains acceptance, it becomes easier to recognize when a design pattern has been used. This recognition helps you and other developers read and make changes to VIs that are based on design patterns.

There are many design patterns for LabVIEW. Most applications use at least one design pattern. In this course, you explore the State Machine design pattern. Learn more about design patterns in *LabVIEW Basics II*.

Topics

- A. Sequential Programming
- B. State Programming
- C. State Machines
- D. Parallelism

A. Sequential Programming

In Lesson 1, *Problem Solving*, you designed a flow chart for a Temperature Weather Station. The Temperature Weather Station is a set of steps:

1. Read the temperature
2. Test the temperature for limits and display warnings
3. Graph and log the temperature

After you reach the end of the sequence of events, you check to see if the stop button has been pressed, and, if not, the sequence repeats.

Many of the VIs you write in LabVIEW accomplish sequential tasks. How you program these sequential task can be very different. Consider the block diagram in Figure. In this block diagram, a voltage signal is acquired, then a dialog is presented to the user asking them to turn on the power, then the voltage signal is acquired again, and the user is asked to turn off the power. However, in this example, there is nothing in the block diagram to force the execution order of these events. Any one of these events could happen first.

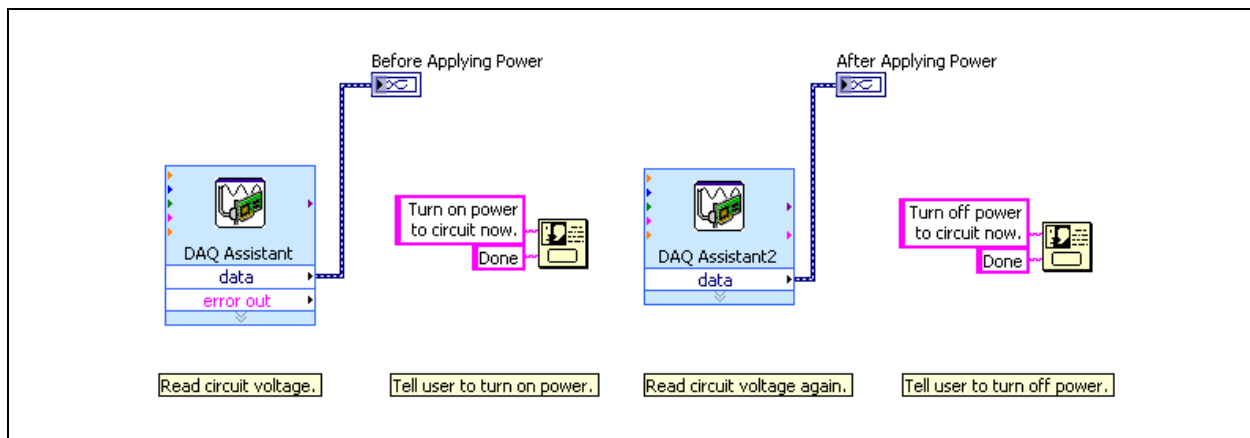


Figure 11-1. Tasks Not Sequenced

In LabVIEW, you can complete sequential tasks by placing each task in a separate subVI, and wiring the subVIs in order using the error cluster. However, in this example, only two of the tasks have a error cluster. Using the error clusters, you can force the execution order of the two DAQ Assistants, but not the dialog boxes, as shown in Figure 11-2.

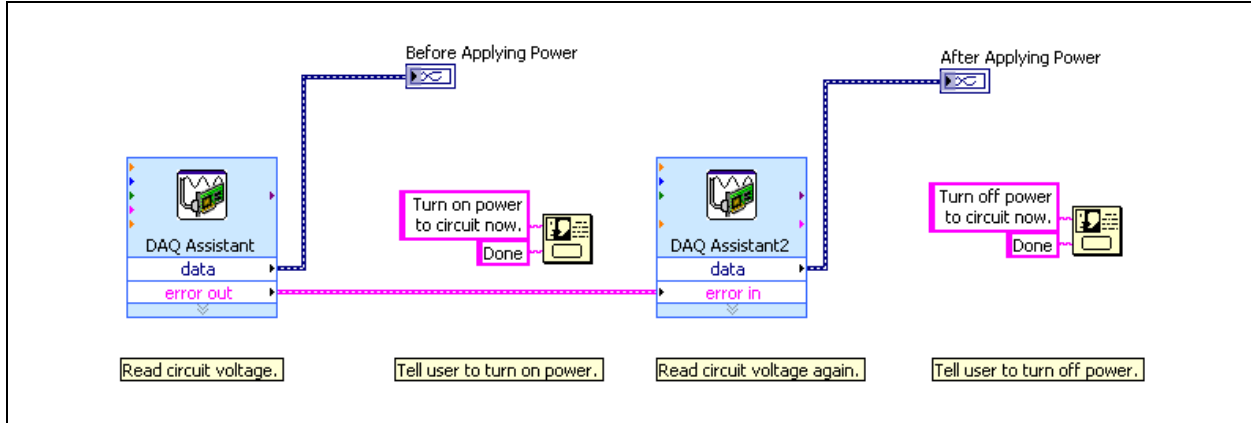


Figure 11-2. Partially Sequenced Tasks

Use a Sequence structure to force the order of operations of a block diagram object. A Sequence structure is simply a structure with frames, where each frame executes in order; the second frame cannot begin execution until everything in the first frame has completed execution. Figure 11-3 shows an example of this VI using a Sequence structure to force execution order.

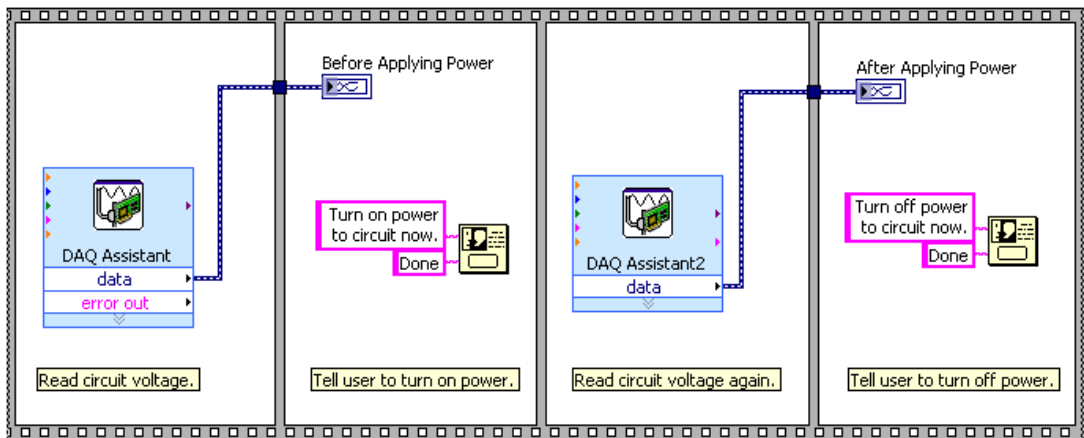


Figure 11-3. Tasks Sequenced with a Sequence Structure

To take advantage of the inherent parallelism in LabVIEW, avoid overusing Sequence structures. Sequence structures guarantee the order of execution and prohibit parallel operations. Another negative to using Sequence structures is that you cannot stop the execution part way through the sequence. A good way to use Sequence structures for this example is shown in Figure 11-4.

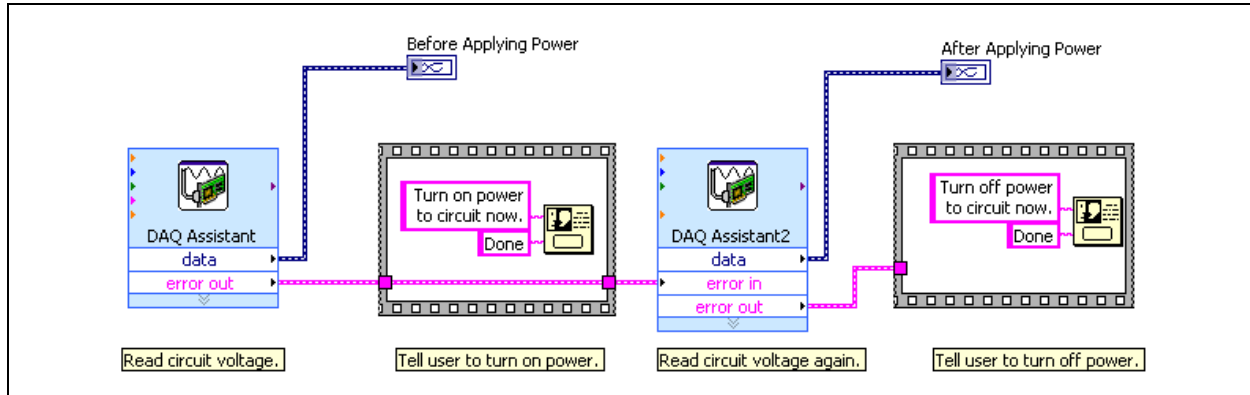


Figure 11-4. Tasks that have been sequenced using Sequence Structures and an error cluster

The best way to write this VI is to enclose the dialog boxes in a case structure, wiring the error cluster to the case selector.

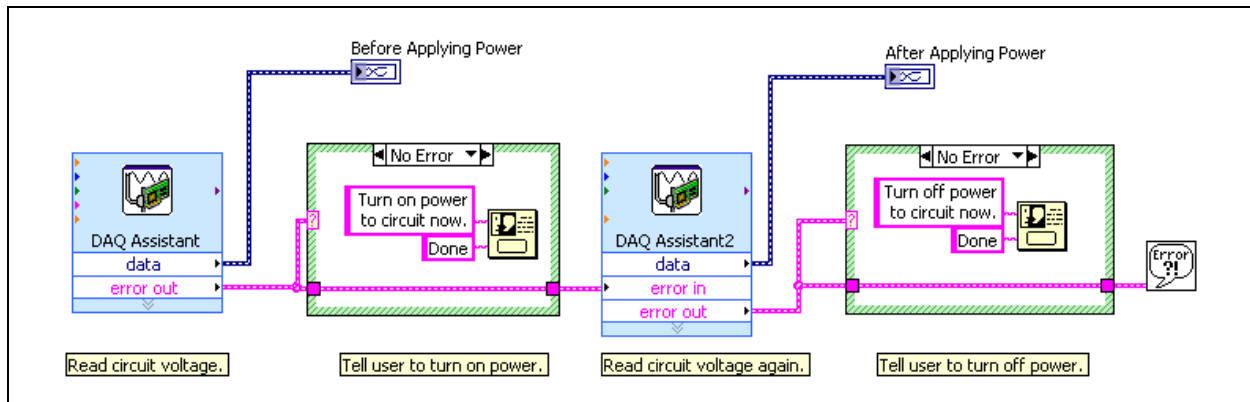


Figure 11-5. Tasks that have been sequenced using an error cluster and Case structures

B. State Programming

Although a Sequence structure and sequentially wired subVIs both accomplish the task, sometimes more is necessary:

- What if you must change the order of the sequence?
- What if you must repeat one item in the sequence more often than the other items?
- What if some items in the sequence execute only when certain conditions are met?
- What if you must stop the program immediately, rather than waiting until the end of the sequence?

