LabVIEW™ Core 3 Course Manual

Course Software Version 2013
August 2013 Edition
Part Number 375510C-01

Worldwide Technical Support and Product Information

`ni.com`

Worldwide Offices

Visit `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.

National Instruments Corporate Headquarters

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

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. NI Certification v
B. Course Description vi
C. What You Need to Get Started vi
D. Installing the Course Software vii
E. Course Goal vii

Lesson 1
Developing Successful Applications

A. Scalable, Readable, and Maintainable VIs 1-2
B. Software Development Process Overview 1-4
C. Requirements 1-11
D. Task Analysis 1-12

Lesson 2
Organizing the Project

A. Project Libraries 2-2
B. Project Explorer Tools and Organization 2-6
C. Project Conflicts 2-10

Lesson 3
Creating an Application Architecture

A. Architecture Testing 3-2
B. LabVIEW Style Guidelines 3-5
C. User Events 3-9
D. Queued Message Handler 3-13
E. Application Data Types 3-16
F. Notifiers 3-19

Lesson 4
Customizing the User Interface

A. User Interface Style Guidelines 4-2
B. User Interface Prototypes 4-10
C. Customizing a User Interface 4-11
D. Extending a User Interface 4-17
E. Window Appearance 4-19
F. User Documentation 4-21
G. User Interface Initialization 4-23
H. User Interface Testing 4-27

Lesson 5
Managing and Logging Errors
A. Error Testing 5-2
B. Local Error Handling 5-3
C. Global Error Handling 5-5
D. Error Logging 5-9

Lesson 6
Creating Modular Code
A. Designing Modular Applications 6-2
B. Code Module Testing 6-3
C. Integration Testing 6-5

Appendix A
Boiler Controller Requirements
A. Boiler Fundamentals A-1
B. General Application Requirements A-2
C. Application Requirements A-3

Appendix B
Boiler Controller User Stories

Appendix C
IEEE Requirements Documents
A. Institute of Electrical and Electronic Engineers (IEEE) Standards C-2

Appendix D
Additional Information and Resources

Student Guide

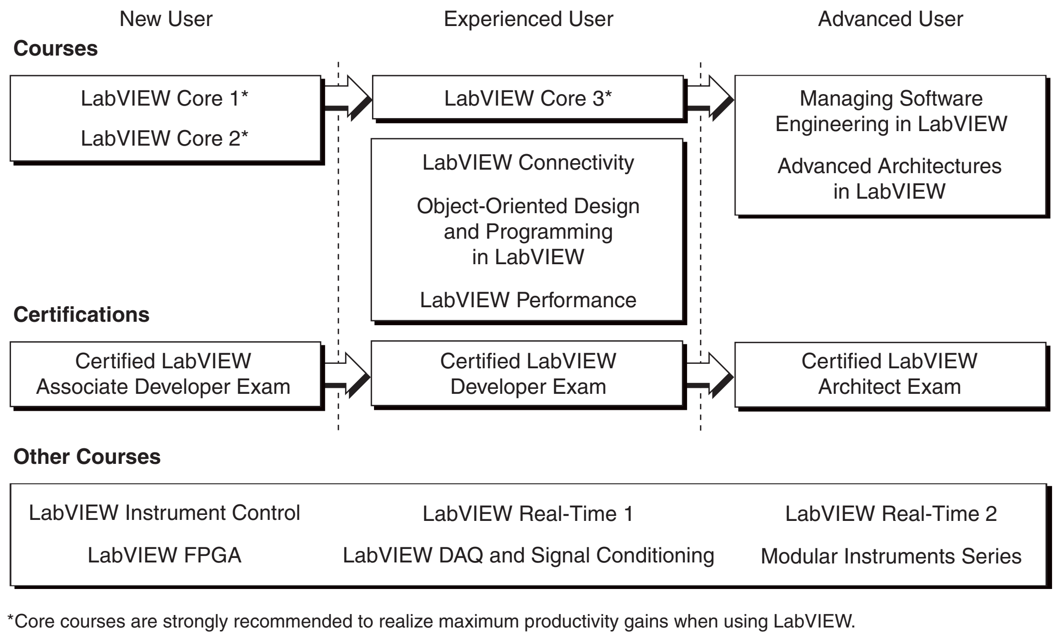
Thank you for purchasing the *LabVIEW Core 3* course kit. This course manual and the accompanying software are used in the three-day, hands-on *LabVIEW Core 3* course.

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 and exercise manual updates and corrections, refer to `ni.com/info` and enter the Info Code `core3`.

A. NI Certification

The *LabVIEW Core 3* 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.



New User
Experienced User
Advanced User
Courses
LabVIEW Core 1*
LabVIEW Core 2*
LabVIEW Core 3*
LabVIEW Connectivity
Object-Oriented Design and Programming in LabVIEW
LabVIEW Performance
Managing Software Engineering in LabVIEW
Advanced Architectures in LabVIEW
Certifications
Certified LabVIEW Associate Developer Exam
Certified LabVIEW Developer Exam
Certified LabVIEW Architect Exam
Other Courses
LabVIEW Instrument Control
LabVIEW Real-Time 1
LabVIEW Real-Time 2
LabVIEW FPGA
LabVIEW DAQ and Signal Conditioning
Modular Instruments Series
*Core courses are strongly recommended to realize maximum productivity gains when using LabVIEW.

B. Course Description

LabVIEW Core 3 introduces you to structured practices to design, implement, document, and test LabVIEW applications. This course focuses on developing hierarchical applications that are scalable, readable, and maintainable. The processes and techniques covered in this course help reduce development time and improve application stability. By incorporating these design practices early in your development, you avoid unnecessary application redesign, increase VI reuse, and minimize maintenance costs.

This course assumes that you have taken the *LabVIEW Core 1* and *LabVIEW Core 2* courses or have equivalent experience.

This course kit is designed to be completed in sequence. The course and exercise manuals are divided into lessons, described as follows.

In the course manual, each lesson consists of the following:

- An introduction that describes the purpose of the lesson and what you will learn
- A discussion of the topics in the lesson
- A summary quiz that tests and reinforces important concepts and skills taught in the lesson

In the exercise manual, each lesson consists of the following:

- A set of exercises to reinforce the topics in the lesson
- Some lessons include optional and challenge exercise sections or additional exercises to complete if time permits

Note The exercises in this course are cumulative and lead toward developing a final application at the end of the course. If you skip an exercise, use the solution VI for that exercise, available in the `<Solutions>\LabVIEW Core 3` directory, in later exercises.

C. What You Need to Get Started

Before you use this course manual, make sure you have the following items:

☐ Windows XP or later installed on your computer

☐ LabVIEW Professional Development System 2012 or later

☐ *LabVIEW Core 3* course CD, containing the following folders:

Filename	Description
`Exercises`	Folder containing VIs and other files used in the course
`Solutions`	Folder containing completed course exercises

D. Installing the Course Software

Complete the following steps to install the course software.

1. Insert the course CD in your computer.
2. Follow the prompts to install the course material.

The installer places the `Exercises` and `Solutions` folders at the top level of the root directory. Exercise files are located in the `<Exercises>\LabVIEW Core 3` directory.

Tip Folder names in angle brackets, such as `<Exercises>`, refer to folders in the root directory of your computer.

E. Course Goal

Given a requirements document for a LabVIEW development project, you will follow a software development process to design, implement, document and test the key application features in a manner that satisfies requirements for readability, scalability, and maintainability.

1 Developing Successful Applications

This lesson describes various development techniques that enable you, as a developer, to create scalable, readable, and maintainable VIs. You learn the importance of following a software development process and how to evaluate a requirements document.

Note This course approaches these topics from the perspective of a LabVIEW developer. The Managing Software Engineering in LabVIEW course approaches many of the same topics from the perspective of a LabVIEW architect.

Topics

A. Scalable, Readable, and Maintainable VIs

B. Software Development Process Overview

C. Requirements

D. Task Analysis

A. Scalable, Readable, and Maintainable VIs

LabVIEW applications can range from a simple, single VI, to extensive applications that contain many VIs organized in a complex hierarchy. As you expand the use of LabVIEW and create more complex applications, the code comprised by the applications becomes more complex.

Successful LabVIEW applications use VIs that are scalable, readable, and maintainable.

- **Scalable**—Easy to expand the design to handle more work without completely redesigning the VI.
- **Readable**—Easy to visually inspect the design of a VI and understand its purpose and functionality.
- **Maintainable**—Easy to add new features to a VI without affecting the original functionality.

When you program with LabVIEW you encounter many of the same design issues that you encounter when you program in text-based languages. However, LabVIEW provides programming techniques that enable you to focus on producing a solution to a problem rather than focusing on syntax or memory issues.

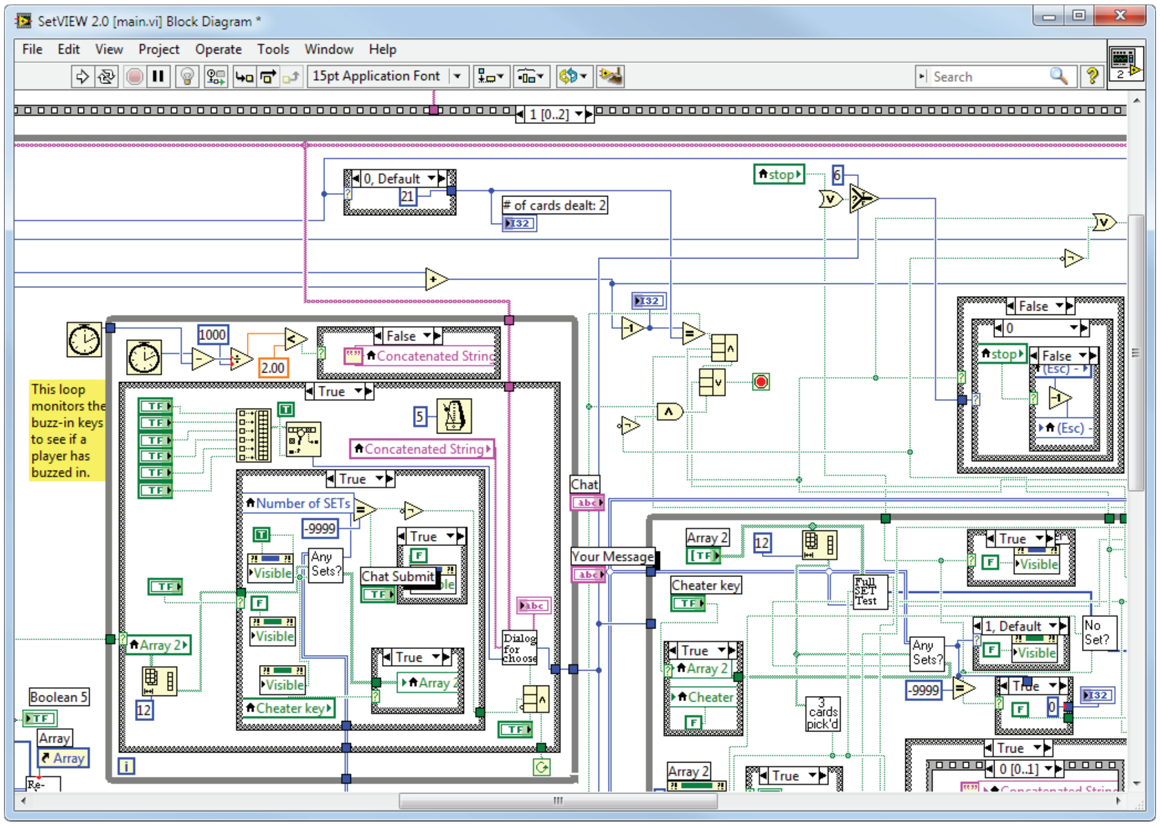
Scalable VIs

In order to create a scalable VI, begin thinking about the design of the application early in the design process. A well-designed, scalable VI allows you to easily expand the original design to handle more work. For example, consider a data acquisition VI that acquires data from three thermocouples. If the requirements of the application change and you need to acquire data from hundreds of thermocouples, being able to extend the original VI is easier than designing a new VI. When you design any application, consider the purpose of the application and how to manage changes when the scale of the application goes beyond the original specification.

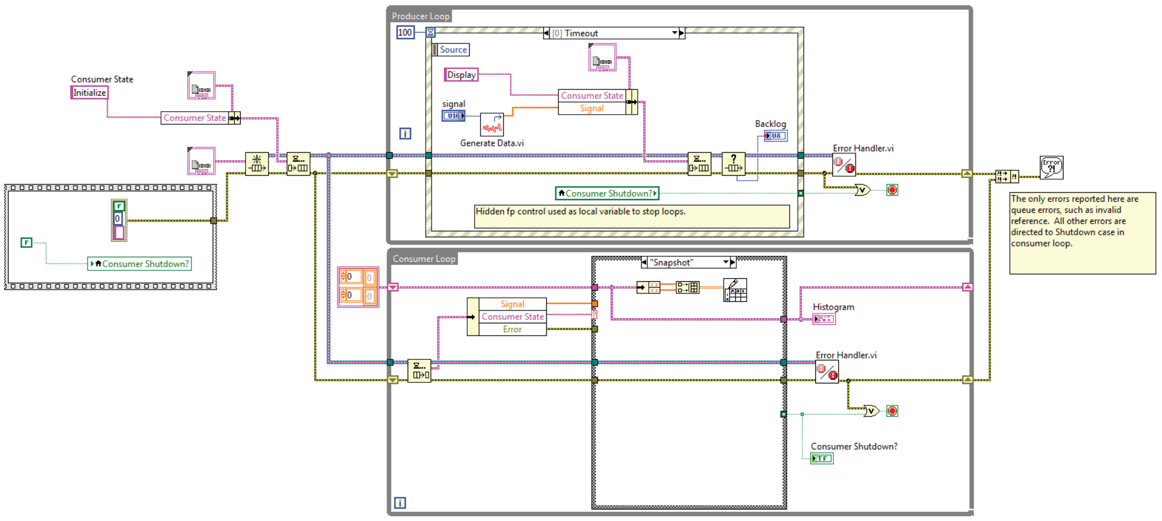
Readable VIs

In your experience working with LabVIEW, you may have seen block diagrams that were unstructured, difficult to read, and difficult to understand. Confusing and unreadable code can make it impossible to decipher the functionality of a block diagram. Figure 1-1 shows poorly a designed block diagram and a well-designed block diagram.

Figure 1-1. Examples of Poorly Designed and Well-Designed Block Diagrams


SetVIEW 2.0 [main.vi] Block Diagram *
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
This loop monitors the buzz-in keys to see if a player has buzzed in.
Concatenated String
Number of SETs
Chat Submit
Your Message
Cheater key
Any Sets?
No Set?

1


Producer Loop
Consumer Loop
Consumer State
Initialize
Consumer Shutdown?
Timeout
Source
Display
signal
Generate Data.vi
Backlog
Error Handler.vi
Hidden fp control used as local variable to stop loops.
The only errors reported here are queue errors, such as invalid reference. All other errors are directed to Shutdown case in consumer loop.
"Snapshot"
Signal
Error
Histogram

2

1 Poorly Designed	2 Well-Designed

Tip Code that is difficult to read and difficult to understand is difficult to maintain.