Although your program may not have any of the above requirements, there is always the possibility that the program must be modified in the future. For this reason, a state programming architecture is a good choice, even if a sequential programming structure would be sufficient.

C. State Machines

The state machine design pattern is a common and very useful design pattern for LabVIEW. You can use the state machine design pattern to implement any algorithm that can be explicitly described by a state diagram or flow chart. A state machine usually implements a moderately complex decision-making algorithm, such as a diagnostic routine or a process monitor.

A *state machine*, which is more precisely defined as a finite state machine, consists of a set of states and a transition function that maps to the next state. Finite state machines have many variations. The two most common finite state machines are the Mealy machine and the Moore machine. A Mealy machine performs an action for each transition. A Moore machine performs a specific action for each state in the state transition diagram. The state machine design pattern template in LabVIEW implements any algorithm described by a Moore machine.

Use state machines in applications where distinguishable states exist. Each state can lead to one or multiple states or end the process flow. A state machine relies on user input or in-state calculation to determine which state to go to next. Many applications require an initialization state, followed by a default state, where many different actions can be performed. The actions performed can depend on previous and current inputs and states. A shutdown state commonly performs clean up actions.

State machines are commonly used to create user interfaces. In a user interface, different user actions send the user interface into different processing segments. Each processing segment acts as a state in the state machine. Each segments can lead to another segment for further processing or wait for another user action. In this example, the state machine constantly monitors the user for the next action to take.

Process testing is another common application of the state machine design pattern. For a process test, a state represents each segment of the process. Depending on the result of each state's test, a different state might be called. This can happen continually, resulting in an in-depth analysis of the process you are testing.

The advantage of using a state machine is that once you have created a state transition diagram, you can build LabVIEW VIs easily.

State Machine Infrastructure

Translating the state transition diagram into a LabVIEW block diagram requires the following infrastructure components:

- **While Loop**—Continually executes the various states
- **Case Structure**—Contains a case for each state and the code to execute for each state
- **Shift Register**—Contains state transition information
- **State Functionality Code**—Implements the function of the state
- **Transition Code**—Determines the next state in the sequence

Figure 11-5 shows the basic structure of a state machine implemented in LabVIEW for a temperature data acquisition system.

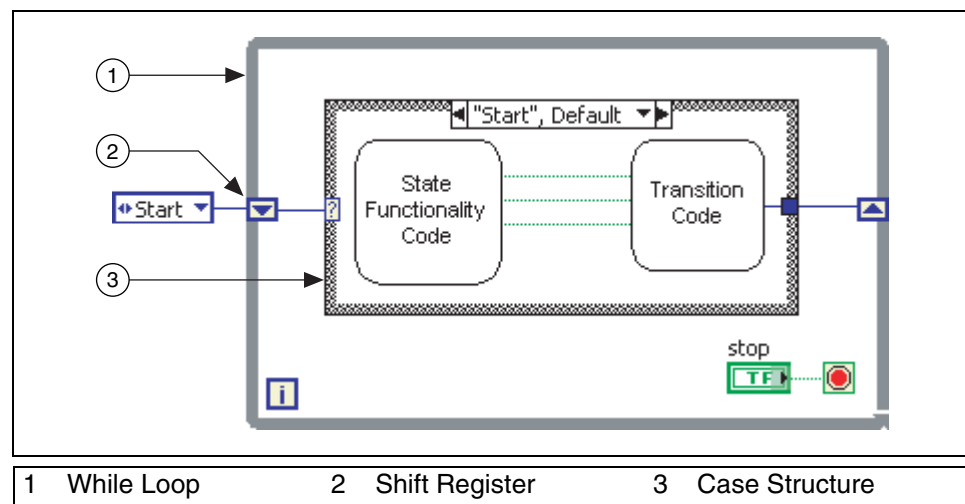


Figure 11-6. Basic Infrastructure of a LabVIEW State Machine

The flow of the state transition diagram is implemented by the While Loop. The individual states are represented by cases in the Case structure. A shift register on the While Loop keeps track of the current state and communicates the current state to the Case structure input.

Controlling the State Machine

There are many ways to control what case a Case structure executes in a state machine. Choose the method that best suits the function and complexity of your state machine. Of the methods to implement transitions in state machines, the most common and easy to use is the single Case structure transition code. This method provides for the most scalable, readable, and maintainable state machine architecture. The other methods can be useful in specific situations, and it is important for you to be familiar with them.

Enumerated Type Control

The best method for controlling the initialization and transition of state machines is the enumerated type control. Enumerated type controls are widely used as case selectors in state machines. However, if the user attempts to add or delete a state from the enumerated type control, the remaining wires that are connected to the copies of this enumerated type control breaks. This is one of the most common obstacles when implementing state machines with enumerated type controls. One solution to this problem is to create a new control with the enumerated type control, then right-click the enumerated type control and select **typedef** from the shortcut menu. Creating a type defined enumerated type control causes all the enumerated type control copies to automatically update if you add or remove a state.

Default Transition

Use enumerated type controls to control the initialization and transition of state machines. There are several common design patterns used to develop the transition code. Figure 11-7 shows a design pattern that uses a default transition implemented for a temperature data acquisition system. For the default transition, no code is needed to determine the next state.

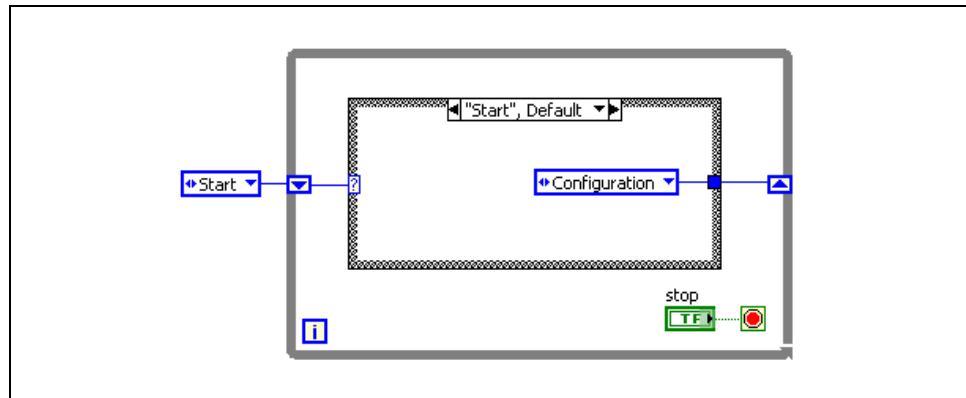


Figure 11-7. Single Default Transition

Transition between Two States

The following method involves making a decision on a transition between two states. There are several patterns commonly used to accomplish this. Figure 11-8 shows the Select function transition implemented for a temperature data acquisition system.

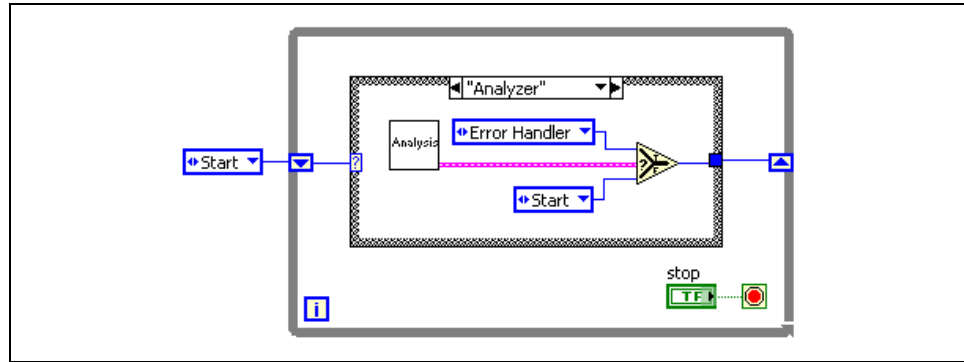


Figure 11-8. Transition between Two States

This method works well if you know that the individual state always transitions between two states. However, this method limits the scalability of the application. If you need to modify the state to transition among more than two states, this solution would not work and would require a major modification of the transition code.

Transition among Two or More States

Create a more scalable architecture by using one of the following methods to transition among states.

- **Case Structure**—Use a Case structure instead of the Select function for the transition code.

Figure 11-9 shows the Case structure transition implemented for a temperature data acquisition system.

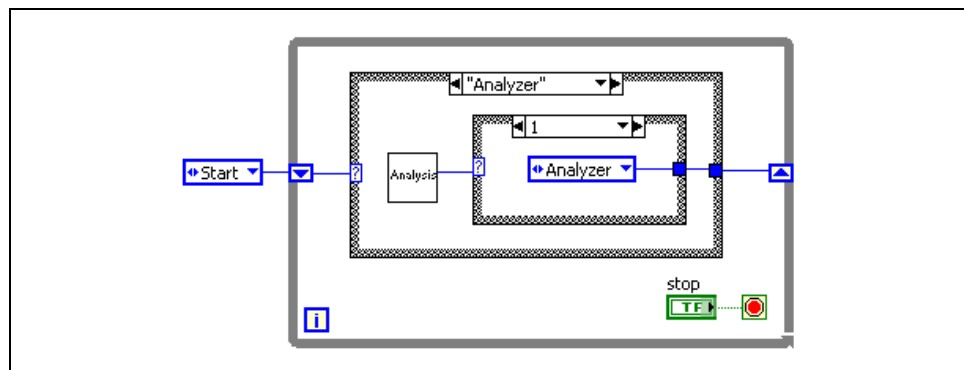


Figure 11-9. Case Structure Transition Code

One advantage to using a Case structure is that the code is self-documenting. Because each case in the Case structure corresponds to an item in the enumerated type control, it is easy to read and understand the code. A Case structure also is scalable. As the application grows, you can add more transitions to a particular state by adding more cases to the Case structure for that state. A disadvantage to using a Case structure is that not all the code is visible at once. Because of the nature of the Case

structure, it is not possible to see at a glance the complete functionality of the transition code.

- **Transition Array**—If you need more of the code to be visible than a Case structure allows, you can create a transition array for all the transitions that can take place in the transition code for a state and index the Boolean array for the next state.

Figure 11-10 shows the transition array implemented for a temperature data acquisition system.

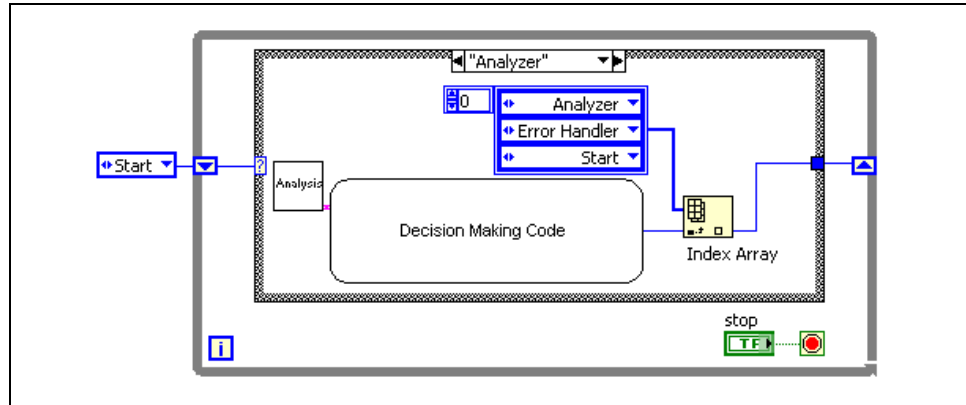


Figure 11-10. Transition Array Transition Code

In this example, the array of enumerated type controls are indexed by the first True Boolean constant in the Boolean array that corresponds to the index of the new state in the array of enumerated type controls. This design pattern makes the transition code scalable and easy to read. One disadvantage of this pattern is that you must use caution when developing the transition code. You must ensure that there is a one-to-one correspondence of the items in the Boolean array with the items in the transition array. Also, the Search 1D Array function returns a -1 if the item is not found. For the Index Array function to perform with the expected behavior, you must increment the output of the Search 1D Array function by 1. In Figure 11-10, if an item is not found, the state machine remains in the Analyzer state.

- **State Diagram Toolkit**—Another pattern that implements the transition code is also used by the NI LabVIEW State Diagram Toolkit. This pattern uses a large Case structure for every state and a smaller While Loop that iterates through the state transitions until the proper state transition is met. Figure 11-11 shows the LabVIEW State Diagram Toolkit implementation for a temperature data acquisition system.

The LabVIEW State Diagram Toolkit adds the State Diagram Editor function to LabVIEW to visually draw the logic that defines an application. As this visual representation of the logic is created, the State Diagram Editor generates the LabVIEW code that acts as the foundation of your application.



Note The LabVIEW State Diagram Toolkit is not included in the LabVIEW Professional Development System, it is available for purchase separately.

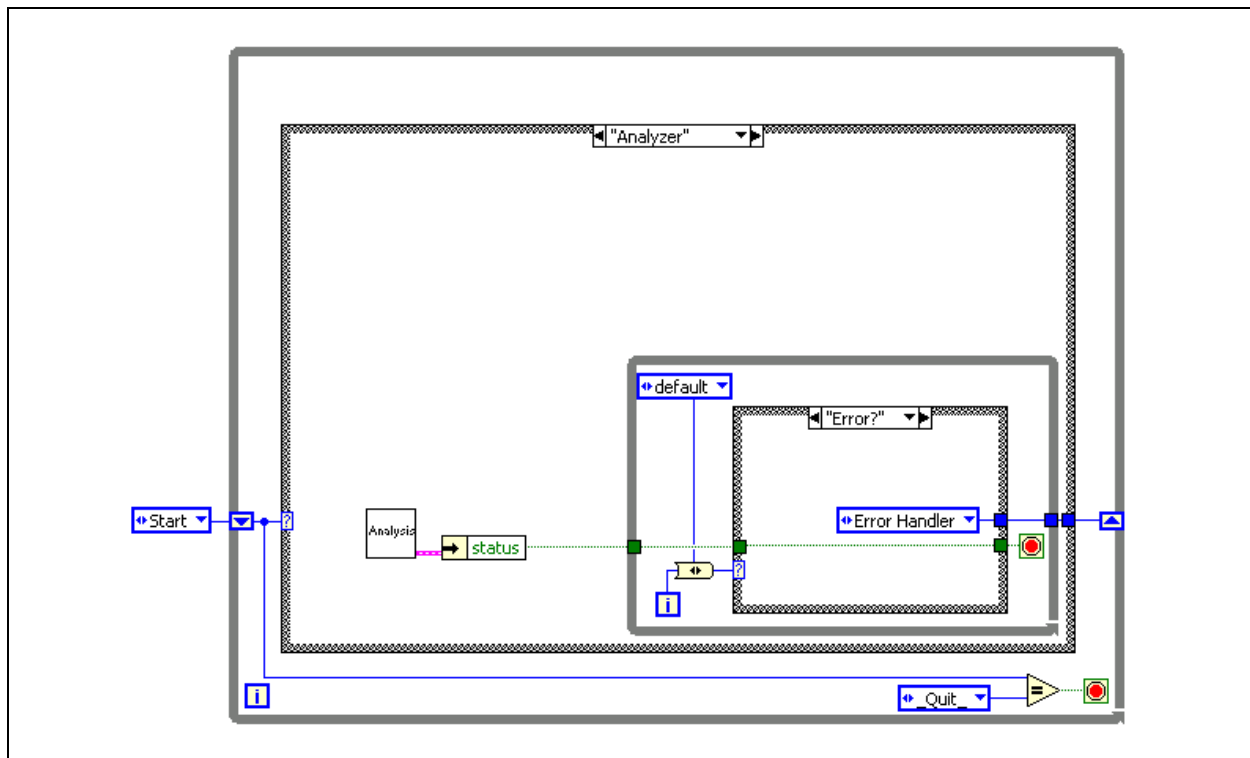


Figure 11-11. State Diagram Toolkit State Machine

The large Case structure selects the appropriate code to execute for the current active state. Notice that the shift register is initialized with a value from the enumerated type control to reflect the initial state specified in the state transition diagram. The large Case structure contains one subdiagram for every state that specifies which set of instructions to execute when each state becomes active. The code for evaluating transition conditions is contained within a While Loop inside the code diagram for that state. This While Loop contains another enumerated data type (unique to the state) that contains one entry for each transition that leaves the state. The entries are ordered by transition priority. Higher priority transitions have lower values in the enumeration. When the small While Loop executes, the values are fed one by one, in order from smallest value to largest, into a smaller transition Case structure. The (integer) values from the loop index terminal are converted into the transition data type. The transition Case structure outputs two values—the value of the transition condition for the transition being tested and the value of the new active state, assuming the condition for this transition evaluates to TRUE. As soon as a TRUE transition occurs, the While Loop stops, and the state machine moves on to the next active state.

The following conditions must exist in the State Diagram Toolkit state machine:

- There must always be a default transition.
- The default transition must always have lowest priority so it is evaluated last.
- The default transition condition must always be TRUE. You can wire a Boolean constant to the condition output for the default transition.

Exercise 11-1 Project: Temperature Weather Station

Goal

Create a state machine and practice the skills learned throughout this course.

Scenario

Complete the course project using a state machine and the state flow chart developed earlier in the course.

The project acquires a temperature every half a second, analyzes each temperature to determine if the temperature is too high or too low, then alerts the user if there is a danger of heatstroke or freeze. The program logs the data if a warning has occurred. If the user has not pressed the stop button, the entire process repeats.

Design

You have already completed the design stage for this project. The inputs and outputs and flowchart are repeated here.

User Interface Inputs and Outputs

Type	Name	Properties
Numeric Control	Upper Limit	Double Precision
Numeric Control	Lower Limit	Double Precision
Waveform Chart	Temperature History	
String Indicator	Warning	Four possible values: Heatstroke Warning, Freeze Warning, No Warning and Upper Limit < Lower Limit
Stop Button	Stop	

External Inputs and Outputs

- Inputs:** Current temperature retrieved from a sensor attached to AI0 of the data acquisition board.
 Conversion formula: $\text{Voltage} \times 100 = \text{Celsius}$
- Outputs:** tab-delimited ASCII file containing current temperature, temperature limits, and warning string for each warning that occurs.