Maintainable VIs

When you develop an application, keep in mind that another programmer might need to use and modify the VI in the future. By using forethought in designing and creating an application, you can create VIs that are more maintainable. A VI written using good program design and architecture allows you, as well as other programmers, to add new features without completely rewriting the application.

B. Software Development Process Overview

LabVIEW makes it easy to assemble components of data acquisition, test, and control systems. Because creating applications in LabVIEW is so easy, many people begin to develop VIs immediately with relatively little planning. For simple applications, such as quick lab tests or monitoring applications, this approach can be appropriate. However, for larger development projects, good project planning is vital.

It is the software architect's responsibility to select a process model. The software developers follow the model chosen by the architect.

Software development projects can become complex. To deal with these complexities, many developers adhere to a core set of development principles. These principles form a software development process model, also known as a software life-cycle model, that describes steps to follow when developing software—from the initial concept stage to the release, maintenance, and subsequent upgrading of the software.

Many different software life-cycle models exist. Each has advantages and disadvantages in terms of time-to-release, quality, and risk management. This topic describes some of the most common models used in software engineering but engineers often customize a model to fit the requirements of a project.

Tip To learn more about selecting a software development process model, refer to the *Managing Software Engineering in LabVIEW* course.

Code and Fix Model

The code and fix model is probably the most frequently used development methodology in software engineering. It starts with little or no initial planning. You immediately start creating code, fixing problems as they occur, until the project is complete.

Code and fix is a tempting choice when you face tight development schedules because you begin developing code right away and see immediate results.

Unfortunately, if you find major architectural problems late in the process, you usually have to rewrite large parts of the application. Alternative development models can help you catch these problems in the early concept stages, when making changes is easier and less expensive.

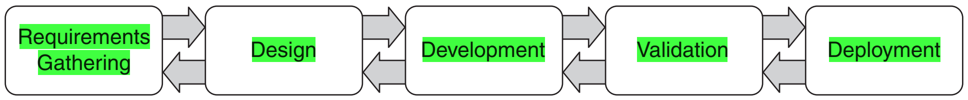
The code and fix model is appropriate only for small projects that are not intended to serve as the basis for future development.

Waterfall Model

The waterfall model is a classic model of software engineering. This model is one of the oldest models and is widely used in government projects and in many major companies. Because the model emphasizes planning in the early stages, it catches many design flaws before they develop. Also, because the model is document- and planning-intensive, it works well for projects in which quality control is a major concern.

The pure waterfall model consists of several non-overlapping stages, as shown in Figure 1-2. This model serves as a baseline for many other life cycle models.

Figure 1-2. The Waterfall Model Stages



- **Requirements Gathering**—Gather all system and software requirements. Establishes the components for building the system (hardware requirements, software tools, and other necessary components) and the expectations for software functionality.
- **Design**—Includes design of the software framework and the software components needed to meet the specified requirements.
- **Development**—Implements the detailed design specification.
- **Validation**—Determines whether the software meets the specified requirements and finds any errors present in the code.
- **Deployment**—Distribution of code to end-users.

Although the waterfall model has its weaknesses, it is instructive because it emphasizes important stages of project development. Even if you do not apply this model, consider each of these stages and its relationship to your own project.

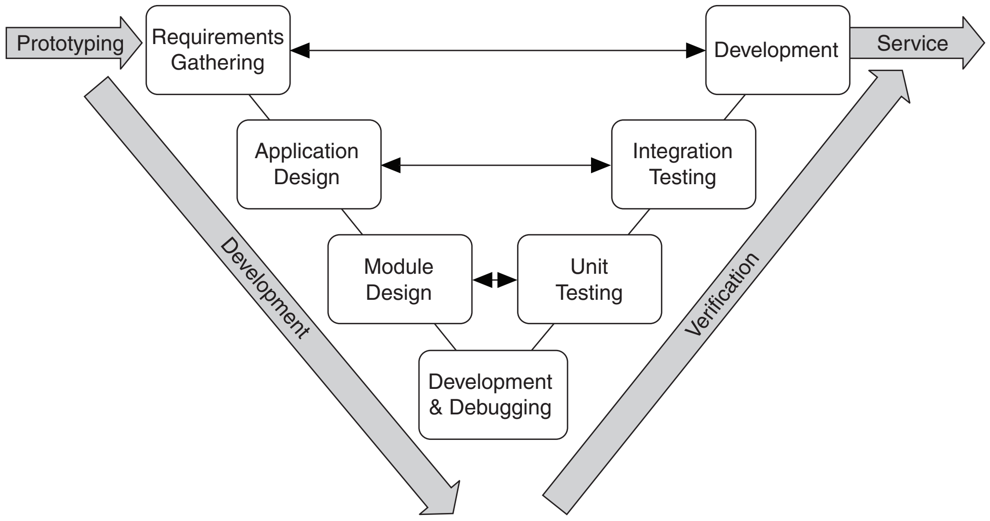
The following list outlines key aspects of the waterfall model:

- Emphasizes early planning.
- Can mitigate feature creep.
- Emphasizes documentation at each stage.
- Has deliverables at each stage of development that require a formal review and signoff from stakeholders.
- May require repetition of previously completed phases if you need to return to an earlier phase.

V-Model

The V-model is a variation of the waterfall model that bends upward after the development and debugging stage, as shown in Figure 1-3. The objective of this model is to demonstrate the relationships between the design stages of the process with their associated testing stages.

Figure 1-3. Software Engineering V-Model



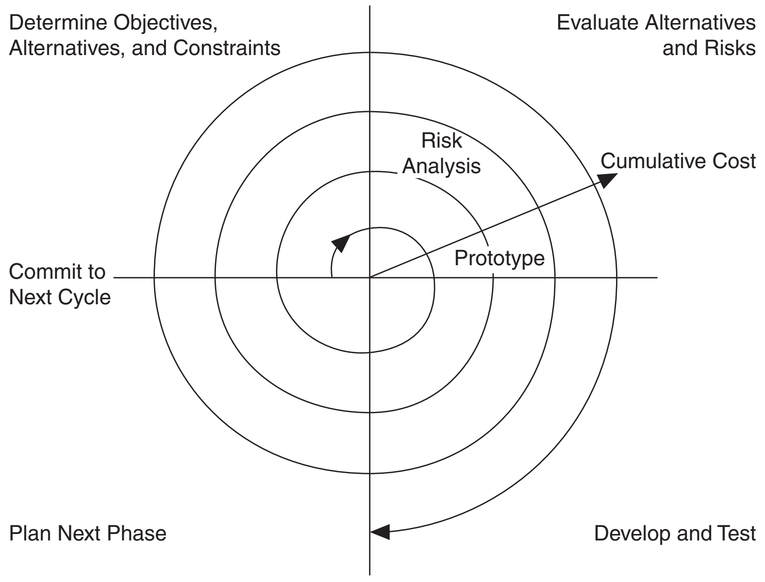
As with the waterfall model, the V-model is a structured process that requires completion and thorough documentation of previous stages. One major advantage of this model over the waterfall model is that validation activities like test design can begin earlier in the process, potentially saving a significant amount of time.

Spiral Model

The spiral model is an iterative approach to software development and emphasizes risk management. The process entails building prototypes of smaller versions of the project that become iteratively more complicated as you add additional features and functionality.

Figure 1-4 shows the spiral software life-cycle model.

Figure 1-4. The Spiral Model



Each iteration includes the following steps:

1. Determine the objectives, alternatives, and constraints on the new iteration.
2. Evaluate alternatives.
3. Identify and resolve risks.
4. Develop and test the product.
5. Plan the next iteration.

Risk assessment identifies the risks of different design issues, such as omitting or including a feature. Risk assessment can mitigate the impact of a risk on the schedule and cost of the project. Risks are any issues that are not clearly defined or have the potential to affect the project adversely. For each risk, consider the following two things:

- The likelihood of the risk occurring (probability)
- The severity of the effect of the risk on the project (impact)

As a team, assign a value of 1 to10 for the probability and impact of each risk, where 1 is the lowest probability or impact and 10 represents the highest. Risk exposure is the product of these rankings.

In general, you deal with the risks that have the highest exposure first. Using data acquisition as an example, the first spiral can deal with the potential of the data acquisition rates being too high. If after the first spiral, you demonstrate that the rates are high, you can change to a different hardware configuration to meet the acquisition requirements. However, changing hardware configuration can introduce a new risk. In this example, using more powerful hardware can introduce higher costs to the project.

In the final phase, you evaluate the results with the customer. Based on customer input, you can reassess the situation, decide on the next highest risk, and start the cycle over. This process continues until the software is finished or you decide the risks are too great and terminate development. It is possible that none of the options are viable because the options are too expensive, time-consuming, or do not meet the requirements.

The advantage of the spiral model over the waterfall model is that you can evaluate which risks to handle with each cycle. Because you can evaluate risks with prototypes much earlier than in the waterfall model, you can deal with major obstacles and select alternatives in the earlier stages, which is less expensive. With a standard waterfall model, assumptions about the risky components can spread throughout the design, and when you discover the problems, the rework involved can be very expensive.

Agile Software Development

Agile software development is another iterative approach that emphasizes collaboration, working software, and responsiveness to change. The agile model is a general approach that incorporates many different types of development methods. These methods promote development iterations, teamwork, collaboration, and process adaptability throughout the life-cycle of the project.

Agile development promotes:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

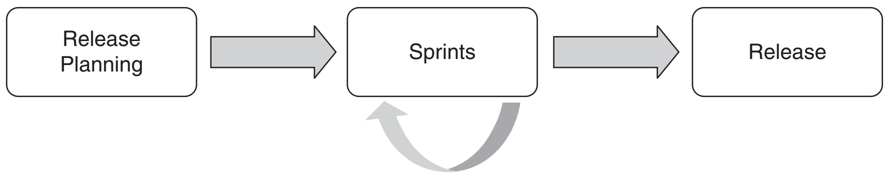
Refer to the *Manifesto for Agile Software Development* at `www.agilemanifesto.org` for more information

Because many companies favor agile methods, you can benefit by becoming familiar with the process.

Scrum Development

Scrum development is a framework to help development teams work in a more agile manner. Figure 1-5 shows the scrum development process.

Figure 1-5. Scrum Development Process



Meetings

Scrum development emphasizes daily face-to-face communication over production of elaborate documentation. If the team is spread out over a number of locations, then daily contact is maintained through e-mail, video conferencing or other methods. This daily communication prevents possible issues or problems from being hidden.

The scrum process focuses on three stages:

- Release planning stage—Create a requirements document and a ranked list of user stories for the project. A user story informally describes an aspect of the application from the perspective of the end-users.

  Refer to the *User Stories* section for more information about user stories.
- Sprint stage—Develop working product based on the user stories selected. The sprint stage involves many individual development iterations known as sprints. Refer to the *Sprints* section for more information about the sprint process.
- Release stage—Ship the final product.

Scrum Roles

Scrum teams are small (five to nine people) groups with little consideration given to the corporate roles of team members. For larger projects, multiple teams may work on different parts of the application.

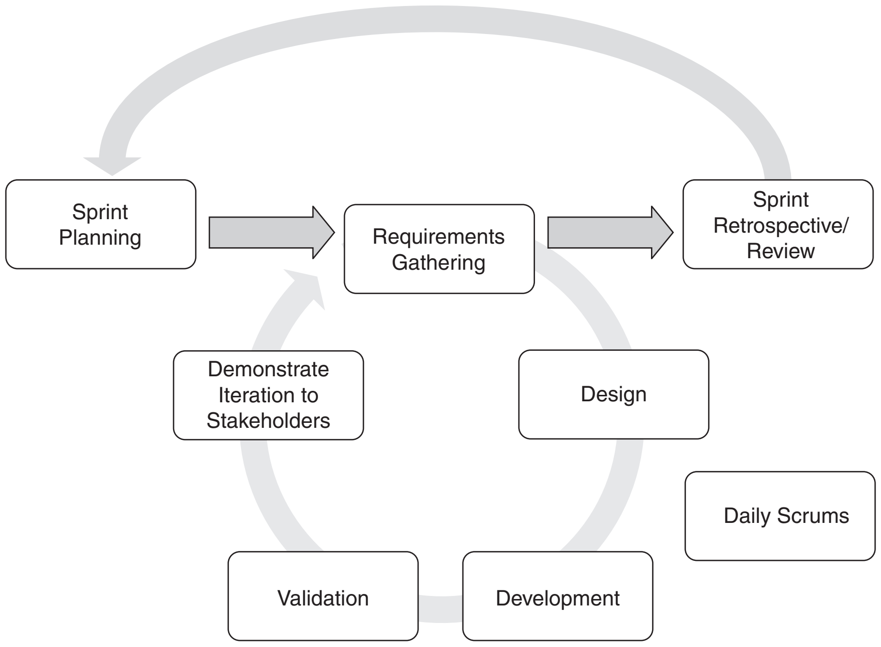
Each team includes the following roles:

- **Product owner**—Team member that is responsible for directing the team toward the product vision and acts as a liaison to stakeholders, such as the customer.
- **Scrum master**—Team member in charge of the scrum process
- **Developer**—Team member with the expertise necessary to create the product. Teams benefit by having a cross-functional group of developers.

Sprints

Agile development iterations, called sprints, have quick completion times that generally last between one and four weeks. Therefore, the agile process requires that the application be divided into smaller tasks that developers can accomplish during a sprint. At the beginning of the sprint, the team decides which tasks to complete, and team members develop the assigned tasks using a process model such as the one shown in Figure 1-6.

Figure 1-6. Agile Model Sprint Process



The sprint begins with a sprint planning meeting. In this meeting, the team identifies the tasks to complete as part of this sprint. The workload should be such that the sprint requires one to four weeks to finish. Next, the team decides on the development process for that sprint. The sprint development can follow any process to complete the chosen tasks.

Daily scrums are short, less than 15 minutes long, meetings that briefly cover the following items:

- Team progress since the last scrum meeting.
- Plan for the next day of work.
- Identification of impediments to progress.
- Individual member reports answering the following questions:
  - What have I done since the last meeting?
  - What will I do before the next meeting?
  - What, if anything, is in my way of getting things done?

At the end of each sprint, the development team should be able to demonstrate a working portion of the application to stakeholders. The software produced at the end of each iteration serves as the main measure of progress for the project.

After the team completes the tasks for this sprint, two meetings should occur:

- Sprint retrospective—the team identifies the ways to improve the sprint development process.
- Sprint review—The team identifies what tasks, if any, were not completed.

The relatively short time frame for the sprint allows the project to easily adapt to any changes in requirements or feedback from the stakeholders. Create documentation as required for each iteration, but the focus is on creating a release candidate with minimal bugs at the end of each iteration. Depending on the size of the application, the project may require multiple iterations to release a brand new product or to implement a complicated new feature.

C. Requirements

Virtually all software development processes begin with establishing requirements for the project. Requirements define the necessary pieces to develop the software and describe exactly what the software should accomplish. In general, it is the role of the software architect or product owner to meet with the customer and together, create the requirements document.

Tip The Institute of Electrical and Electronic Engineers (IEEE) defines standards for software engineering, including standards for requirements documents. Refer to Appendix A, *Boiler Controller Requirements*, for information about IEEE standards.

Functional and Non-Functional Requirements

Requirements define the necessary pieces to develop the software. Requirements are often categorized as functional and non-functional requirements.

- **Functional requirements**—Define the functions that the software must perform. Examples include:
  - Must sample data at 100 KS/s.
  - Must perform a fast Fourier transform.
  - Must store data to a database.
- **Non-Functional requirements**—Define the characteristics of implementation. Examples include:
  - System must be reliable for up to ten simultaneous users.
  - Graph style must use company colors.
  - System must be implemented by a certain date.

Other categories of requirements may include the following:

- **Data requirements**—Describes data the system will use.
- **Behavioral requirements**—Describes the behavior of the system. Useful when illustrating how an application performs and uses resources.
- **Interface requirements**—Describes the user interface requirements.
- **Output requirements**—Describes file formats, or data sink requirements.

Reviewing the Requirements Document

As a developer, it is your responsibility to review the requirements document and identify any areas that are lacking in definition or specific details. If the requirement document is not detailed enough, you might misunderstand the intent and implement code that does not meet the needs of the customer. Work with the software architect or product owner to fill in those details.

Tip For more information about the process for evaluating customer needs and identifying requirements, refer to the *Managing Software Engineering in LabVIEW* course.

To practice the concepts in this section, complete Exercise 1-1.

D. Task Analysis

Developing more complex applications requires task analysis that can realize complex interactions in the abstract before implementing them in software.

Abstraction

Abstraction intuitively describes an application without describing how to write the application. In other words, abstraction generalizes the application and eliminates details.[1] By abstracting components before you begin coding, you can more easily focus on the high-level design of the application without getting lost in functional details.

Tip Refer to the *Managing Software Engineering in LabVIEW* and *Object-Oriented Design and Programming in LabVIEW* courses for more information about the process of identifying abstract software components.

User Stories

Users stories are the type of abstraction that the scrum process uses. User stories incorporate the who, what, and why of a feature.

In the scrum process, the product owner meets with the customer and possibly the end-users to create a list of user stories for the project. Each user story informally describes an aspect of the application from the perspective of the end-user and indicates what the user needs the application to do.

When writing a user story, use the following template:

As a [*user role*], I want [*functionality*], so that [*value*].

For example:

- As a boiler operator, I want the user interface to remain responsive while boiler operations occur, so that I can always shut down the boiler in the event of an emergency.
- As a boiler operator, I want to test the functionality of the application by running it with code that simulates the behavior of a boiler.

[1] Jeri R. Hanly and Elliot B. Koffman, *Problem Solving and Program Design in C* (Reading:Addison-Wesley, 1996) 622.

Each sprint in the scrum process focuses on implementing one or more user stories.

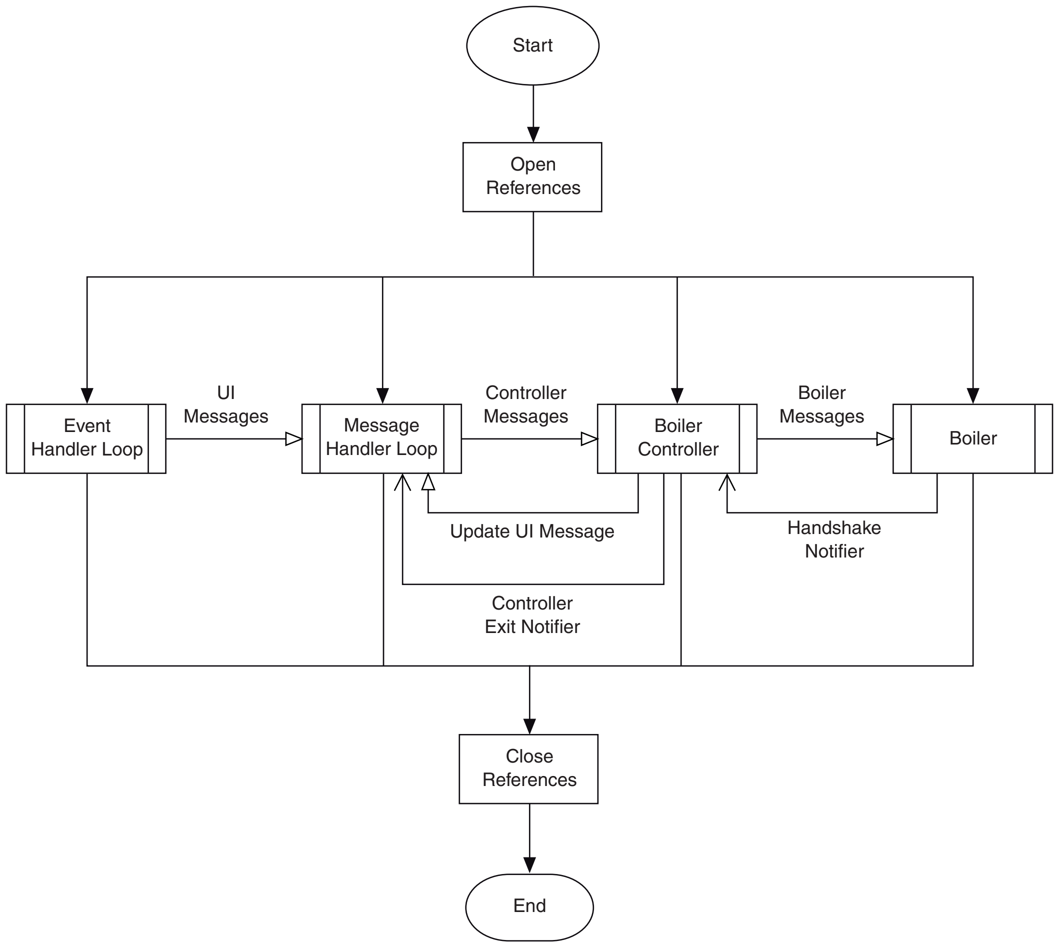
Drawing Abstracted Components

Use flowcharts to organize ideas related to a piece of software. Flowcharts illustrate the application flow. The block diagram programming paradigm used in LabVIEW is similar to a flowchart. Many engineers already use flowcharts to describe systems. The flowchart makes it easier to convert the design into executable code.

Course Project: High-Level Flowchart

Figure 1-7 illustrates an abstraction of the course project.

Figure 1-7. High-Level Flowchart of the Class Boiler Project



To practice the concepts in this section, complete Exercise 1-2.

Self-Review: Quiz

1. Match each software design principle to its definition.

Scalable	a. Easy to visually inspect the design of a VI and understand its purpose
Readable	b. Easy to expand the design to handle more work
Maintainable	c. Easy to add new features to a VI without affecting the original functionality

2. Match each software development process to the description that best fits it.

Waterfall	a. Iteratively develop with greater emphasis on face-to-face communication versus written documentation
V-model	b. Clearly define, document, and implement each phase
Spiral model	c. Iteratively develop and document a prototype that becomes more complex
Agile development	d. Demonstrates the relationship between development and testing

3. Match each document to its purpose.

Requirements document	a. Provides a common ground for the programmer and the customer to work from
List of user stories	b. Provides a good understanding of application flow
Flowchart	c. Describes aspects of the application from the perspective of the end-user

Self-Review: Quiz Answers

1. Match each software design principle to its definition.

Scalable	**b. Easy to expand the design to handle more work**
Readable	**a. Easy to visually inspect the design of a VI and understand its purpose**
Maintainable	**c. Easy to add new features to a VI without affecting the original functionality**

2. Match each software development process to the description that best fits it.

Waterfall	**b. Clearly define, document, and implement each phase**
V-model	**d. Demonstrates the relationship between development and testing**
Spiral model	**c. Iteratively develop and document a prototype that becomes more complex**
Agile development	**a. Iteratively develop with greater emphasis on face-to-face communication versus written documentation**

3. Match each document to its purpose.

Requirements document	**a. Provides a common ground for the programmer and the customer to work from**
List of user stories	**c. Describes aspects of the application from the perspective of the end-user**
Flowchart	**b. Provides a good understanding of application flow**

Notes

2 Organizing the Project

As projects get more complex, it can become difficult to manage all of the files and conflicts may arise.

This lesson describes tools and techniques for organizing and managing files in a LabVIEW project. You will learn how to create and use project libraries, how to use various tools to learn more about each LabVIEW project file, and how to identify, resolve, and prevent cross-linking errors and file conflicts.

Topics

A. Project Libraries
B. Project Explorer Tools and Organization
C. Project Conflicts

A. Project Libraries

LabVIEW project libraries are collections of related VIs, type definitions, shared variables, palette files, and other files, including other project libraries. When you create and save a new project library, LabVIEW creates a project library file (`.lvlib`), which includes the properties of the project library and paths to files that the project library owns.

Project Library Basics

Use project libraries to organize a virtual, logical hierarchy of items. A project library file does not actually contain the files it owns, unlike an LLB. Files that a project library owns still appear individually on disk in the directories where you saved them.

A project library file is an XML file containing information about the project and files, as shown in Figure 2-1.

Figure 2-1. Portion of an Project Library File Shown in Notepad

```
<?xml version='1.0' encoding='UTF-8'?>
<Library LVVersion="12008004">
        <Property Name="NI.Lib.Icon" Type="Bin">%A#!"!!!!!)!"1!&amp;!!!-!%!!!@````]!!!!"!!%!!!(]!!
81N=?"5X&lt;A91M&lt;/W-,&lt;'&amp;&lt;9+K1,7Q,&lt;)%N&lt;!NMA3X)DW?-RJ(JQ"I\%%Z,(@`BA#==ZB3RN;]28_
D7K6;G-RV3P)R`ZS%=_]J'XP/5N&lt;XH,7V\SEJ?]Z#5P?=J4HP+5JTTFWS%0?=B$DD1G(R/.1==!IT.+D)`B':\B'2Z@9XC'
S0I4&lt;*&lt;)?=:XA-(]X40-X40-VDSGC?"GC4N9(&lt;)"D2,L;4ZGG?ZH%;T&gt;-]T&gt;-]T?.S.%`T.%`T.)^&lt;NF
[;*YCK=ASI2F=)1I.Z5/Z5PR&amp;)^@54T&amp;5TT&amp;5TQO&lt;5_INJ6Z;"[(H#&gt;ZEC&gt;ZEC&gt;Z$"(*ETT*ET
+5A?0^NOS?UJ^3&lt;*\9B9GT@7JISVW7*NIFC&lt;)^:$D`5Q9TWE7)M@;V&amp;D,6;M29DVR]6#R],%GC47T9_/=@&gt;Z5
YX7ZRP6\D=LH%_8S/U_E5R_-R$I&gt;$\0@\W/VW&lt;[_"&lt;Y[X&amp;],0^^+,]T_J&gt;`J@_B_]'_.T`$KO.@I"O[^NF
        <Property Name="NI.Lib.SourceVersion" Type="Int">302022660</Property>
        <Property Name="NI.Lib.Version" Type="Str">1.0.0.0</Property>
        <Property Name="NI.LV.All.SourceOnly" Type="Bool">false</Property>
        <Property Name="NI.SortType" Type="Int">3</Property>
        <Item Name="Boiler.vi" Type="VI" URL="../Boiler.vi"/>
        <Item Name="Change Flame Level.vi" Type="VI" URL="../Change Flame Level.vi">
                <Property Name="NI.LibItem.Scope" Type="Int">2</Property>
        </Item>
        <Item Name="Change Flame Level - Next.vi" Type="VI" URL="../Change Flame Level - Next.vi">
                <Property Name="NI.LibItem.Scope" Type="Int">2</Property>
        </Item>
        <Item Name="Boiler Configuration.ctl" Type="VI" URL="../Boiler Configuration.ctl"/>
```

Tip A LabVIEW class is a special type of project library that you can use to define a new data type in LabVIEW. Refer to the *Object-Oriented Design and Programming in LabVIEW* course for more detail about LabVIEW classes.

Benefits of Using Project Libraries

Project libraries are useful if you want to do the following for your project.

- Organize files into a single hierarchy of items.
- Avoid potential VI name duplication.
- Limit public access to certain files.
- Limit editing permission for a collection of files.
- Set a default palette file for a group of VIs.

Tip You can drag items that a project library owns from the **Project Explorer** window to the block diagram or front panel.

Viewing a Project Library

You can view the structure of a project library from the **Project Explorer** window or in a stand-alone project library window. If you are not in the **Project Explorer** window, right-click a project library file and select **Open** from the shortcut menu to open it in the project library window.

Note If the project library file you select is not the top-level project library file, the stand-alone window that opens is the project library window of the top-level project library. The project library file you select is in the contents tree of the top-level project library window.

Note Only one project library can own a specific VI. However, you can associate a file not specific to LabVIEW, such as a text file or HTML file, with multiple project libraries.

Creating a Project Library

You can create project libraries from project folders. You also can convert LLBs to project libraries. LLBs have different features than project libraries, so consider the ways in which you might use an LLB before you decide whether to convert it to a project library. You can include project library files in an LLB to get the features of both.

Refer to the *Creating a Project Library* topic of the *LabVIEW Help* for more information about creating project libraries from scratch.

Creating a Project Library from a Project Folder

You can create LabVIEW project libraries from virtual folders in a LabVIEW project. The new project library owns the items that the folder contained.

From the **Project Explorer** window, right-click a virtual folder to convert and select **Convert to Library** from the shortcut menu. LabVIEW converts the folder to a project library, which appears in the **Project Explorer** window with the items the library owns listed under the library.

Note You cannot convert an auto-populated folder into a project library.

You can name the new project library file when you save it. Right-click the project library and select **Save** from the shortcut menu.

Including Palette Files in Libraries

If you include a palette file (`.mnu`) in a project library, you can set it as the default palette file for all VIs that the project library owns. The default palette file for a project library is the palette available in the shortcut menu when you right-click a sub-VI call to any VI that the project library owns, just as source palettes are available in the shortcut menus for many VIs and functions placed on the block diagram from the **Functions** palette. However, unlike source palettes, the default palette file for a project library does not have to contain any VIs from the project library it belongs to. From the **General Settings** page of the **Project Library Properties** dialog box, select the palette file in the **Default Palette** ring control. You also can set the default palette file from the **Item Settings** page. Select the `.mnu` file in the **Contents** tree and place a checkmark in the **Default Palette** checkbox.

Configuring Access Scope

You can configure access settings for VI, items, and folders that a LabVIEW project library owns. Items within a library are known as members, such as member VIs. You can configure library members as public, community, or private.

- **Public**—Any VI can call the member VI.
- **Community**—Only VIs within the same library, friends of the library, or VIs within a friend library can call the member VI. Community member VIs display a dark blue key glyph in the **Project Explorer** window.
- **Private**—Only VIs within the same library can call the member VI. Private member VIs display a red key glyph in the **Project Explorer** window.

Tip Configure most member VIs as either public or private.

Public Versus Private Access

Use project libraries to limit access to certain types of files. You can configure access scope for items and folders in a project library as public or private to prevent users from accessing certain items. When you set access for a folder as private, all VIs in that folder also have private access.