State Transition Diagram

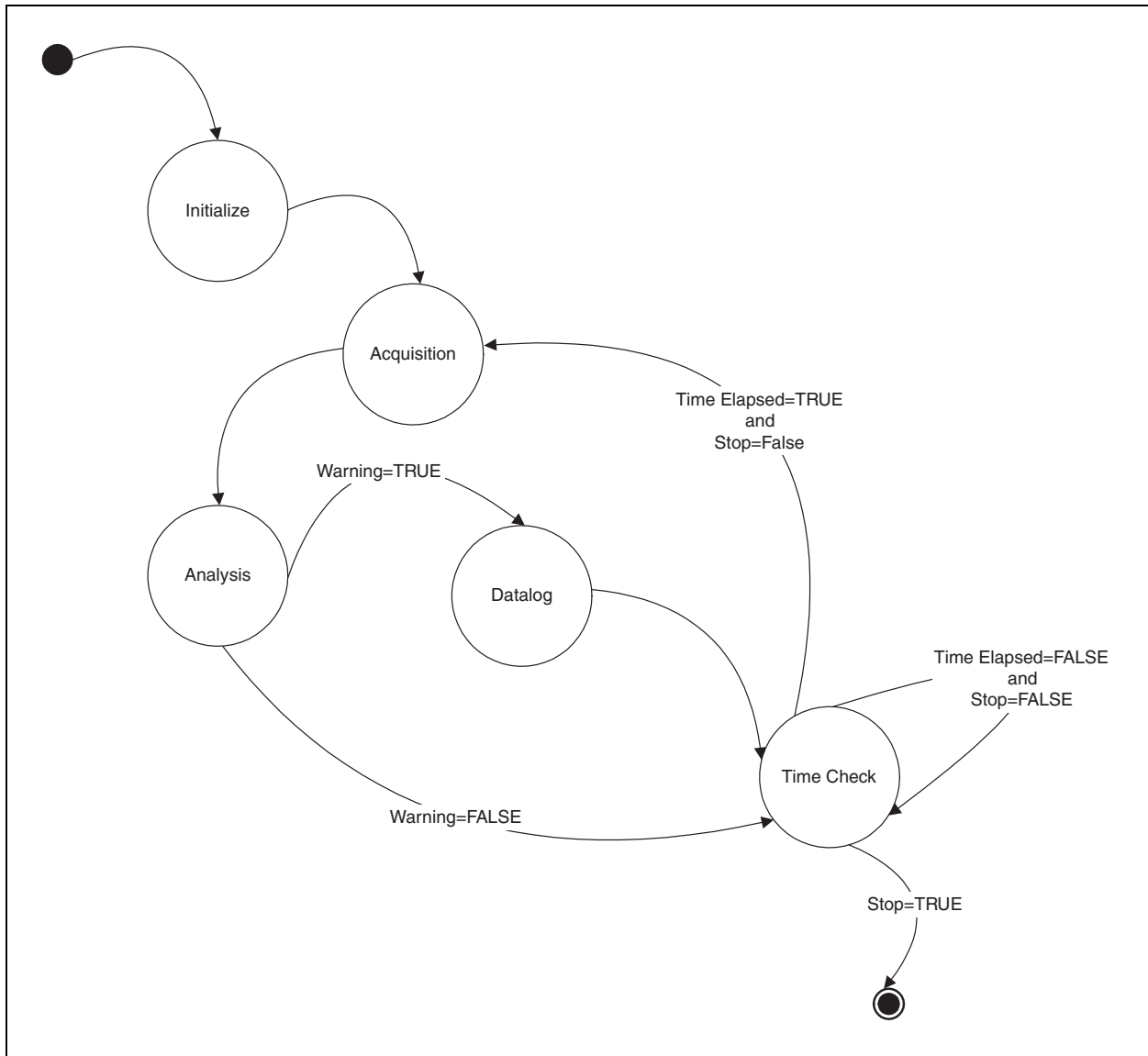


Figure 11-12. State Transition Diagram for the Course Project

Implementation

In the following steps, you set up the state machine in the Weather Station UI VI.

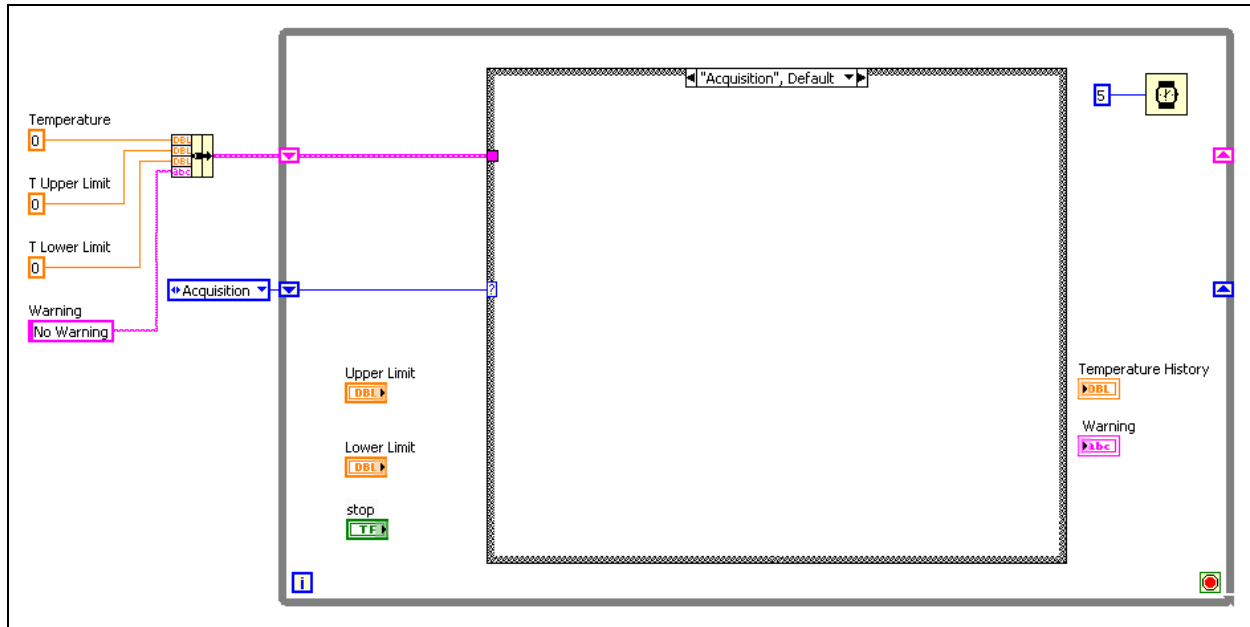


Figure 11-13. Setting up the State Machine

1. Open the weather station project.
 - Select **File»Open Project**. Navigate to the `C:\Exercises\LabVIEW Basics I\Course Project` directory.
 - Select **Weather Station.proj**.
2. Open the user interface for the weather station project.
 - In the **Project Explorer** window, double-click the **Weather Station UI.vi**.
3. Switch to the block diagram.
4. Create a state machine on the block diagram using the existing enumerate type control.
 - Place a While Loop from the **Structures** palette on the block diagram, as shown in Figure 11-13.
 - Place a Case structure inside the While Loop.
 - Place the Weather Station State enumerated type constant outside the While Loop.

- Connect the Weather Station State enumerated type constant to the case selector terminal.
 - Right-click the Case structure and select **Add Case for Every Value** from the shortcut menu.
 - Change the enumerated type constant tunnel to a shift register.
5. Place a Wait function in the loop so that the processor can handle other tasks when needed.
- Place a Wait (ms) function inside the While Loop, outside the Case structure.
 - Right-click the **milliseconds to wait** terminal and select **Create»Constant** from the shortcut menu.
 - Set the constant to a value of 5.
6. Pass the temperature cluster into the state machine so that it can be accessed from any state.
- Wire the output of the Bundle function to the While Loop to create a tunnel.
 - Change the cluster tunnel to a shift register.
 - Wire the shift register to the Case structure.
7. Confirm that your block diagram is similar to the one shown in Figure 11-13.
8. Save the VI.

In the following sections, complete each state of the state machine.

Acquisition State

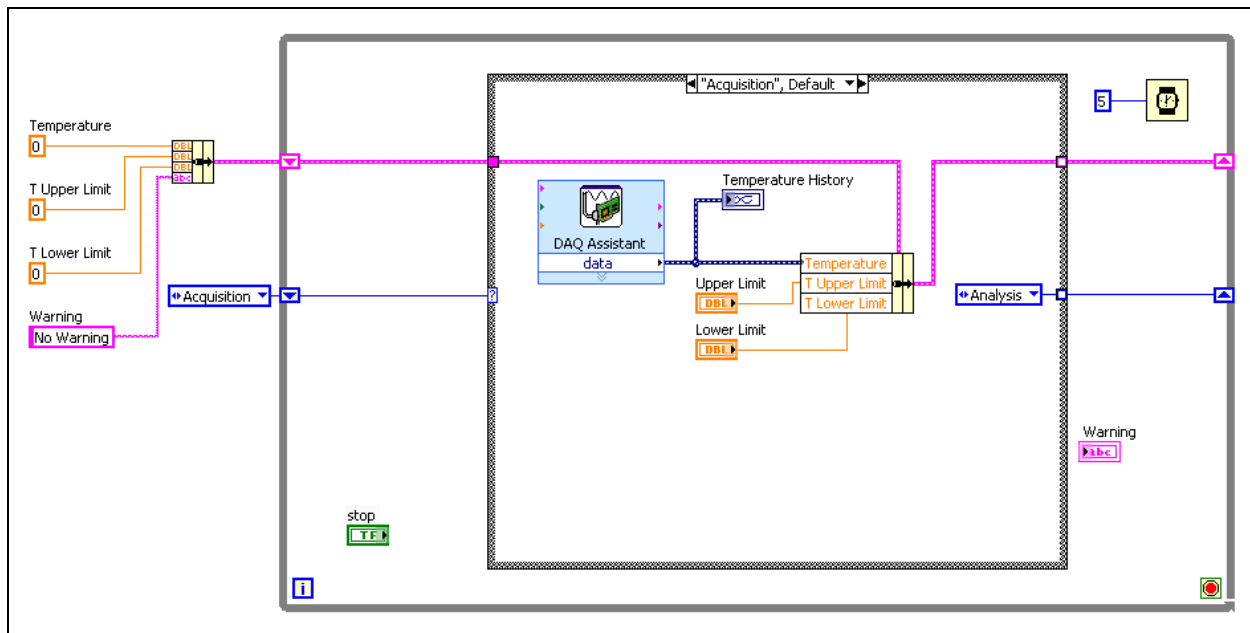


Figure 11-14. Acquisition State

1. Switch to the Acquisition state of the state machine.
2. Acquire a single temperature reading.

Hardware

- Place the DAQ Assistant in the Acquisition state.
- Select **Analog Input»Voltage** for the task type.
- Select **ai0** on your DAQ device for the channel.
- Click the **Finish** button.
- Select **Temperature** from the **Custom Scaling** pull-down menu.
- Set the signal input range to **0 to 50 degrees Celsius**.
- Set the Task Timing to **1 Sample (On Demand)**.
- Click the **OK** button.
- Skip to step 3.

No Hardware

- Select the Temperature Simulator VI in the **Project Explorer** window and drag it into the Acquisition state. In future steps, you wire the inputs and outputs of the Temperature Simulator VI instead of the DAQ Assistant.
3. Display the acquired temperature on the Temperature History Waveform Chart.
 - Move the Temperature History terminal inside the Acquisition state.
 - Wire the data output of the DAQ Assistant to the Temperature History terminal. Notice that the waveform graph data type automatically changes to match the incoming data.
 4. Update the temperature cluster with the current temperature, upper limit, and lower limit.
 - Place an Unbundle by Name function inside the Acquisition state.
 - Wire the **temperature** cluster to the Unbundle by Name function.
 - Expand the Unbundle by Name function to show three elements.
 - If necessary, change the contents to show the following elements in this order: Temperature, T Upper Limit, T Lower Limit.
 - Move the Upper Limit and Lower Limit terminals into the Acquisition state.
 - Wire the **data** output of the DAQ Assistant to the Temperature element of the Unbundle by Name function.
 - Wire the Upper Limit terminal to the T Upper Limit element of the Unbundle by Name function.
 - Wire the Lower Limit terminal to the T Lower Limit element of the Unbundle by Name function.
 5. Set the state machine to go to the Analysis state next.
 - Place a copy of the Weather Station State enumerated type constant inside the Acquisition state.
 - Select the **Analysis** element of the enumerated type constant.
 - Wire the enumerated type constant to the output state shift register.

6. Save the VI.

Analysis State

In previous exercises, you wrote a subVI that performs the analysis required for this state machine: Determine Warnings VI. This VI is already added to the project file, but needs to be placed in the main VI.

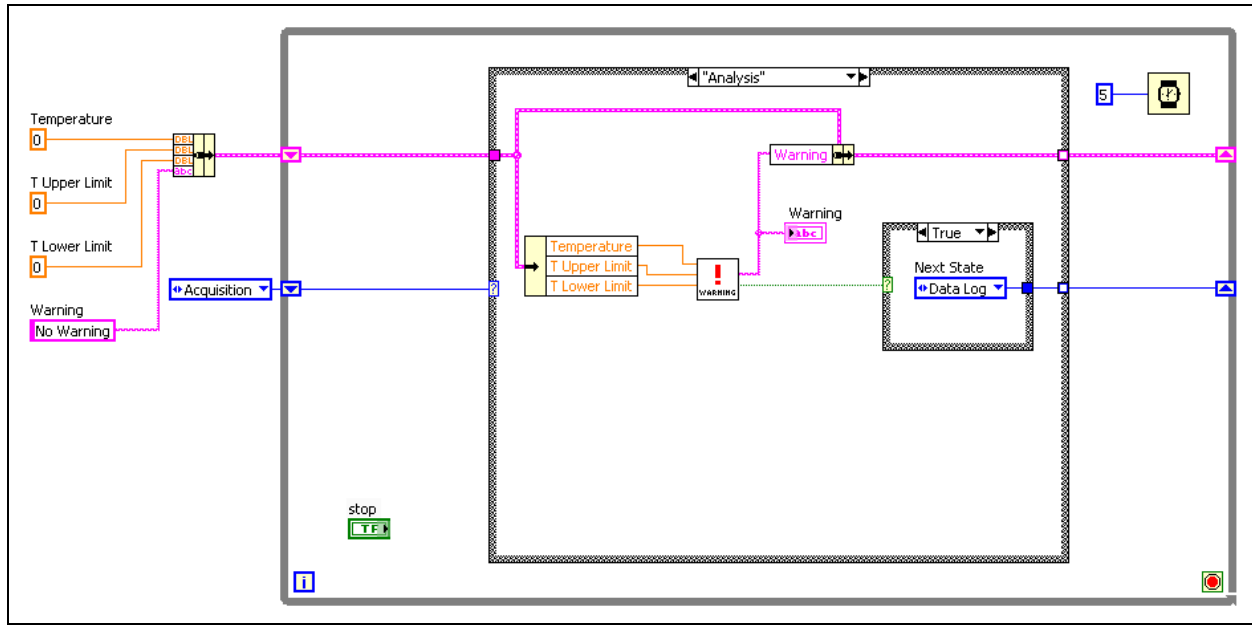


Figure 11-15. Analysis State

1. Switch to the Analysis state of the state machine.
2. Place the Determine Warnings VI in the Project Explorer and place in the Analysis state.
 - Switch to the **Project Explorer** window.
 - Select the Determine Warnings VI and drag it into the Analysis state.
3. Pass the temperature values to the Determine Warnings VI.
 - Place an Unbundle by Name function inside the Analysis state.
 - Wire the temperature cluster to the Unbundle by Name function.
 - Expand the Unbundle by Name function to show three elements.
 - If necessary, change the contents to show the following elements in this order: Temperature, T Upper Limit, T Lower Limit.

- Wire each of the Unbundle by Name elements to the appropriate terminal of the Determine Warnings VI.
4. Display the output of the Determine Warnings VI.
- Move the Warning terminal inside the Analysis state.
 - Wire the Warning Text terminal of the Determine Warnings VI to the Warning terminal.
5. Update the warning value in the temperature cluster.
- Place a Bundle by Name function inside the Analysis state.
 - Wire the **temperature** cluster to the input cluster terminal of the Bundle by Name function.
 - Choose the Warning element in the Bundle by Name function.
 - Wire the **Warning Text** terminal of the Determine Warnings VI to the **Warning** terminal.
 - Wire the output of the Bundle by Name function to the output temperature cluster shift register.
6. If a warning has occurred, log the data; otherwise check to see if it is time to acquire the next piece of data.
- Place a Case structure inside the Analysis state.
 - Wire the Warning? terminal of the Determine Warnings VI to the case selector of the new Case structure.
 - Switch to the True case of the Case structure.
 - Place a copy of the Weather Station State enumerated type constant inside the True case.
 - Select the Data Log element of the enumerated type control.
 - Wire the enumerated type control to the output state shift register.
 - Switch to the False case of the Case structure.
 - Place a copy of the Weather Station State enumerated type control inside the True case.
 - Select the Time Check element of the enumerated type control.

- Wire the enumerated type constant to the tunnel of the Case structure.

7. Save the VI.

Data Log State

In this state, the data is logged in a tab-delimited ASCII file. You must open the file before the state machine executes, and close the file after the state machine executes.

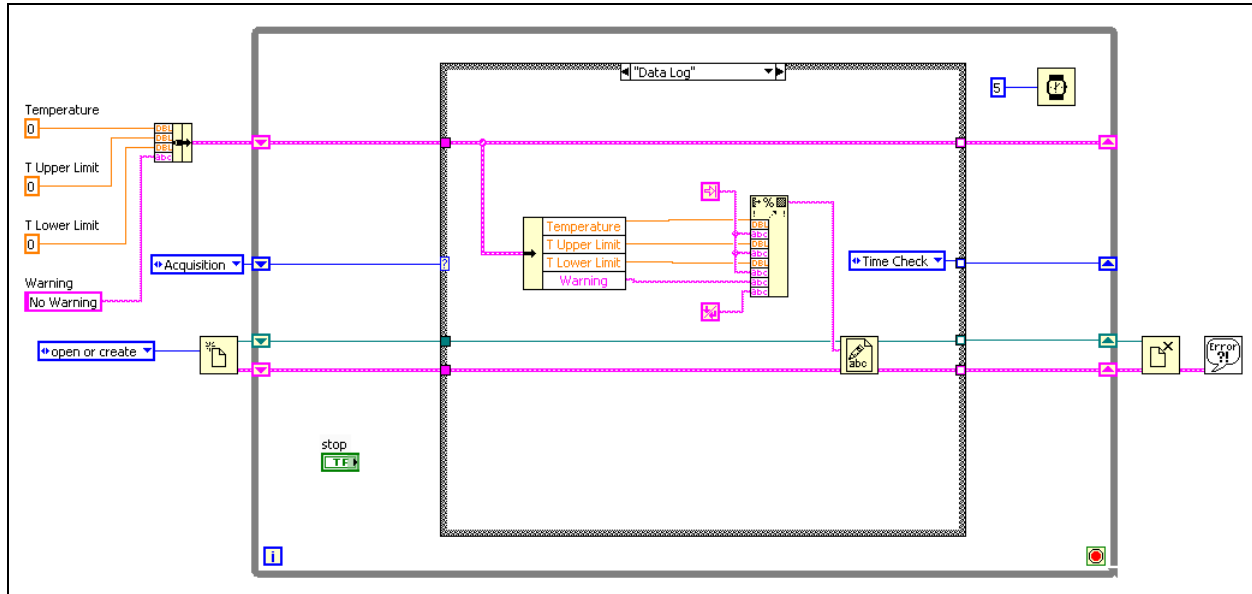


Figure 11-16. Data Log State

1. Switch to the Data Log state of the state machine.
2. Place the Open/Create/Replace File function to the left of the state machine.
3. Set the Open/Create/Replace File function to open or create.
 - Right-click the **operation** terminal of the Open/Create/Replace File function and select **Create»Constant** from the shortcut menu.
 - Select the **open or create** element of the constant.
4. Pass the error cluster and file refnum into the state machine so that it can be used by all states.
 - Wire the **error out** terminal of the Open/Create/Replace File function to the While Loop.
 - Replace the tunnel with a shift register.

- Wire the refnum out terminal of the Open/Create/Replace File function to the While Loop.
 - Replace the tunnel with a shift register.
5. Prepare the temperature data for writing by combining it all in one string.
- Place an Unbundle by Name function inside the Data Log state.
 - Wire the **temperature** cluster to the input of the Unbundle by Name function.
 - Resize the function to show four elements.
 - Select the elements so they are in the following order: Temperature, T Upper Limit, T Lower Limit, and Warning.
 - Place a Format Into String function to the right of the Unbundle by Name function.
 - Resize the function to have eight inputs.
 - Place a tab constant in the Data Log state.
 - Place an End of Line constant in the Data Log state.
 - Wire the inputs in the following order: temperature, <tab>, T Upper Limit, <tab>, T Lower Limit, <tab>, Warning, <end of line>.
6. Write the temperature string to the file.
- Place the Write To Text File function in the Data Log state.
 - Wire the output of the Format Into String function to the **text** input of the Write To Text File function.
 - Wire the error clusters to and from the Write To Text File function.
 - Wire the file refnum to and from the Write To Text File function.
7. Close the file.
- Place the Close File function to the right of the state machine.
 - Wire the **error** cluster and the **file refnum** to the Close File function.
 - Place a Simple Error Handler to the right of the Close File function.

- Wire the **error out** terminal from the Close File function to the Simple Error Handler.
- 8. Place a copy of the Weather Station State enumerated type control inside the Data Log state.
- 9. Select the Time Check element of the constant.
- 10. Wire the constant to the tunnel of the Case structure.
- 11. Save the VI.

Time Check State

In this state, determine whether half a second has passed. If it has, acquire the next piece of data.