Determine which items in the project library you want to set as public and which as private. Public items might include palette VIs, instrument drivers, and other tools you want users to find and use. Private items might include support VIs, copyrighted files, or items you might want to edit later without taking the risk of breaking users' code. You cannot use a private VI as a subVI in other VIs or applications that the project library does not own.

Complete the following steps to configure access options in a project library.

1. Right-click the project library icon in the **Project Explorer** window or stand-alone project library window and select **Properties** from the shortcut menu to display the **Project Library Properties** dialog box.
2. From the **Item Settings** page, click an item in the **Contents** tree to select it. The current access settings for the item appear in the **Access Scope** box. Click the radio buttons in the **Access Scope** box to change the access scope for that item.
3. Click the **OK** button to incorporate the changes into the project library and close the dialog box.

Items set as private appear in the **Project Explorer** window with a private icon.

Protecting Project Libraries

You can limit editing permission by locking or password-protecting LabVIEW project libraries. When you lock a project library, users cannot add or remove items and cannot view items that you set as private. When you assign a password to a project library, users cannot add or remove items or edit project library properties without a password. Users can open the **Project Library Properties** dialog box, but all dialog box components except protection options are disabled. Users must unlock the project library or enter a password to enable the dialog box components.

Note Adding password protection to a project library does not add password protection to the VIs it owns. You must assign password protection to individual VIs.

Right-click the project library icon in the **Project Explorer** or stand-alone project library window and select **Properties** from the shortcut menu to display the **Project Library Properties** dialog box. To set the protection for the project library, select the General Settings pane in the Project Library Properties dialog box. From there, you set the library as one of the following:

- **Unlocked (no password)**—Users can view public and private items that the project library owns and can edit the project library and its properties.
- **Locked (no password)**—Users cannot add or remove items from the project library, edit project library properties, or view private items that the project library owns. For example, if you are developing a project library and do not want anyone to view private files, you should lock the project library.
- **Password-protected**—Users cannot add or remove items from the project library, edit project library properties, or view private items that the project library owns. Users must enter a password to edit the project library. For example, if you are developing a project library and want only a few people on the development team to have editing permission, set a password for the project library and give the password to those people.

Using Project Libraries

You can create an organizational structure for files that a LabVIEW project library owns. A well-organized structure for project library items can make it easier for you to use source control, avoid filename conflicts, and divide the project library into public and private access areas.

The following list describes some of the caveats and recommendations to consider when you organize project libraries and the files that the project libraries own.

- Create each project library within a separate LabVIEW project that contains only files related to that project library, including example files and the files you use to create and test the project library. Give the project and project library similar filenames. If a project library includes several separate areas of functionality, consider using project sublibraries for each area.
- Create a separate directory of files for each project library you create. You can include the files that the project library owns in the directory. If you include files for more than one project library in the same directory, conflicts might occur if you try to include VIs of the same name in different libraries. Organizing project library files into separate directories makes it easier to identify files related to specific project libraries on disk.

- If you move files on disk that a project library owns, re-open and re-save the project library to ensure that the project library links correctly to the moved items.
- **(Windows)** If you are building an installer that includes a project library, make sure you save the files that the project library owns on the same drive as the project library. If some files are on a different drive, such as a network drive, project library links will break if you include the project library in an installer.
- Determine which items in a project library you want to set as private and which as public. Users cannot use private VIs as subVIs in other VIs or applications. Public items provide the interface to the project library functionality and might include palette VIs, instrument drivers, and tools you want users to find and use. Private items might include support VIs, copyrighted files, or items you might want to edit later without taking the risk of breaking users' code. Consider the following recommendations.
  - Create a folder in the project library named private. From the **Item Settings** page of the **Project Library Properties** dialog box, configure the access settings as private for the folder. LabVIEW automatically sets as private any project library files you add to the private folder, so you do not have to configure access settings for individual VIs.
  - Assume that all project library files that are not in the private folder are public. You do not need to create a folder for public files.
  - You also can organize public and private items in a project library by creating folders for each functionality group within a project library and adding a private subfolder within each functionality group folder.

Note If a project contains a project library and an auto-populating folder, any file that is part of that library and resides in the auto-populated folder appears under the project library only because the hierarchy of the project library takes precedence over the hierarchy of the auto-populated folder.

To practice the concepts in this section, complete Exercise 2-1.

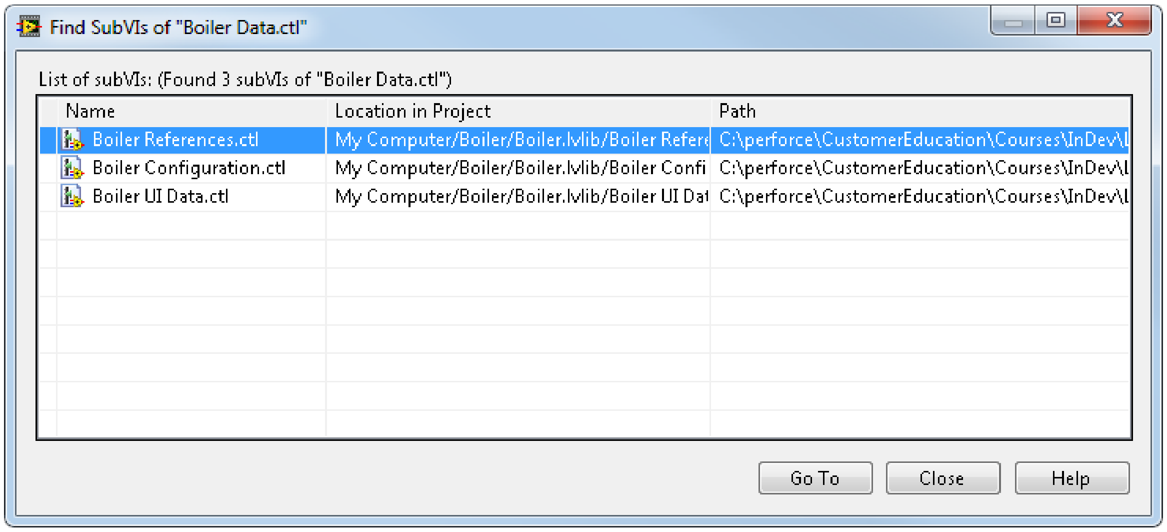
B. Project Explorer Tools and Organization

The **Project Explorer** window has multiple tools to help you navigate items and dependencies in complex projects.

Find SubVIs

Right-click an item in the **Project Explorer** window and select **Find»SubVIs** from the shortcut menu to display the **Find SubVIs** dialog box. Use this dialog box to find all subVIs of a specific item in the project.

Figure 2-2. Find SubVIs Dialog Box


Find SubVIs of "Boiler Data.ctl"

List of subVIs: (Found 3 subVIs of "Boiler Data.ctl")

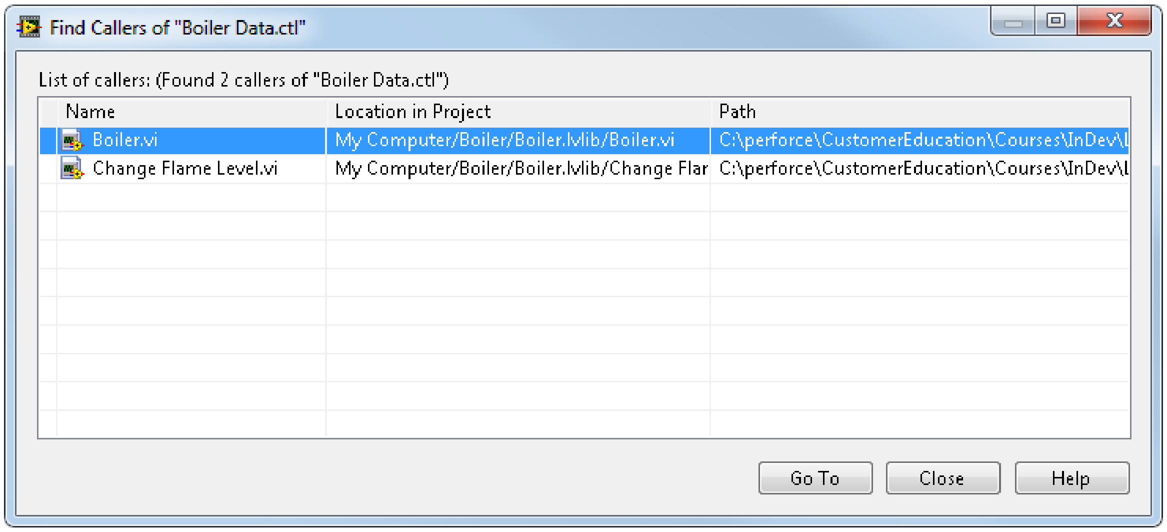
Name	Location in Project	Path
Boiler References.ctl	My Computer/Boiler/Boiler.lvlib/Boiler Refere	C:\perforce\CustomerEducation\Courses\InDev\L
Boiler Configuration.ctl	My Computer/Boiler/Boiler.lvlib/Boiler Confi	C:\perforce\CustomerEducation\Courses\InDev\L
Boiler UI Data.ctl	My Computer/Boiler/Boiler.lvlib/Boiler UI Dat	C:\perforce\CustomerEducation\Courses\InDev\L

If the item has only one caller, LabVIEW highlights the caller in the **Project Explorer** window.

Find Callers

Right-click an item in the **Project Explorer** window and select **Find»Callers** from the shortcut menu to display this dialog box. Use the **Find Callers** dialog box to find all callers of a specific item in the project.

Figure 2-3. Find Callers Dialog Box


Find Callers of "Boiler Data.ctl"

List of callers: (Found 2 callers of "Boiler Data.ctl")

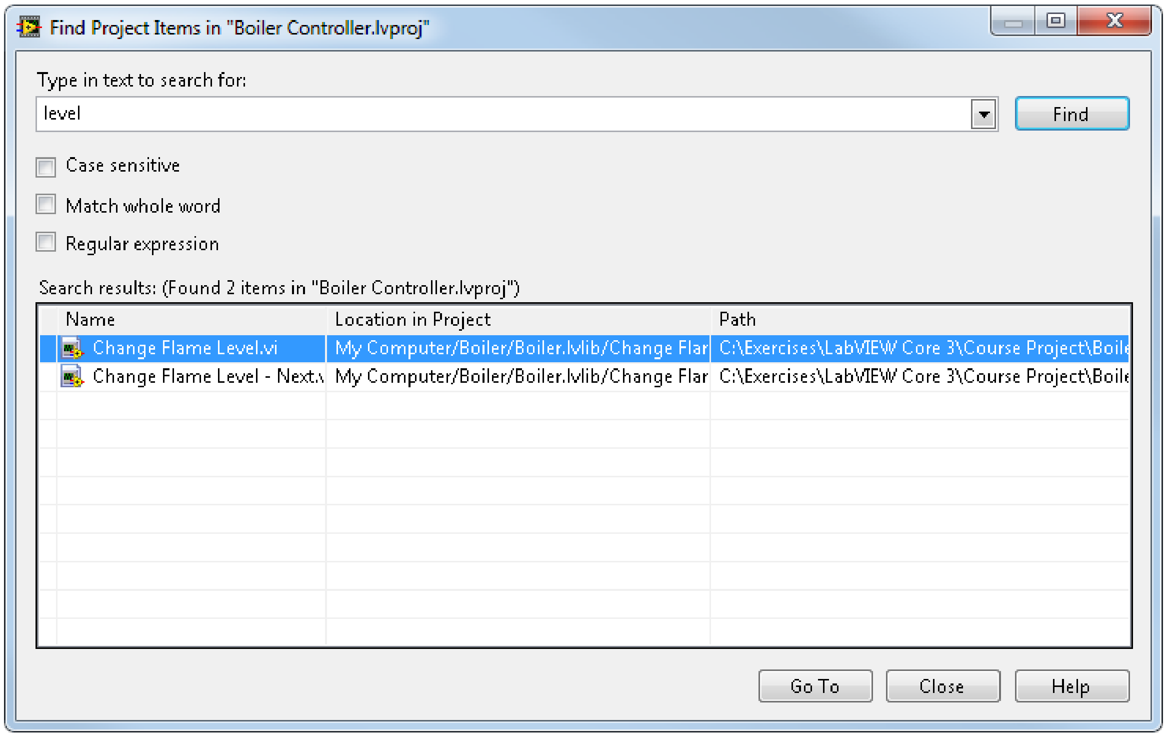
Name	Location in Project	Path
Boiler.vi	My Computer/Boiler/Boiler.lvlib/Boiler.vi	C:\perforce\CustomerEducation\Courses\InDev\L
Change Flame Level.vi	My Computer/Boiler/Boiler.lvlib/Change Flar	C:\perforce\CustomerEducation\Courses\InDev\L

This dialog box displays the VIs in which the selected item exists. It does not display the instances of the item within each VI. If the item has only one caller, LabVIEW highlights the caller in the **Project Explorer** window.

Find Project Items

Use the Find Project Items dialog box to find project items by the text in the names. To launch the Project Items dialog box from the Project Explorer window, select **Edit»Find Project Items**.

Figure 2-4. Find Project Items Dialog Box



Find Project Items in "Boiler Controller.lvproj"

Type in text to search for:

level

☐ Case sensitive

☐ Match whole word

☐ Regular expression

Search results: (Found 2 items in "Boiler Controller.lvproj")

Name	Location in Project	Path
Change Flame Level.vi	My Computer/Boiler/Boiler.lvlib/Change Flar	C:\Exercises\LabVIEW Core 3\Course Project\Boile
Change Flame Level - Next.v	My Computer/Boiler/Boiler.lvlib/Change Flar	C:\Exercises\LabVIEW Core 3\Course Project\Boile

This tool is useful when working in projects with a large number of project libraries and virtual folders. You can also use this dialog to check that your item names are sufficiently unique.

Show Item Paths

The **Project Explorer** window includes two pages. The **Items** page displays the contents of the project. The **Files** page displays the project items that have a corresponding file on disk. You can switch from one page to the other by right-clicking a folder or item under a target and selecting **Show in Items View** or **Show in Files View** from the shortcut menu.

You can view the path to each item in the Item view, as well. From the **Items** page in the **Project Explorer** window, select **Project»Show Item Paths**. You also can right-click the project root and select **View»Full Paths** from the shortcut menu to display the **Paths** column and view the file path.

Organizing the File Structure

Use the following guidelines to help you maintain a well-organized LabVIEW project. This section describes the following checklist in more detail.

☐ Organize the VIs in the file system to reflect the hierarchical nature of the software.