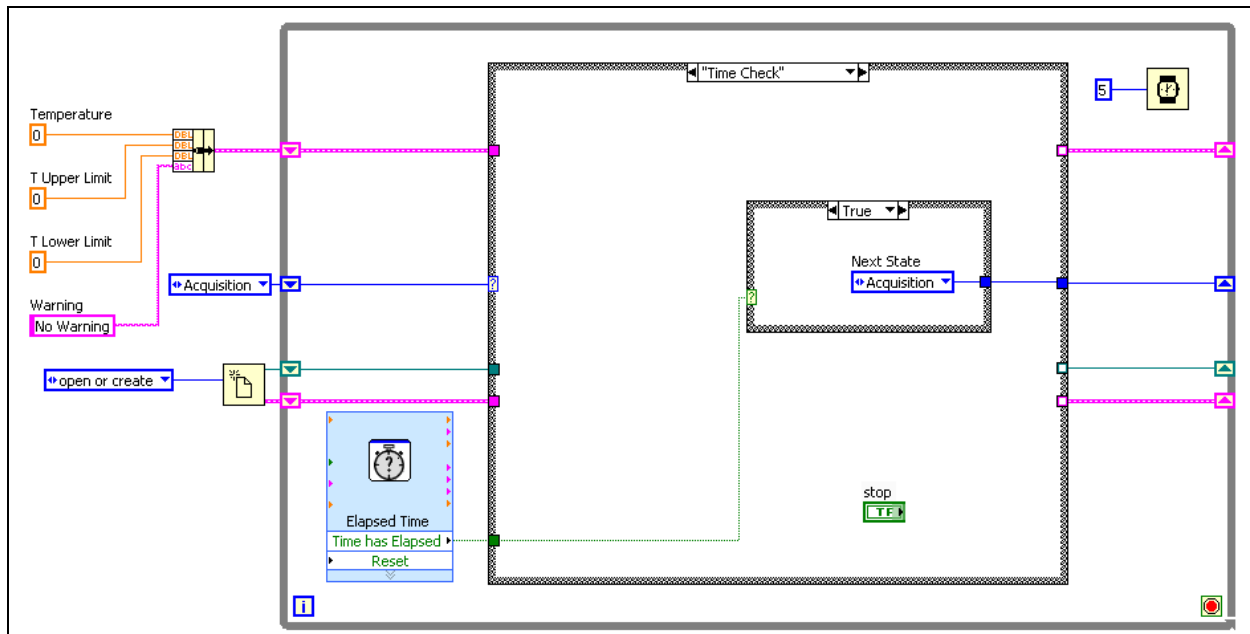


Figure 11-17. Time Check State

1. Switch to the Time Check state of the state machine.
2. Move the Stop terminal inside the Time Check state.
3. Determine when half a second has passed since the last data acquisition.
 - Place the Elapsed Time Express VI on the block diagram inside the While Loop, but outside the Case structure.
 - Configure the **Elapsed Time** to 0.5 seconds

- Remove the checkmark from the **Automatically reset after time target** checkbox.
 - Click the **OK** button.
4. Navigate to the Acquisition state if the time has elapsed.
 - Place a Case structure inside the Time Check state.
 - Wire the **Time has Elapsed** terminal from the Elapsed Time Express VI to the case selector.
 - Switch to the True case.
 - Place a copy of the Weather Station State constant inside the True case.
 - Select the Acquisition element of the constant .
 - Wire the constant to the state tunnel of the Case structure.
 5. Navigate to the Time Check state if the time has not elapsed.
 - Switch to the False case.
 - Place a copy of the Weather Station State constant inside the False case.
 - Select the Time Check element of the constant.
 - Wire the constant to the state tunnel of the Case structure.
 6. Save the VI.

Finish the State Machine

1. Wire the temperature data cluster, the error cluster and the file refnum through all states.
2. Determine when to stop the state machine. Stop when an error occurs or the user has clicked the stop button, but only after a full sequence has occurred (Acquisition, Analysis, Data Log).
 - Place an Unbundle by Name function inside the While Loop to the right of the Case structure.
 - Wire the error cluster to the Unbundle by Name function.
 - Place the Or function next to the loop condition terminal.

- Wire the **status** element of the error cluster to the top input of the Or function.
 - From the Time Check state, wire the **Stop** button to the bottom input of the Or function.
 - Wire the output of the Or function to the loop condition terminal.
 - Switch to the Acquisition state.
 - Right-click the tunnel attached to the bottom input of the Or function and select **Create»Constant** from the shortcut menu to wire a False constant to the Boolean tunnel.
 - Switch to the Analysis state.
 - Right-click the tunnel attached to the bottom input of the Or function and select **Create»Constant** from the shortcut menu to wire a False constant to the Boolean tunnel.
 - Switch to the Data Log State.
 - Right-click the tunnel attached to the bottom input of the Or function and select **Create»Constant** from the shortcut menu to wire a False constant to the Boolean tunnel.
3. Set the Elapsed Time Express VI to start counting when the VI begins.
 - Place a True constant outside the While Loop, to the left of the Elapsed Time Express VI.
 - Wire the constant to the **Reset** terminal of the Elapsed Time Express VI.
 - Replace the Reset tunnel with a shift register.
 4. Document the block diagram.
 5. Figures 11-18 through 11-21 show an example of the final block diagram for each state. Complete any block diagram wiring that is still necessary. Pay particular attention to the Booleans in the lower right of each case. These Booleans determine when the state machine stops, and when the Elapsed Time Express VI count is reset.