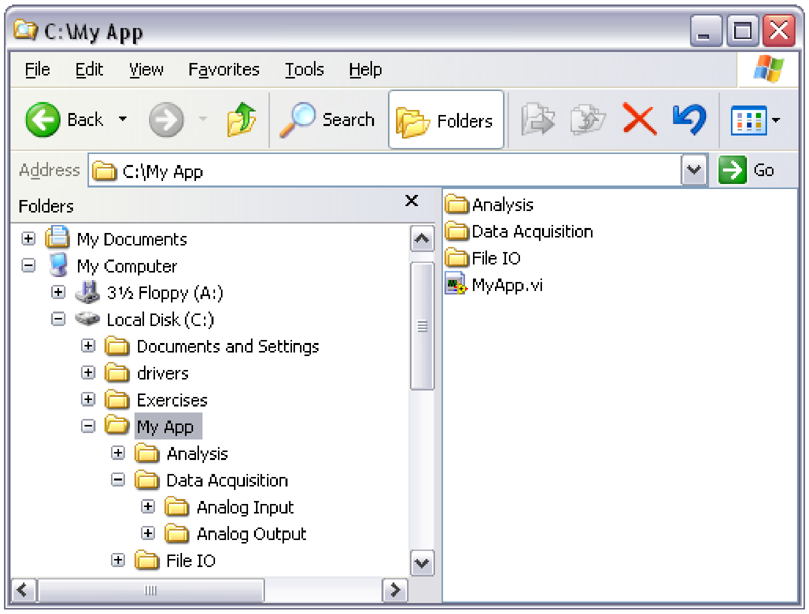
- ☐ Create a directory for all the VIs for one application and give it a meaningful name.
- ☐ Organize the VIs and subVIs modularly according to the functionality of the subVIs.
- ☐ Create a LabVIEW project hierarchy similar to the file organization on disk.
- ☐ Make top-level VIs directly accessible.
- ☐ Avoid using characters not all file systems accept when naming VIs, LLBs, and directories.
- ☐ Avoid creating files with the same name anywhere within the hierarchy.

  Only one VI of a given name can be in memory at a time. If you have a VI with a specific name in memory and you attempt to load another VI that references a subVI of the same name, the VI links to the VI in memory. If you make backup copies of files, be sure to save them into a directory outside the normal search hierarchy so that LabVIEW does not mistakenly load them into memory when you open development VIs.
- ☐ Avoid using absolute paths in VIs.

  Using absolute paths might cause problems when you build an application or run the VI on a different computer. If you must use an absolute path, ensure that you include code to test that the path exists and to create the path if it does not exist.

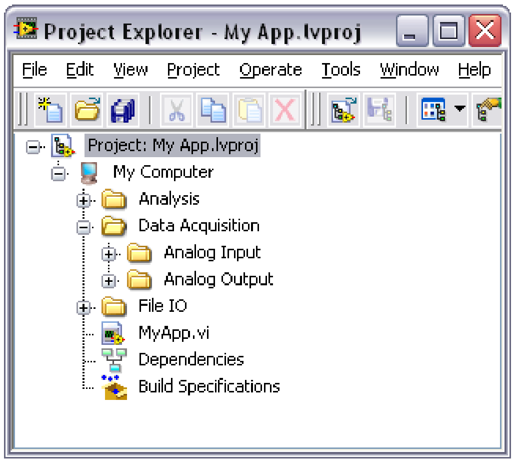
Figure 2-5 shows a folder, `MyApp`, containing a VI-based application. The main VI, `MyApp.vi`, resides in this folder along with the folders containing all the subVIs.

Figure 2-5. Directory Hierarchy



If you create a LabVIEW project in the **Project Explorer** window, the project hierarchy should be similar to the file organization of the files on the system. Figure 2-5 shows an example project hierarchy based on the files in Figure 2-6.

Figure 2-6. Project Hierarchy



Refer to the *Saving VIs* topic of the *LabVIEW Help* for more information about saving VIs individually and in VI libraries.

C. Project Conflicts

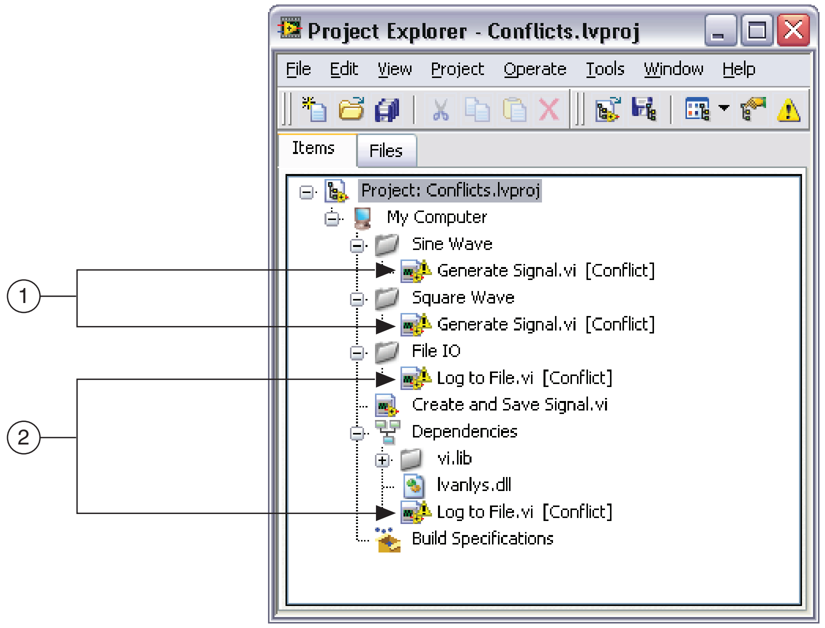
If you are not careful in naming and managing project items, you may accidentally create a cross-link or a conflict. A cross-link occurs when a VI calls a subVI from an unexpected location. Cross-linking occurs most commonly when you copy files from one location on disk to another, or if you have multiple copies of the same VI stored in different locations.

A conflict is a cross-link issue that occurs when LabVIEW tries to load a VI that has the same qualified name as an item already in the project. For example, if a VI calls an open subVI of the same qualified name as an item already in the project from a different path, a conflict occurs because of the cross-link. Most conflicts exist because other items in the project reference a conflicting item.

Note Cross-linking does not always result in a conflict.

Two common examples of conflicts are shown in Figure 2-7.

Figure 2-7. Examples of Conflicts in a LabVIEW Project


Project Explorer - Conflicts.lvproj
File Edit View Project Operate Tools Window Help
Items
Files
Project: Conflicts.lvproj
My Computer
Sine Wave
Generate Signal.vi [Conflict]
Square Wave
Generate Signal.vi [Conflict]
File IO
Log to File.vi [Conflict]
Create and Save Signal.vi
Dependencies
vi.lib
lvanlys.dll
Log to File.vi [Conflict]
Build Specifications
1
2

1 Two VIs in a project with the same name
2 A VI in the project is calling a subVI outside the project that has the same name as a VI in the project

Methods for Resolving Conflicts

Use the following tools to help you identify and resolve project conflicts and cross-linking errors.

- **Files View** in Project Explorer window
- **Show Item Paths** dialog box
- **Find Callers** command
- **Find Conflicts** command
- **Resolve Project Conflicts** dialog box

The best way to determine if cross-linking exists is to view the full path to the item. Right-click the project root and select **View»Full Paths** from the shortcut menu to display the **Paths** column and view the file paths that correspond to the project items. Refer to the *Show Item Paths* section for more information about viewing paths.

Right-click a specific conflicting item on the **Items** page and select **Find»Conflicts** to view all conflicting items in the **Find Conflicts** dialog box. If the item only conflicts with one other item, LabVIEW highlights the item in the **Project Explorer** window.

Note A yellow warning triangle appears on any conflicting items in the **Project Explorer** window.

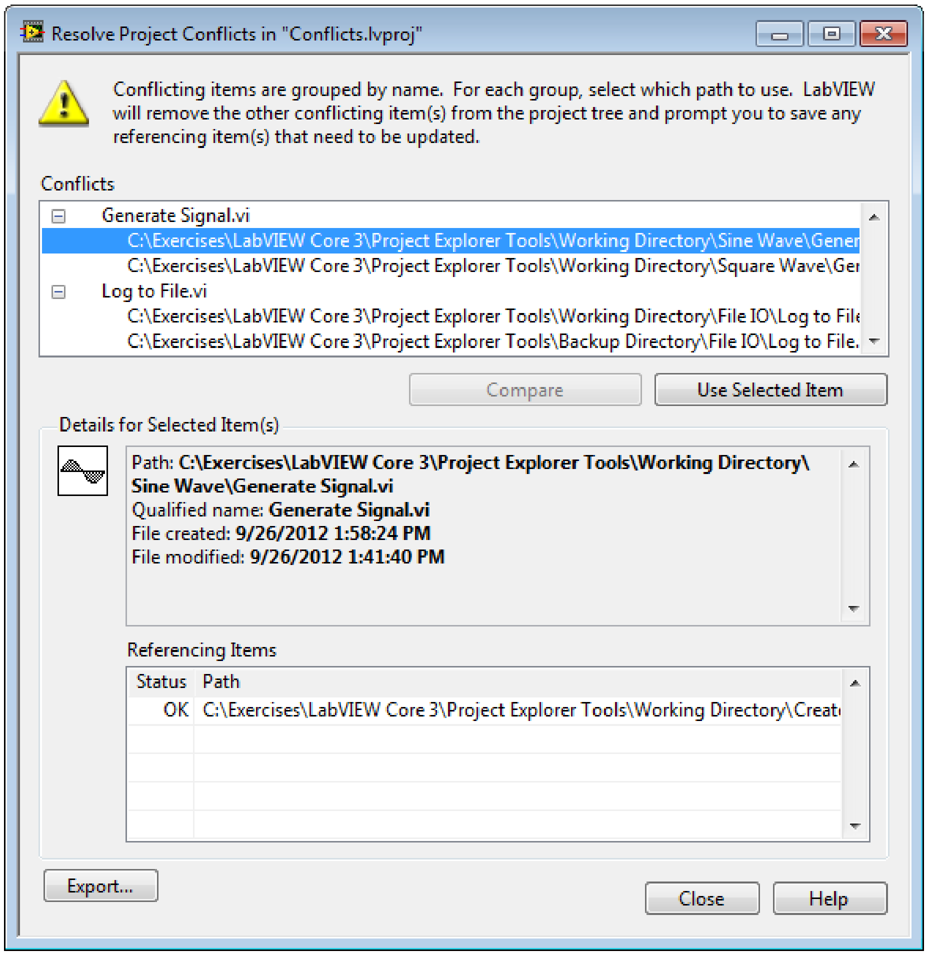
Resolve Project Conflicts Dialog Box

Use the **Resolve Project Conflicts** dialog box to identify and sometimes resolve project conflicts. To view the **Resolve Project Conflicts** dialog box, click the **Resolve Conflicts** button, shown below.

You also can select **Project»Resolve Conflicts** from the project menu to display this dialog box or right-click a conflicting item and select **Resolve Conflicts** from the shortcut menu.

Note LabVIEW disables the **Resolve Conflicts** menu and toolbar button unless the project contains conflicting items.

Figure 2-8. Resolve Project Conflicts Dialog Box



Resolve Project Conflicts in "Conflicts.lvproj"

Conflicting items are grouped by name. For each group, select which path to use. LabVIEW will remove the other conflicting item(s) from the project tree and prompt you to save any referencing item(s) that need to be updated.

Conflicts

Generate Signal.vi
C:\Exercises\LabVIEW Core 3\Project Explorer Tools\Working Directory\Sine Wave\Gener
C:\Exercises\LabVIEW Core 3\Project Explorer Tools\Working Directory\Square Wave\Ger
Log to File.vi
C:\Exercises\LabVIEW Core 3\Project Explorer Tools\Working Directory\File IO\Log to File
C:\Exercises\LabVIEW Core 3\Project Explorer Tools\Backup Directory\File IO\Log to File.

Details for Selected Item(s)

Path: **C:\Exercises\LabVIEW Core 3\Project Explorer Tools\Working Directory\Sine Wave\Generate Signal.vi**
Qualified name: **Generate Signal.vi**
File created: **9/26/2012 1:58:24 PM**
File modified: **9/26/2012 1:41:40 PM**

Referencing Items

Status	Path
OK	C:\Exercises\LabVIEW Core 3\Project Explorer Tools\Working Directory\Creat

For each conflict, select the path to the item in the **Conflicts** tree that you want to use and click **Use Selected Item**.

Note Under certain conditions, LabVIEW cannot use the item you select to resolve conflicts and disables the **Use Selected Item** button.

Refer to the *Resolve Project Conflicts Dialog Box* topic of the *LabVIEW Help* for more information about using this dialog to resolve project conflicts.

Methods for Manually Resolving Cross-Linking and Conflicts

In addition to the **Resolve Project Conflicts** dialog box described in the *Resolve Project Conflicts Dialog Box* section, you can use the following methods to manually resolve cross-links and conflicts.

Removing Conflicting Items

You can remove caller VIs from the project to resolve conflicts when a hierarchy of VIs conflicts with the hierarchy of other contents in the project. Right-click a VI or type definition and select **Find»Callers** or **Find»SubVIs** from the shortcut menu to highlight the caller or subVI the item references in the **Project Explorer** window. If more than one caller or subVI exists in the project, the **Find Callers** or **Find SubVIs** dialog box appears.

Right-click a project root or target and select **Find Items with No Callers** from the shortcut menu to display the **Find Items with No Callers** dialog box and find all top-level items. If no callers reference a conflicting subVI, remove the subVI from the project.

Note Removing a conflicting subVI from the project might not resolve the conflict because other VIs in the project may still reference the conflicting subVI. The item is a conflicting item until you remove all callers that call the conflicting item from the project. Deleting an item that has callers from the project moves the item to **Dependencies**.

Renaming Conflicting Items

If you do not want to remove the conflicting item and you detect it has the same qualified name as another item in the project, you can rename the item or add the item to a project library.

Renaming the item loads the callers that reference the incorrect item path, renames the item, and saves the item and all callers. The callers reference the new name. If you do not save the callers, the original item appears under **Dependencies** because callers still reference the item.

Tip Change the qualified name of a conflicting item by adding it to a LabVIEW project library.

When a VI is part of a project library, LabVIEW qualifies the VI name with the project library name to avoid cross-linking. A qualified name includes the filename and the qualified name of the owning project library filename. The qualified name changes without changing the path or filename.

For example, if you build a VI named `caller.vi` that includes a subVI named `init.vi` that `library1.lvlib` owns, you also can include a different subVI named `init.vi` that `library2.lvlib` owns and avoid cross-linking problems. The qualified names that LabVIEW records when you save `caller.vi` are `library1.lvlib:init.vi` and `library2.lvlib:init.vi`, respectively.

If a LabVIEW project library in memory conflicts with another project item, you must rename at least one conflicting library before loading. Right-click the library and select **Unload** from the shortcut menu. After LabVIEW unloads the library, you can reload the library and the VIs from the correct paths. If a library conflicts with another project item and is not in memory, you can right-click the library and select **Load** from the shortcut menu. After LabVIEW loads the library, you can edit the library or its contents and load VIs from the correct paths.

Redirecting Conflicting Items

When two or more items have the same qualified name, and only one item exists on disk, you can right-click a conflicting item and select **Replace with Item Found by Project** from the shortcut menu. LabVIEW updates the callers of the incorrect item to reference the item found on disk.

If you detect that one or more VIs incorrectly refers to the wrong subVI, redirect all callers to reference a subVI with a different path. Right-click a conflicting VI in the **Project Explorer** window and select **Replace with** from the shortcut menu to choose the correct subVI on disk. Select a replacement file from the file dialog that appears. LabVIEW automatically updates all items that reference the incorrect path to reference the replacement. You also can load each VI that refers to a conflicting item. The **Resolve Load Conflict** dialog box appears. You can choose a specific caller VI to load.