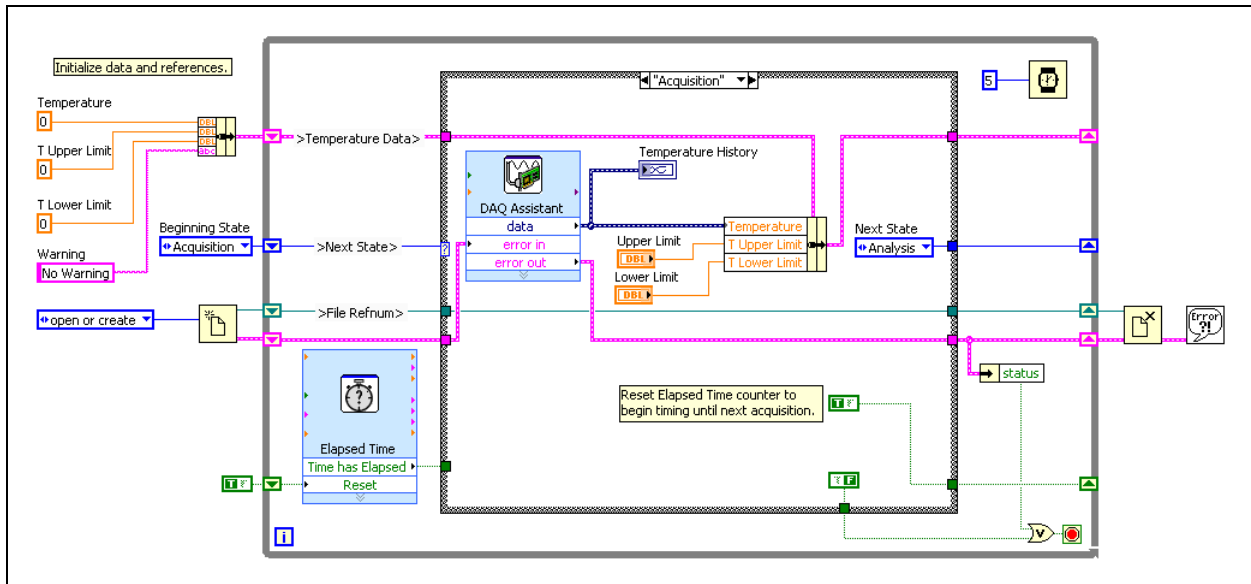


Figure 11-18. Complete Acquisition State

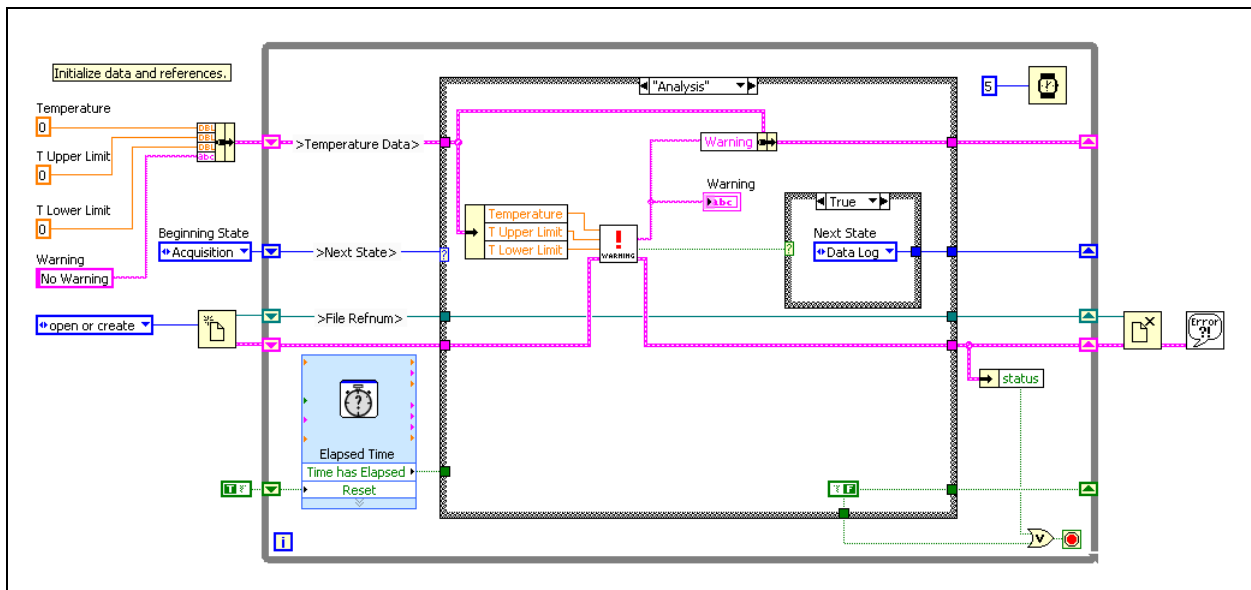


Figure 11-19. Complete Analysis State

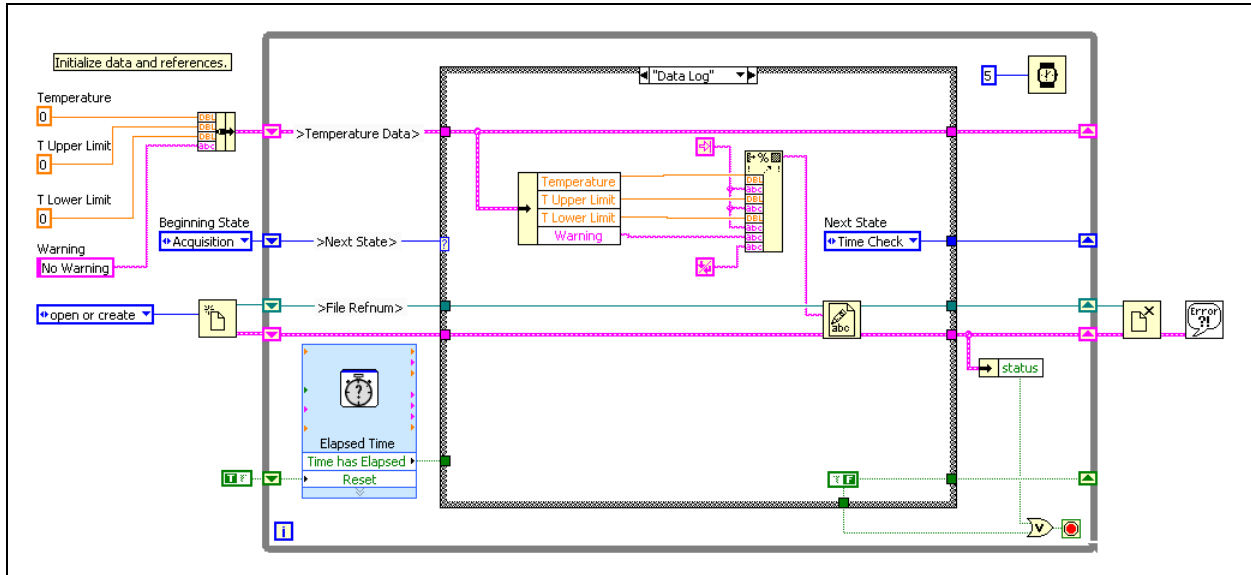


Figure 11-20. Complete Data Log State

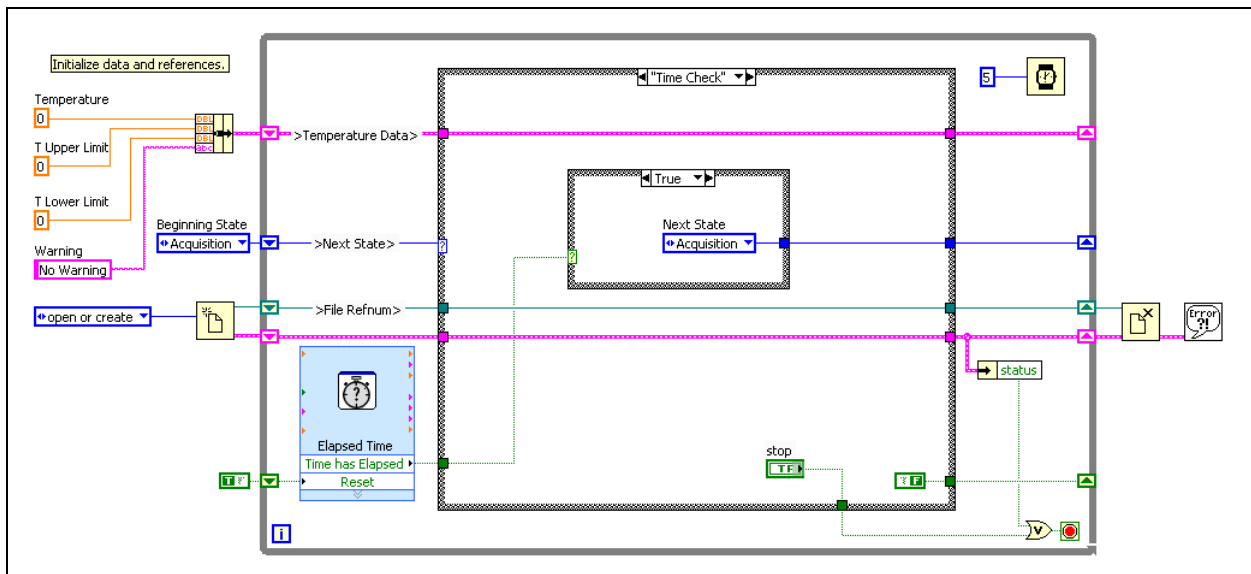


Figure 11-21. Complete Time Check State

6. Switch to the front panel.
7. Save the VI.
8. Save the project.

Testing

1. Run the VI.
2. Enter 30 in the **Upper Limit**.

3. Enter 20 to the **Lower Limit**.
4. When prompted, provide a name and a location for the file.
5. Modify the **Lower Limit** until it is higher than the current temperature reading.
6. The Freeze Warning should appear.
7. Set the **Lower Limit** back to 20.
8. Modify the **Upper Limit** until it is lower than the current temperature reading.
9. The Heatstroke Warning should appear.
10. Set the **Upper Limit** to 15.
11. The Upper Limit < Lower Limit Warning should appear.
12. Stop the VI.
13. Open Microsoft Excel.
14. Select **File»Open**.
15. Navigate to the file created by the VI.
16. Select **Open**.
17. Examine the file created by the VI.
18. Close the file when you are finished.
19. Close the VI.

End of Exercise 11-1

D. Parallelism

Often, you need to program multiple tasks so that they execute at the same time. In LabVIEW tasks can run in parallel if they do not have a data dependency between them, and if they are not using the same shared resource. An example of a shared resource is a file, or an instrument.

You learn about LabVIEW design patterns for executing multiple tasks at the same time in the *LabVIEW Basics II* course. These design patterns include parallel loops, master/slave and producer/consumer.

Summary

The benefits of using a state machine instead of a sequential programming structure include:

- You can change the order of the sequence.
- You can repeat individual items in the sequence.
- You can set condition to determine when an item in the sequence should execute.
- You can stop the program at any point in the sequence.

Notes

Additional Information and Resources

This appendix contains additional information about National Instruments technical support options and LabVIEW resources.

National Instruments Technical Support Options

Visit the following sections of the National Instruments Web site at ni.com for technical support and professional services.

- **Support**—Online technical support resources at ni.com/support include the following:
 - **Self-Help Resources**—For answers and solutions, visit the award-winning National Instruments Web site for software drivers and updates, a searchable KnowledgeBase, product manuals, step-by-step troubleshooting wizards, thousands of example programs, tutorials, application notes, instrument drivers, and so on.
 - **Free Technical Support**—All registered users receive free Basic Service, which includes access to hundreds of Application Engineers worldwide in the NI Developer Exchange at ni.com/exchange. National Instruments Application Engineers make sure every question receives an answer.

For information about other technical support options in your area, visit ni.com/services or contact your local office at ni.com/contact.

- **System Integration**—If you have time constraints, limited in-house technical resources, or other project challenges, National Instruments Alliance Partner members can help. The NI Alliance Partners joins system integrators, consultants, and hardware vendors to provide comprehensive service and expertise to customers. The program ensures qualified, specialized assistance for application and system development. To learn more, call your local NI office or visit ni.com/alliance.

If you searched ni.com and could not find the answers you need, contact your local office or NI corporate headquarters. Phone numbers for our worldwide offices are listed at the front of this manual. You also can visit the Worldwide Offices section of ni.com/niglobal to access the branch office Web sites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.

Other National Instruments Training Courses

National Instruments offers several training courses for LabVIEW users. These courses continue the training you received here and expand it to other areas. Visit ni.com/training to purchase course materials or sign up for instructor-led, hands-on courses at locations around the world.

National Instruments Certification

Earning an NI certification acknowledges your expertise in working with NI products and technologies. The measurement and automation industry, your employer, clients, and peers recognize your NI certification credential as a symbol of the skills and knowledge you have gained through experience. areas. Visit ni.com/training for more information about the NI certification program.

LabVIEW Resources

This section describes how you can receive more information regarding LabVIEW.

LabVIEW Publications

The following publications offer more information about LabVIEW.

LabVIEW Technical Resource (LTR) Newsletter

Subscribe to *LabVIEW Technical Resource* to discover tips and techniques for developing LabVIEW applications. This quarterly publication offers detailed technical information for novice users and advanced users. In addition, every issue contains a disk of LabVIEW VIs and utilities that implement methods covered in that issue. To order the *LabVIEW Technical Resource*, contact LTR publishing at (214) 706-0587 or visit www.ltrpub.com.

LabVIEW Books

Many books have been written about LabVIEW programming and applications. The National Instruments Web site contains a list of all the LabVIEW books and links to places to purchase these books.

info-labview Listserve

`info-labview` is an email group of users from around the world who discuss LabVIEW issues. The list members can answer questions about building LabVIEW systems for particular applications, where to get instrument drivers or help with a device, and problems that appear.

To subscribe to info-labview, send email to:
info-labview-on@labview.nhmfl.gov

To subscribe to the digest version of info-labview, send email to:
info-labview-digest@labview.nhmfl.gov

To unsubscribe to info-labview, send email to:
info-labview-off@labview.nhmfl.gov

To post a message to subscribers, send email to:
info-labview@labview.nhmfl.gov

To send other administrative messages to the info-labview list manager,
send email to:
info-labview-owner@nhmfl.gov

You might also want to search previous email messages at:
www.searchVIEW.net

- The info-labview web page is available at
www.info-labview.org

Index

A

arrays
 2D, 4-3
 auto-indexing, 4-4
 creating 2D arrays using auto-indexing, 4-7
 creating constants, 4-4
 creating controls and indicators, 4-3
 dimensions, 4-2
 examples of 1D arrays, 4-2
 examples of 2D arrays, 4-3
 restrictions, 4-2

auto-indexing
 creating 2D arrays, 4-7
 using to set For Loop count, 4-5

automatic wires, 2-22

automatic wiring, 2-22

B

block diagram
 automatic wiring, 2-22
 creating cluster constants, 4-16
 data flow, 2-39
 nodes, 2-19
 toolbar, 2-23
 wiring automatically, 2-22
 wiring to charts, 3-56