Note LabVIEW dims **Replace with Item Found by Project** and **Replace with** if the item is a project library or a member of a project library.

To practice the concepts in this section, complete Exercise 2-2.

Self-Review: Quiz

1. Which of the following is NOT a benefit of using project libraries to organize your project?
   a. Avoid potential VI name duplications
   b. Distribute a single library file instead of multiple VIs and controls
   c. Limit public access to certain files
   d. Organize related files into a shared hierarchy

2. Match each project explorer tool to its purpose:

Show Item Paths	a. Locates all items that use a specific item.
Find SubVIs	b. Locates all items that a specific item uses.
Find Callers	c. Locates all items whose names contain specific text
Find Project Items	d. Locates all items on disk

3. Which of the following is NOT true about file conflicts?
   a. Result whenever a VI is loaded from an unexpected path
   b. Can be avoided by using project libraries
   c. Can be avoided by using unique file names
   d. Result from LabVIEW trying to load two VIs with the same name

Self-Review: Quiz Answers

1. Which of the following is NOT a benefit of using project libraries to organize your project?
    a. Avoid potential VI name duplications
    b. Distribute a single library file instead of multiple VIs and controls
    c. Limit public access to certain files
    d. Organize related files into a shared hierarchy

On disk, a project library is a separate file from the files it contains. A project library stores a list of its member files and its member files link to the project library file. When distributing a project library, you must transfer each member file as well as the project library file.

2. Match each project explorer tool to its purpose:

Show Item Paths	**d. Locates all items on disk**
Find SubVIs	**b. Locates all items that a specific item uses**
Find Callers	**a. Locates all items that use a specific item**
Find Project Items	**c. Locates all items whose names contain specific text**

3. Which of the following is NOT true about file conflicts?
    a. Result whenever a VI is loaded from an unexpected path
    b. Can be avoided by using project libraries
    c. Can be avoided by using unique file names
    d. Result from LabVIEW trying to load two VIs with the same name

Cross-linking occurs when you load a VI from an unexpected path. However, cross-linking does not ALWAYS result in a file conflict. A file conflict occurs only when two VIs with the same name are loaded into memory.

Notes

3

Creating an Application Architecture

This lesson introduces techniques and programming practices for creating intuitive and robust architectures for large applications. You will learn the importance of testing your top-level architecture, the value of following established style guidelines, how to implement user events and notifiers, and how to use the queued message handler project template to begin development of a multi-loop application.

Topics

A. Architecture Testing
B. LabVIEW Style Guidelines
C. User Events
D. Queued Message Handler
E. Application Data Types
F. Notifiers

A. Architecture Testing

As you develop an application architecture, it is important that you test it to ensure that it satisfies the needs of the application. Use techniques described in this section to verify code and test the entire system for functionality, performance, and reliability.

Code Reviews

When you develop a VI, you can lose objectivity about the code you produce. The best way to determine that the code is correct is to perform code reviews.

To perform a code review, give one or more developers printouts of the VIs to review. You also can perform the review online because VIs are easier to read and navigate online. Talk through the design and compare the description to the actual implementation. Consider many of the same issues included in a design review. During a code review, ask and answer some of the following questions:

- What happens if a specific VI or function returns an error? Are errors dealt with and/or reported correctly?
- Are there any race conditions? A race condition occurs when two or more pieces of code that execute in parallel change the value of the same shared resource. Because the outcome of the VI depends on which action executes on the shared resource first, race conditions cause unpredictable outcomes. Race conditions often occur with the use of local and global variables or an external file, although race conditions can exist any time more than one action updates the value of the same stored data.
- Is the block diagram implemented well? Are the algorithms efficient in terms of speed and/or memory usage?
- Is the block diagram easy to maintain? Does the developer make good use of hierarchy, or is he placing too much functionality in a single VI? Does the developer adhere to established guidelines?

There are a number of other features you can look for in a code review. Take notes on the problems you encounter and add them to a list you can use as a guideline for other walk-throughs.

Focus on technical issues when doing a code review. Remember to review only the code, not the developer who produced it. Do not focus only on the negative; be sure to raise positive points as well. Several third-party resources contain more information about walk-through techniques.

Note Refer to the *Managing Software Engineering in LabVIEW* course for more information about how to conduct a code review.

System Tests

System tests evaluate aspects of the top-level application or architecture. System testing is necessary to ensure that the customer receives the VI they expect. System testing also helps you move the project toward a final sign off on the VI. System tests focus on the following areas:

- Configuration
- Performance
- Stress/Load
- Functionality
- Reliability

Note Every individual or company has its own method for testing a system. The guidelines described in this section are suggestions for system testing, not requirements. Do not change your existing testing strategy unless it is deficient.

Tip Refer to the *Managing Software Engineering in LabVIEW* course for information about tools that they can use to automate module and system testing.

Configuration Tests

Configuration tests evaluate the execution of the application on systems with different hardware and software configurations. Consider implementing the following configuration tests.

- Test on multiple operating systems unless the requirements document clearly specifies the operating system for the VI.
- Test the VI running on different screen resolutions. Make sure the VI can display the required information on the minimum screen resolution stated in the requirements document. Also, test the VI on higher screen resolutions to ensure that the user interface items remain proportional and are not distorted.
- Test with different hardware configurations.

Performance Tests

Performance tests define and verify performance benchmarks on the features that exist in the VI. The requirements document should indicate any performance requirements for the VI. Make sure you test each performance requirement in the requirements document.

Consider implementing performance tests to evaluate the following.

- Execution time
- Memory usage
- File sizes

The final implementation of the system must be able to meet the performance requirements. The system should respond to the user within the predefined limits specified in the requirements.

Performance testing also tests how well the software performs with external hardware and interfaces. Check all interfaces outside the system to make sure the software responds within set limits. You can use benchmarking in performance tests to evaluate the performance of the system.

Stress/Load Tests

Stress/load tests define ways to push a VI beyond the required limits in the specification. Stress/load testing helps guarantee that a VI performs as required when it is deployed.

Typical stress/load tests answer the following questions.

- How does the application handle large quantities of data?
- What happens when the application runs for an extended time?
- How well does the VI run concurrently with a large number of applications?

Functionality Tests

The final implementation of a system must be functional to the level stated in the requirements document. The easiest way to test for functionality is to refer to the original requirements document. Check that there is a functional implementation of each requirement listed in the requirements document. Functional testing also must involve the customer at some point to guarantee that the software functions at the level stated in the requirements. After you complete functional testing and the software passes the software test, you can verify the software as functional.

Reliability Tests

Reliability tests evaluate the execution of the application in a non-development environment. The goal of this testing is to identify and resolve issues that developers did not consider. This often involves some form of pilot testing.

Alpha Tests

Schedule alpha testing to begin when the software first reaches a stage where it can run as a system. Alpha testing typically involves a developer running the system on another computer.

The main focus of alpha testing is to execute system tests, fix errors, and produce beta-quality software. Beta-quality software is defined as software ready for external customer use.

System tests should verify behavior from a user-interaction perspective. Usually a developer does not test his or her own features. Execute system tests and any relevant regression tests from previous releases. You also may want to test existing features that have not been modified but that are affected by new functionality.

Alpha testing usually involves making the software available to other internal departments for testing, including departments whose products interact with or depend on the software.

Beta Tests

Beta testing begins when most of the system is operational and implemented and ready for user feedback. Beta testing involves sending a beta version of the software to a select list of customers for testing. Include a test plan that asks the users to test certain aspects of the software.

Write additional systems tests to execute during the beta phase. Include performance and usability tests. The tests executed during the alpha phase should be executed again as part of beta testing. Also include time to test the product by freely developing applications like users do. Consider that you might need to create more test cases or update existing tests.

Note Alpha and beta testing are the only testing mechanisms for some companies. However, alpha and beta testing are not a substitute for other forms of testing that rigorously test each component to verify that the component meets stated objectives. When this type of testing is done late in the development process, it is difficult and costly to incorporate changes suggested as a result.

B. LabVIEW Style Guidelines

Good software design techniques ensure that you create LabVIEW modules and VIs that are scalable, readable, and maintainable.

Practicing good LabVIEW style is one of the best ways to prevent bugs or errors in the code. Keeping the block diagram clean and easy to read can minimize the amount of debugging an application requires. Using good style might increase the time required to implement code, but you actually save time because using good style can reduce the time you spend debugging and testing a VI.

This section presents several recommended guidelines, but this is not a comprehensive list. Refer to the *LabVIEW Help* for additional recommendations.

Maintaining Appropriate Size of Block Diagram

The size of the block diagram window can affect how readable LabVIEW code is to others. Make the block diagram window no larger than the screen size, such as 1024 × 768 pixels. Code that is larger than the window is hard to read because it forces users to scroll through the window. If the code is too large to fit on one screen, make sure the user has to scroll only in one direction to view the rest of the code. If the block diagram requires scrolling, consider using subVIs.

Using Proper Wiring Techniques

Use the **Align Objects** and **Distribute Objects** pull-down menus on the toolbar to arrange objects symmetrically on the block diagram. When objects are aligned and distributed evenly, you can use straight wires to wire the objects together. Using straight wires makes the block diagram easier to read.

The following good wiring tips also help keep the block diagram clean:

- Avoid placing any wires under block diagram objects because LabVIEW can hide some segments of the resulting wire. Draw wires so that you can clearly see if a wire correctly connects to a terminal. Delete any extraneous wires. Do not wire through structures if the data in the wire is not used in the structure.
- Add as few bends in the wires as possible and keep the wires short. Avoid creating wires with long complicated paths because long wires are confusing to follow. If you must create long wires, space parallel wires evenly in straight lines and around corners.
- Avoid using local variables when you can use a wire to transfer data. Every local variable that reads the data makes a copy of the data. Use global and local variables as sparingly as possible.
- Make sure data flows from left to right and wires enter from the left and exit to the right.

  LabVIEW uses a left-to-right layout so block diagrams need to follow this convention. Although the positions of program elements do not determine execution order, avoid wiring from right to left. Only wires and structures determine execution order.

Note You can use the Clean Up Diagram tool to address many of these issues. Refer to the *Automatically Cleaning Up the Block Diagram* topic of the *LabVIEW Help* for more information about using this tool.

Block Diagram Documentation

Create good block diagram documentation to explain algorithms and increase code readability. This makes it easier for other developers to use and modify your code.

Block Diagram Labels

Use the following suggestions for documenting the block diagram.

- Use comments on the block diagram to explain what the code is doing.

  The free label located on the **Decorations** palette has a colored background that works well for block diagram comments. This free label is the standard for comments. Remember that comments in the block diagram are more likely to be read than the VI, so it is important to use correct spelling and grammar.
- Use wire labels on 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.
- 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.

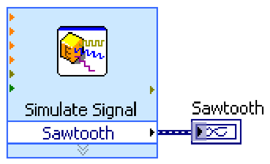
Tip Double-click an open space to add a free label to a VI.

Hiding Functionality

LabVIEW provides several high-level programming tools, such as Express VIs, Call Library Function Nodes, and Data Socket connections, that hide the functionality that the tool performs. These tools provide built-in functionality but require you to provide more documentation to explain how the application uses the tools.

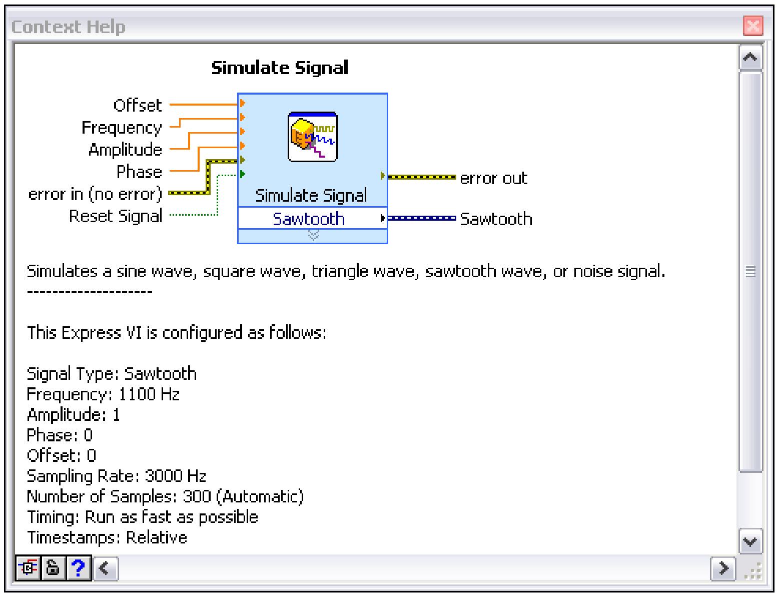
Express VIs allow you to build common measurement tasks without having to write or debug code. But, to understand how an Express VI is configured, you must open its configuration page. For example, the Simulate Signal Express VI shown in Figure 3-1 generates a Sawtooth wave with a frequency of 1.1 kHz. However, there is no information on the block diagram that indicates how the Sawtooth wave is configured.

Figure 3-1. Configured Simulate Signal Express VI



You can dramatically improve the readability of the Simulate Signal Express VI by documenting its configuration. If you idle the mouse over an Express VI when the **Context Help** window is open, the current configuration displays in the **Context Help** window, as shown in Figure 3-2.

Figure 3-2. Context Help for Simulate Signal Express VI



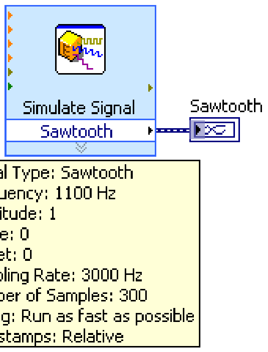
Simulates a sine wave, square wave, triangle wave, sawtooth wave, or noise signal.

This Express VI is configured as follows:

Signal Type: Sawtooth
Frequency: 1100 Hz
Amplitude: 1
Phase: 0
Offset: 0
Sampling Rate: 3000 Hz
Number of Samples: 300 (Automatic)
Timing: Run as fast as possible
Timestamps: Relative

You can use the information that is located in the **Context Help** window to create a free label comment on the block diagram that documents the configuration of Express VIs, as shown in Figure 3-3.

Figure 3-3. Fully Documented Express VI



Signal Type: Sawtooth
Frequency: 1100 Hz
Amplitude: 1
Phase: 0
Offset: 0
Sampling Rate: 3000 Hz
Number of Samples: 300
Timing: Run as fast as possible
Timestamps: Relative

Use labels on Call Library Function Nodes to specify what function the node is calling and the path to the library the node calls.