Boolean controls and indicators, 2-13

Breakpoint tool
 conditional breakpoints with probes, 5-8
 debugging VIs, 5-8

broken VIs
 common causes, 5-3
 correcting, 2-45, 5-2
 displaying errors, 5-2

Bundle By Name function, 4-18

Bundle function, 4-18

C

callers
 chain of, 5-10
 displaying, 5-10

Case structures
 Boolean cases, 3-68
 enum cases, 3-70
 error cases, 5-14
 executing, 3-66
 integer cases, 3-69
 selecting cases, 3-67
 specifying a default case, 3-66
 string cases, 3-69

certification (NI resources), A-2

chain of callers
 displaying, 5-10

charts
 update mode, 3-55
 wiring, 3-56

clusters
 assembling from individual elements, 4-18
 constants, 4-15
 creating, 4-15
 creating constants, 4-16
 disassembling, 4-19
 error, 5-12
 order, 4-16
 order of elements, 4-16
 replacing or accessing elements, 4-18
 wire patterns, 4-15

coercion
 numeric data, 3-42

communicating with instruments, 9-29
 serial, 9-3
 VISA, 9-23

conditional breakpoints
 See probes

conditional terminals, 3-32

configuration

- software (Windows), 9-7
- connector pane
 - assigning terminals to controls and indicators, 6-8
 - configuring, 6-6
 - modifying layout, 6-7
- connector panes, 6-4
 - setting inputs and outputs, 6-9
- constants
 - arrays, 4-4
 - clusters, 4-15, 4-16
- Context Help window, 2-57
 - terminal appearance, 6-9
- Control Editor window, 4-29
- control flow programming model, 2-39
- controls
 - assigning to connector pane, 6-8
 - string display types, 3-19
 - type definitions, 4-31
- controls and indicators
 - Boolean, 2-13
 - creating arrays, 4-3
 - creating clusters, 4-15
 - numeric, 2-13
 - showing terminals, 2-22
- conversion
 - numeric, 3-42
- correcting
 - broken VIs, 2-45, 5-2
- count terminals, 3-41
- creating
 - clusters, 4-15

D

- data coercion
 - numeric, 3-42
- data flow
 - See* dataflow
- dataflow
 - observing, 5-5
- dataflow programming model, 2-39
- debugging
 - automatic error handling, 5-12

- broken VIs, 2-45, 5-2
- error handling, 5-12
- probes, 5-7
 - single-stepping, 5-6
 - suspending execution, 5-9
 - using execution highlighting, 5-5
 - using probes, 5-6
 - using the Breakpoint tool, 5-8
- default cases, 3-66
- default probes, 5-7
- device range
 - ADC precision, 7-13
 - description, 7-13
- diagnostic tools (NI resources), A-1
- dimensions
 - arrays, 4-2
- directory paths
 - See* probes
- displaying
 - chain of callers, 5-10
 - errors, 5-2
- documentation
 - LabVIEW Help, 2-58
 - NI resources, A-1
- drivers (NI resources), A-1

E

- editing
 - icons, 6-5
- Embedded Project Manager, 2-8
- enhanced probes
 - See* probes
- error handling
 - using Case structures, 5-14
- errors
 - automatically handling, 5-12
 - broken VIs, 2-45, 5-2
 - clusters, 5-12
 - codes, 5-12
 - displaying, 5-2
 - finding, 5-2
 - handling, 5-12

- handling automatically, 5-12
- handling using While Loops, 5-14
- I/O, 5-13
- list, 5-2
- methods to handle, 5-12
- window, 5-2

examples (NI resources), A-1

execution

- debugging VIs, 5-5
- flow, 2-39
- highlighting, 5-5
- suspending to debug VIs, 5-9

Execution Highlighting button, 2-23

F

file I/O

- basic operation, 10-9
- overview, 10-11
- refnums, 10-9

finding

- errors, 5-2

fixing

- VIs, 2-45, 5-2

flow of execution, 2-39

For Loops

- auto-indexing arrays, 4-4
- count terminals, 3-41
- iteration terminals, 3-41
- setting count using auto-indexing, 4-5
- stacked shift registers, 3-50

Formula Nodes

- entering C-like statements, 3-78
- entering equations, 3-78

front panel

- array controls and indicators, 4-3
- creating cluster controls and indicators, 4-15

G

GPIB

- communicating with instruments, 9-29
- configuration software (Windows), 9-7

- software architecture, 9-7

graphs

- configuring, 3-57
- multiple-plot waveform, 3-58
- single-plot waveform, 3-58
- XY, 6-2

grouping

- data in arrays, 4-2
- data in clusters, 4-15
- data in strings, 3-18

H

help, 2-58

- technical support, A-1

highlighting execution

- debugging VIs, 5-5

I

I/O

- communicating with instruments, 9-29
- error, 5-13
- GPIB software architecture, 9-7
- serial, 9-3
- serial hardware overview, 9-6
- VISA, 9-23

icons, 6-4

- creating, 6-4

incrementally running VIs, 5-6

indicators

- assigning to connector pane, 6-8
- string display types, 3-19
- type definitions, 4-31

indicators and controls

- Boolean, 2-13
- creating arrays, 4-3
- creating clusters, 4-15
- numeric, 2-13
- showing terminals, 2-22

inputs

- setting, 6-9

instances of subVIs

- determining, 5-10

- suspending execution, 5-9
- instrument drivers (NI resources), A-1
- instruments
 - types, 9-2
- iteration terminals
 - For Loops, 3-41
 - While Loops, 3-33

K

- KnowledgeBase, A-1

L

- Labeling tool, 2-32
- LabVIEW Help, 2-58

M

- multiple-plot waveform graphs, 3-58
- multiplot XY graphs, 3-60

N

- National Instruments support and services, A-1
- NI
 - Certification, A-2
- nodes, 2-19
- numeric controls and indicators, 2-13
- numeric conversion, 3-42

O

- objects
 - wiring automatically on block diagram, 2-22
- online help, 2-58
- Operating tool, 2-29
- order
 - in clusters, 4-16
- order of cluster elements, 4-16
- order of execution, 2-39
- outputs
 - setting, 6-9

P

- palettes
 - Tools, 2-29
- Positioning tool, 2-30
- Probe tool
 - See* probes
- probes
 - debugging VIs, 5-7
 - default, 5-7
 - generic, 5-7
 - indicators, 5-7
 - supplied, 5-7
 - types of, 5-7
- programming examples (NI resources), A-1
- Project Explorer window, 2-8

R

- Read from Measurement File VI, 10-12
- Read From Spreadsheet File VI, 10-11
- refnums
 - file I/O, 10-9

S

- scope chart, 3-55
- SCXI
 - signal conditioning
 - amplification, 7-6
 - linearization, 7-7
 - phenomena and transducers (table), 7-4
 - transducer excitation, 7-7
- searching
 - for controls, VIs, and functions, 2-26
- serial port communication, 9-3
 - hardware overview, 9-6
- shift registers
 - stacked, 3-50
- signal conditioning
 - See also* SCXI
 - amplification, 7-6
 - linearization, 7-7
 - phenomena and transducers (table), 7-4

- transducer excitation, 7-7
- single-plot waveform graphs, 3-58
- single-plot XY graphs, 3-60
- single-stepping
 - debugging VIs, 5-6
- smart probes
 - See* probes
- software (NI resources), A-1
- state machine
 - design pattern
 - Case structure, 11-8
 - controlling
 - default transition, 11-7
 - enumerated type control, 11-7
 - multiple state transitions, 11-8
 - two-state transition, 11-7
 - State Diagram Toolkit, 11-9
 - transition array, 11-9
- infrastructure, 11-6
- Step Into button, 2-23
- Step Out button, 2-23
- stepping through VIs
 - debugging VIs, 5-6
- strict type definitions, 4-32
- strings, 3-18
 - display types, 3-19
- strip chart, 3-55
- structures
 - Case, 3-66
 - stacked shift registers, 3-50
 - tunnels, 3-67
- subVIs
 - assigning controls and indicators to connector pane, 6-8
 - configuring the connector pane, 6-6
 - creating icons, 6-4
 - determining current instance, 5-10
 - displaying chain of callers, 5-10
 - modifying connector pane, 6-7
 - setting inputs and outputs, 6-9
 - suspending execution, 5-9
- supplied probes, 5-7
- support

- technical, A-1
- suspending execution, 5-9
- sweep chart, 3-55

T

- technical support, A-1
- terminals, 2-19
 - conditional, 3-32
 - Context Help window appearance, 6-9
 - count, 3-41
 - iteration on For Loops, 3-41
 - iteration on While Loops, 3-33
 - optional, 6-9
 - recommended, 6-9
 - required, 6-9
 - selector, 3-66
 - showing on block diagram, 2-22
- toolbar
 - block diagram, 2-23
- tools
 - Labeling, 2-32
 - Operating, 2-29
 - positioning, 2-30
 - wiring, 2-32
- Tools palette, 2-29
- training (NI resources), A-2
- transducers
 - excitation, 7-7
 - linearization, 7-7
 - phenomena and transducers (table), 7-4
- troubleshooting (NI resources), A-1
- tunnels, 3-67
- type definitions, 4-31
 - strict, 4-32

U

- Unbundle By Name function, 4-18
- Unbundle function, 4-18
- undefined data
 - preventing, 5-11
- user probes
 - See* probes

V

VIs

- broken, 2-45, 5-2
- configuring the connector pane, 6-6
- correcting, 2-45, 5-2
- error handling, 5-12
- icons, 2-19, 6-2
- nodes, 2-19

VISA, 9-23

- terminology, 9-23

W

waveform charts

- wiring, 3-56

waveform graphs

- multiple-plot, 3-58
- single-plot, 3-58

Web resources, A-1

While Loops

- auto-indexing arrays, 4-4
- conditional terminals, 3-32
- error handling, 5-14
- iteration terminals, 3-33
- stacked shift registers, 3-50

wires

- automatic, 2-22

wiring

- automatically, 2-22
- charts, 3-56
- tunnels, 3-67

Wiring tool, 2-32

Write to Measurement File VI, 10-11

Write to Spreadsheet File VI, 10-11

X

XY graphs

- configuring, 3-57
- multiplot, 3-60
- single-plot, 3-60

Course Evaluation

Course _____

Location _____

Instructor _____ Date _____

Student Information (optional)

Name _____

Company _____ Phone _____

Instructor

Please evaluate the instructor by checking the appropriate circle. Unsatisfactory Poor Satisfactory Good Excellent

Instructor's ability to communicate course concepts

Instructor's knowledge of the subject matter

Instructor's presentation skills

Instructor's sensitivity to class needs

Instructor's preparation for the class

Course

Training facility quality

Training equipment quality

Was the hardware set up correctly? Yes No

The course length was Too long Just right Too short

The detail of topics covered in the course was Too much Just right Not enough

The course material was clear and easy to follow. Yes No Sometimes

Did the course cover material as advertised? Yes No

I had the skills or knowledge I needed to attend this course. Yes No If no, how could you have been better prepared for the course? _____

What were the strong points of the course? _____

What topics would you add to the course? _____

What part(s) of the course need to be condensed or removed? _____

What needs to be added to the course to make it better? _____

How did you benefit from taking this course? _____

Are there others at your company who have training needs? Please list. _____

Do you have other training needs that we could assist you with? _____

How did you hear about this course? NI Web site NI Sales Representative Mailing Co-worker
 Other _____