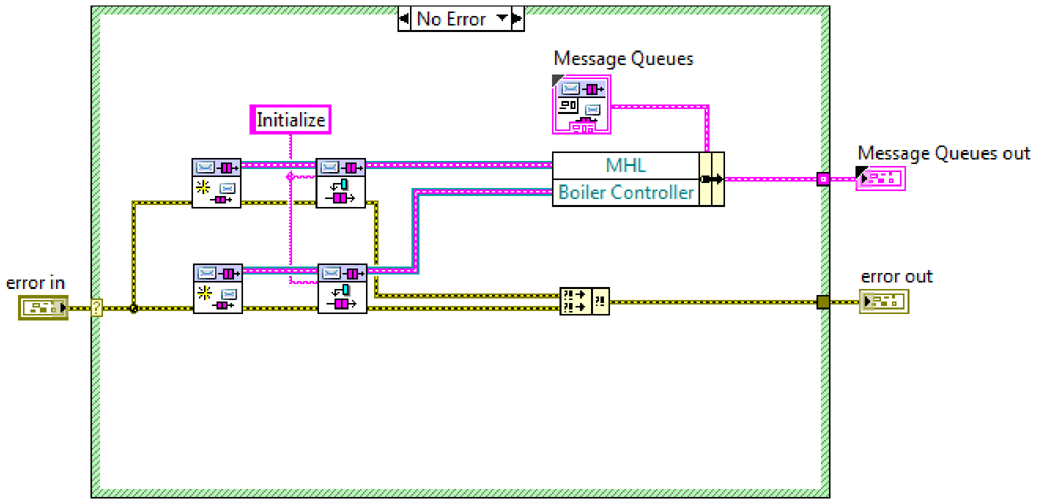
Use a free label with a Data Socket connection to specify the address that information is being broadcast to.

Developing Self-Documenting Code

Some LabVIEW code is self-documenting. In other words, you can easily understand the purpose of the code by simple inspection. When you use self-documenting code you may not need to provide additional free labels to describe the functionality.

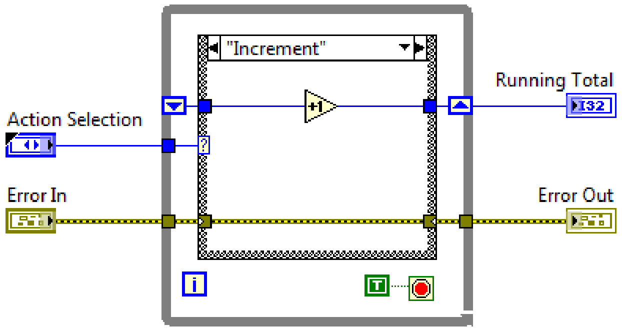
The Bundle by Name and Unbundle By Name functions are examples of self-documenting code. It is easy to see the data these functions use, as shown in Figure 3-4.

Figure 3-4. Self-Documentation with the Unbundle by Name and Bundle by Name Functions



Another example of self-documenting code is an enumerated type control wired to a Case structure. The items listed in the enumerated type control populate the case selector of the Case structure, as shown in Figure 3-5. This helps document the function of each case.

Figure 3-5. Self-Documentation with the Enumerated Type Control wired to a Case Structure Selector



C. User Events

You can programmatically create and name your own events, called user events, to carry user-defined data. Like queues and notifiers, user events allow different parts of an application to communicate asynchronously. You can handle both user interface and programmatically generated user events in the same Event structure.

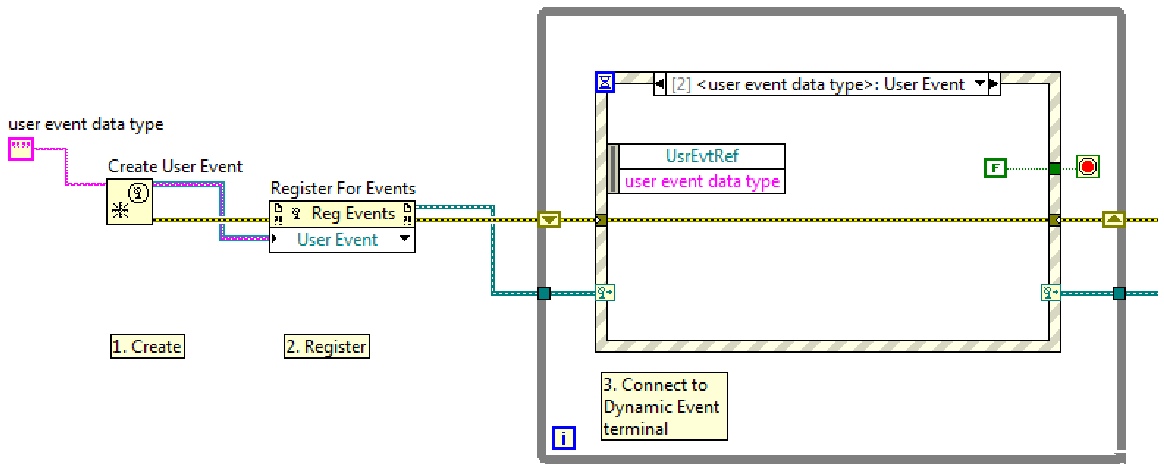
Creating and Registering User Events

To define a user event, wire a block diagram object, such as a front panel terminal or block diagram constant, to the Create User Event function. The data type of the object defines the data type of the user event. The label of the object becomes the name of the user event. If the data type is a cluster, the name and type of each field of the cluster define the data the user event carries. If the data type is not a cluster, the user event carries a single value of that type, and the label of the object becomes the name of the user event and of the single data element.

The **user event out** output of the Create User Event function is a strictly typed refnum that carries the name and data type of the user event. Wire the **user event out** output of the Create User Event function to an **event source** input of the Register For Events function.

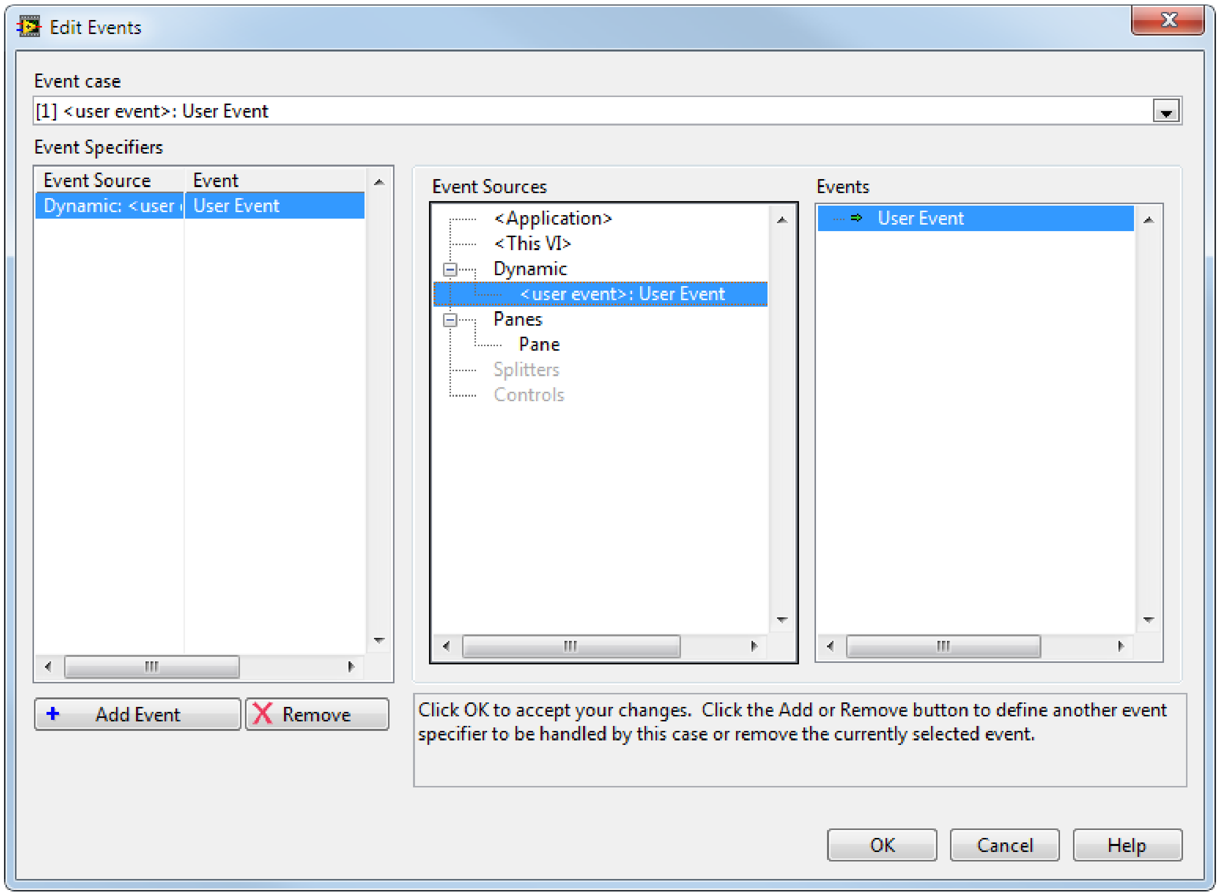
You cannot register for a user event statically. Handle a user event the same way you handle a dynamically registered user interface event. Wire the **event registration refnum** output of the Register For Events function to the dynamic event terminal on the left side of the Event structure.

Figure 3-6. Create and Register a User Event



Use the **Edit Events** dialog box, as shown in Figure 3-7, to configure a case in the Event structure to handle the event. Right-click the Event structure and select **Edit Events Handled by This Case** from the shortcut menu to display this dialog box. You also can right-click the Event structure and select **Add Event Case** or **Duplicate Event Case** from the shortcut menu to display this dialog box.

Figure 3-7. Edit Events Dialog Box



Click OK to accept your changes. Click the Add or Remove button to define another event specifier to be handled by this case or remove the currently selected event.

The name of the user event appears under the **Dynamic** subheading in the **Event Sources** section of the dialog box.

The user event data items appear in the Event Data Node on the left border of the Event structure. User events are notify events and can share the same event case of an Event structure as user interface events or other user events.

You can wire a combination of user events and user interface events to the Register For Events function.

Generating User Events

Use the Generate User Event function to deliver the user event and associated data to other parts of an application through an Event structure configured to handle the event. The Generate User Event function accepts a user event refnum and a value for the event data. The data value must match the data type of the user event.

If the user event is not registered, the Generate User Event function has no effect. If the user event is registered but no Event structure is waiting on it, LabVIEW queues the user event and data until an Event structure executes to handle the event. You can register for the same user event multiple times by using separate Register For Event functions, in which case each queue associated with an event registration refnum receives its own copy of the user event and associated event data each time the Generate User Event function executes.

Unregistering and Destroying User Events

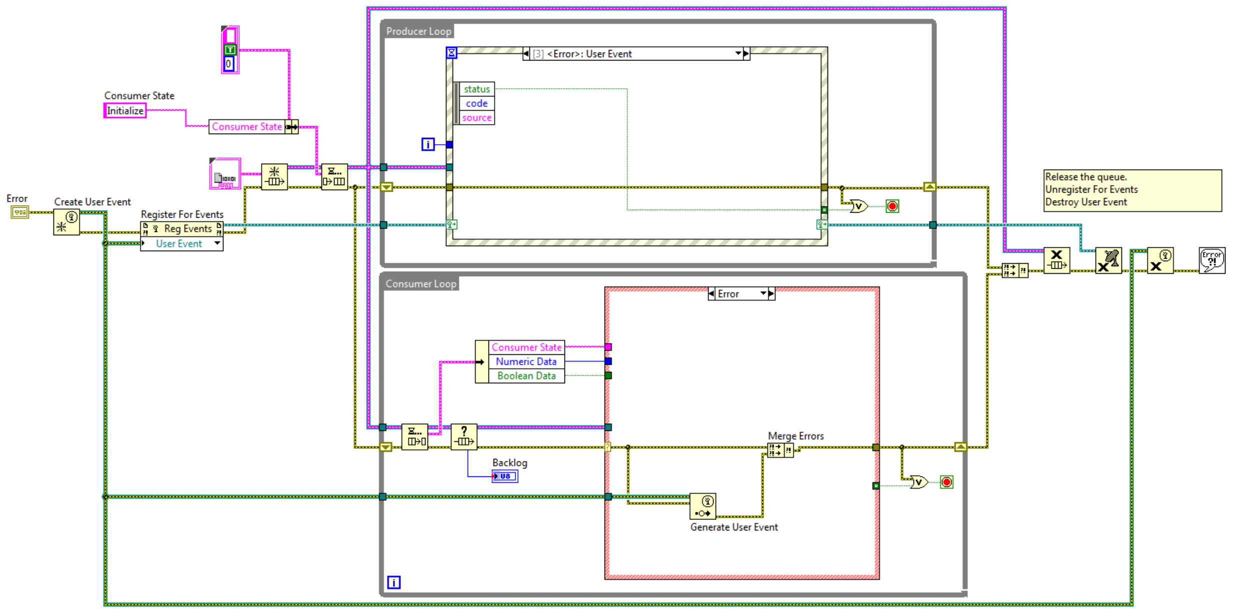
Unregister user events with the Unregister For Events function when you no longer need the user event. In addition, destroy the reference to the user event by wiring the user event refnum to the **user event** input of the Destroy User Event function. Wire the **error out** output of the Unregister For Events function to the **error in** input of the Destroy User Event function to ensure that the functions execute in the correct order.

LabVIEW unregisters all events and destroys existing user events automatically when the top-level VI finishes running. However, National Instruments recommends that you unregister and destroy user events explicitly, especially in a long-running application, to conserve memory resources.

Producer/Consumer Pattern with User Events

Figure 3-8 shows a modified producer/consumer (events) design pattern in which a user event sends a message from the consumer loop to the producer loop. Notice that the data type of the event is the error cluster that is wired to the Create User Event function. With the addition of user events, the producer/consumer (events) design pattern becomes a flexible design pattern that you can use for many types of applications.

Figure 3-8. Producer/Consumer (Events) with User Events



D. Queued Message Handler

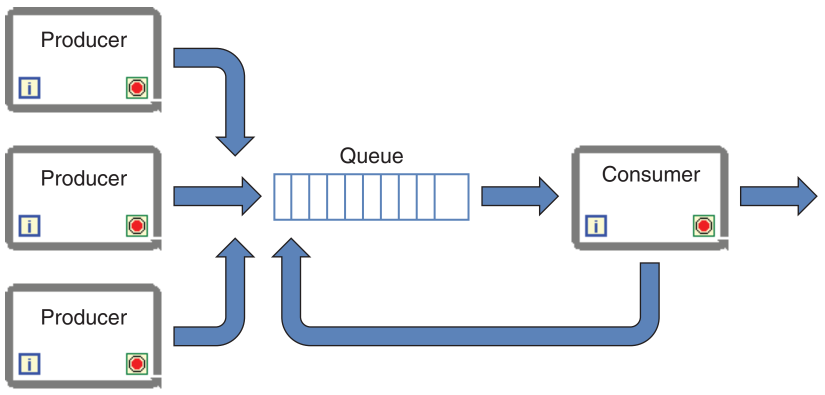
The Queued Message Handler (QMH) is a version of the producer/consumer design pattern where the producer sends messages to the consumer, which can also generate messages. A message is an instruction and the data needed to execute that instruction.

The QMH is useful when you have multiple parallel and independent processes that need to communicate with each other.

Characteristics of the QMH include the following.

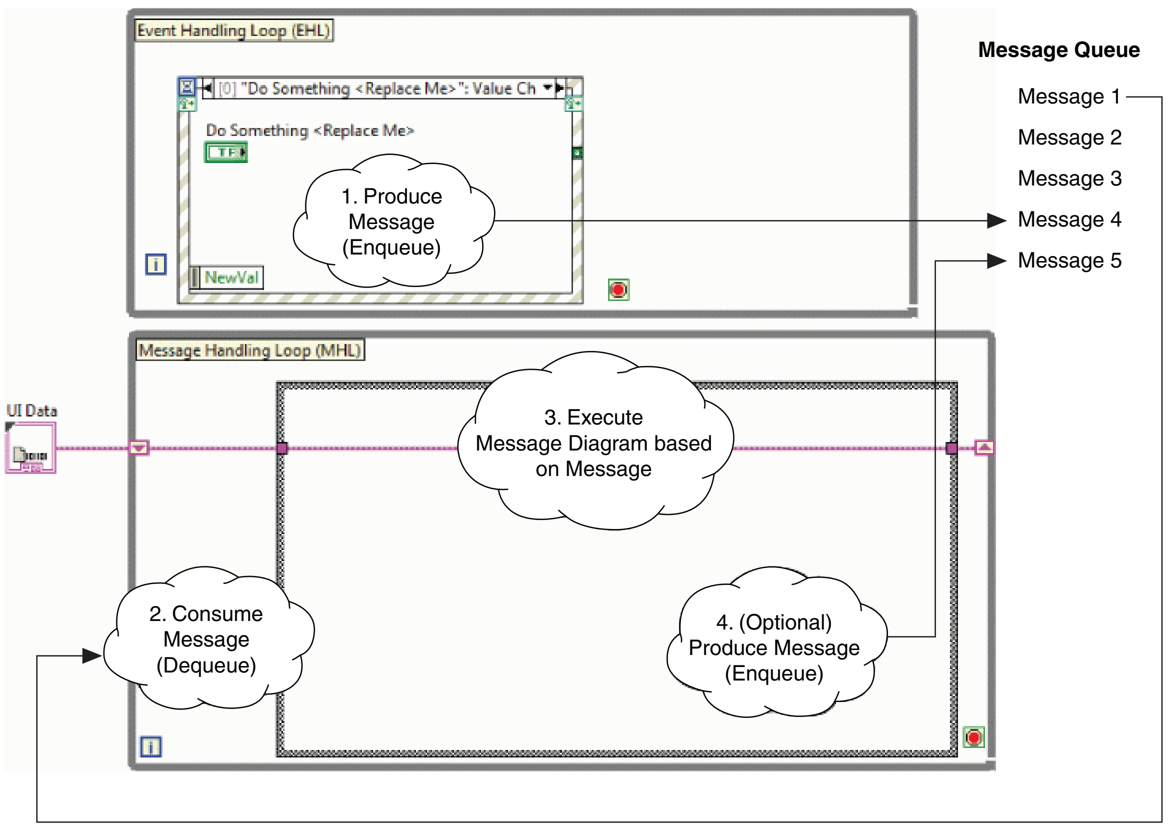
- QMH use queues as mechanism to pass data between independent loops.
- Queues allow loops to run at independent rates so there is no data dependency.
- Each queue can have one consumer, but many producers, including the consumer, as shown in Figure 3-9.
- Applications often have more than one queue, which requires a look-up mechanism to identify which queue is appropriate to enqueue message.

Figure 3-9. A QMH Queue



The Queued Message Handler consists of an event handling loop and at least one message handling loop, as shown in Figure 3-10.

Figure 3-10. Queued Message Handler

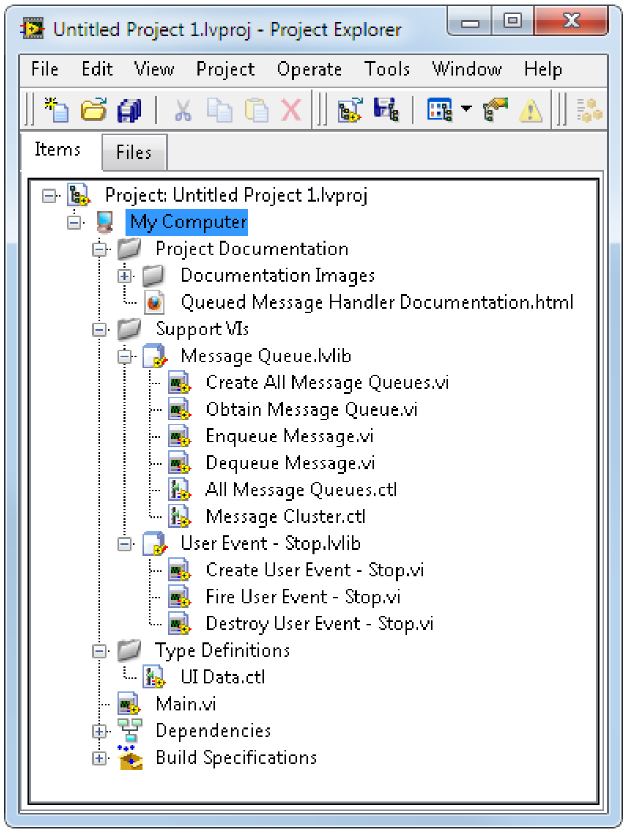

Event Handling Loop (EHL)
[0] "Do Something <Replace Me>": Value Ch
Do Something <Replace Me>
NewVal
1. Produce Message (Enqueue)
Message Queue
Message 1
Message 2
Message 3
Message 4
Message 5
Message Handling Loop (MHL)
UI Data
3. Execute Message Diagram based on Message
2. Consume Message (Dequeue)
4. (Optional) Produce Message (Enqueue)

QMH Project Template

LabVIEW provides a project template using a QMH. Complete the following steps to create a project using the QMH project template.

1. Click the **Create Project** button on the LabVIEW **Getting Started** window to open the **Create Project** dialog box.
2. Select **Queued Message Handler** from the **Choose a starting point for the project** page and click the **Next** button.
3. Enter information about the project, such as filename and root directory, and click the **Finish** button.
4. LabVIEW creates the project and opens it in the **Project Explorer** window as shown in Figure 3-11.

Figure 3-11. The QMH Project Template


Untitled Project 1.lvproj - Project Explorer
File Edit View Project Operate Tools Window Help
Items
Files

```
Project: Untitled Project 1.lvproj
  My Computer
    Project Documentation
      Documentation Images
      Queued Message Handler Documentation.html
    Support VIs
      Message Queue.lvlib
        Create All Message Queues.vi
        Obtain Message Queue.vi
        Enqueue Message.vi
        Dequeue Message.vi
        All Message Queues.ctl
        Message Cluster.ctl
      User Event - Stop.lvlib
        Create User Event - Stop.vi
        Fire User Event - Stop.vi
        Destroy User Event - Stop.vi
    Type Definitions
      UI Data.ctl
    Main.vi
    Dependencies
    Build Specifications
```

The QMH template generates the following code components:

- `Main.vi`—The top-level VI for the template
  - Event Handling Loop—Generates messages based on user-interface actions
  - Message Handling Loop—Processes messages from the event-handling loop or by other messages
- `Message Queue.lvlib`—Project library containing all message-related VIs and controls of the QMH
- `User Event - Stop.lvlib`—Project library containing all VIs and controls related to the Stop user event

Refer to `ni.com/info` and enter the info code `LVqueuedmsg` for information about working with and editing the QMH project template.

To practice the concepts in this section, complete Exercise 3-1.

QMH Sample Project

LabVIEW ships with a sample project that is built from the QMH project template. You can generate this example, Continuous Measurement and Logging, from the Create Project dialog box.

This project acquires measurement data continuously while logging to disk. The sample project includes the following five parallel loops.

- Event handling loop
- UI message loop
- Acquisition message loop
- Logging message loop
- Data display loop

E. Application Data Types

You often need to pass more than one type of data within a VI. Using a message cluster containing variants, strings, and enums gives you the flexibility to pass different types of commands and data to a consumer loop or message handling loop.

Application data types are often type-defined clusters that contain strings and variants. Strings and variants are two data types that LabVIEW can easily adapt to any other data type using the functions shown in Table 3-1.

Table 3-1. Adaptable LabVIEW Data Types

Data Type			
String		Scan from String	Scans the input string and converts the string according to input you provide.
		Format into String	Formats string, path, enumerated type, time stamp, Boolean, or numeric data as text.
Variant		Variant to Data	Converts variant data to a LabVIEW data type so LabVIEW can display or process the data.
		To Variant	Converts any LabVIEW data to variant data.

Variants

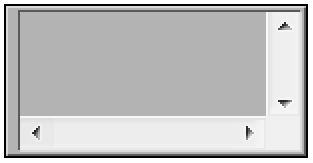
Sometimes you may need a VI to handle data of many different types in a generic way. For example, you may have a VI that performs the same functions on different data types. Instead of writing a different VI for each specific data type, you can use variants in the VI. The variant data type is a generic container for all other types of data in LabVIEW.

Creating Variants

Complete the following steps to create a variant.

1. On the front panel window, navigate to the **Modern»Variant & Class** palette or use the **Quick Drop** function and search for `variant`.
2. Select **Variant** and place the object on the front panel window. By default, LabVIEW places a variant indicator on the front panel window, as shown in Figure 3-12

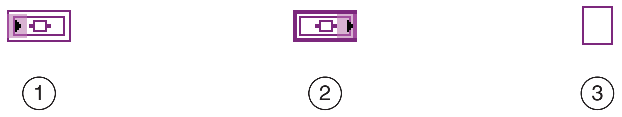
Figure 3-12. Variant Indicator on a Front Panel Window



3. (Optional) To create a control variant, right-click the indicator and select **Change to Control** from the shortcut menu.
4. (Optional) To create a variant constant, complete the following steps.
   a. Right-click the variant and select **Find Terminal** from the shortcut menu.
   b. Right-click the variant terminal and select **Change to Constant** from the shortcut menu.

Figure 3-13 shows how variant data types appear on the block diagram.

Figure 3-13. Block Diagram Terminals for Variants



1 Variant indicator   2 Variant control   3 Variant constant

Converting Variants

When you convert other data to a variant, the variant stores the data and the original data type of the data, which allows LabVIEW to correctly convert the variant data back to the original data at a later time. For example, if you convert string data to a variant, the variant stores the text of the string and an indicator that says that this data was originally a string (as opposed to a path or an array of bytes, or other possible LabVIEW types).

Use the Variant functions, such as the To Variant function and the Variant to Data function, to create and manipulate variant data. You can convert any LabVIEW data type to the variant data type to use variant data in other VIs and functions. Several polymorphic functions return the variant data type.

Use the variant data type when it is important to manipulate data independently of data type, such as when you transmit or store data; read and/or write to unknown devices; or perform operations on a heterogeneous set of controls.

Note Variants are supported on LabVIEW Real-Time, but are incompatible with RT FIFOs. Use strings or LabVIEW classes instead.

Tip In many cases, you can use LabVIEW classes to replace variants. Using LabVIEW classes instead of variants can reduce functional overhead.

Message Clusters

One way to pass data between parallel loops or processes is to use a type-defined message cluster. A message cluster often contains two components, as described in Table 3-2.

Table 3-2. Common Components of a Message Cluster

Component	Purpose	Data Type
Message	Drives the Case structure of the process that receives the message.	String or enum
Message data	Is the data needed to execute the message. Not all messages require data.	String or variant

The Message Component

The message component determines which case the Case structure implements. The message is usually a string or enum data type. Table 3-3 lists the benefits of using string or enum data types for the message component.

Table 3-3. Benefits of Using Different Data Types as the Message Component

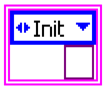
Strings	Enums
Can be created programmatically.	Limit possible message values and prevent typographical errors.
Make specifying messages easy.	Can be type-defined.
Are easier to localize.	Make adding case values easy for every case of a Case Structure.

The Message Data Component

The data type of the message data component should be flexible enough to handle a variety of data types. Because the variant data type is so flexible, the message data component is often a variant so that you can use the message cluster across multiple processes and applications.

Message Cluster Example—Enum-Variant

This example uses a cluster with an enumerated type control and variant, shown below.



The message is an enum data type that enables the producer loop to control what function the consumer loop performs. The message data is a variant data type that enables you to pass any data type from the producer to the consumer loop.

F. Notifiers

Notifiers allow different parts of an application to communicate asynchronously. Notifiers, like queues and user events, are a key tool in event-driven programming.

Like queues, notifiers suspend execution of the block diagram until they receive a message from another section of code. However, unlike queues, notifiers do not buffer multiple messages. If no nodes are waiting on a message when it is sent, the data is lost if another message is sent. Notifiers behave like single-element, bounded, lossy queues. In addition, notifiers allow one message to be read by multiple processes.

Table 3-4 summarizes key differences between queues and notifiers.

Table 3-4. Differences Between Notifiers and Queues

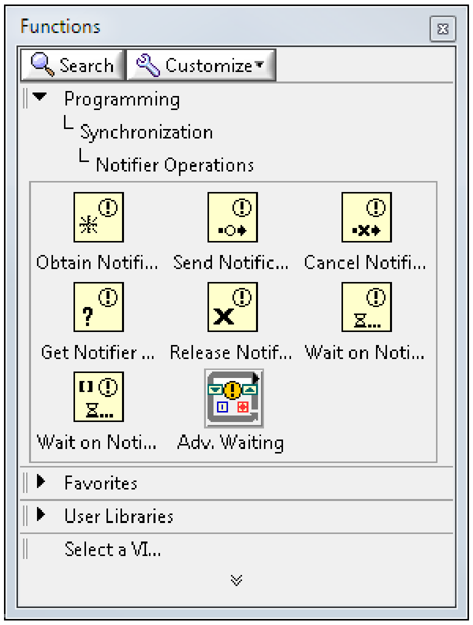
	Suspend execution?	**Buffer multiple messages?**	**Messages can be read by multiple processes?**
Notifiers	Yes	No	Yes
Queues	Yes	Yes	No

Tip Refer to the LabVIEW Core 2 course or the LabVIEW Help for more information about queues and event-driven programming.

Notifier Operations Palette

You can locate the Notifier Operations functions on the **Synchronization VIs and Functions» Notifier Operations** palette, as show in Figure 3-14.

Figure 3-14. The Notifier Operations Palette



Refer to the *Notifier Operations Functions* topic of the *LabVIEW Help* for more information about each function in this palette.

Using Notifiers

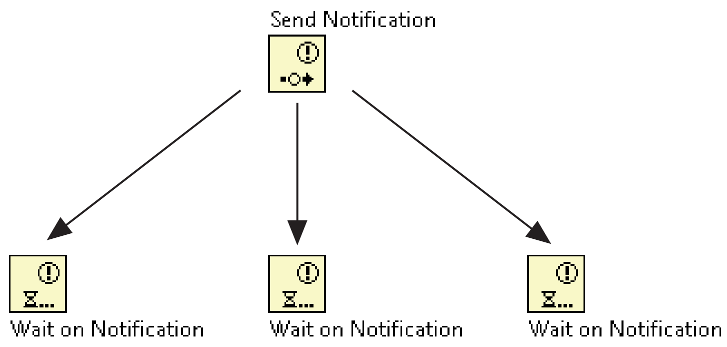
Applications use notifiers in two common designs:

- One writer, multiple readers
- Handshaking

One Writer, Multiple Readers

Figure 3-15 illustrates how the one writer, multiple readers design works.

Figure 3-15. Sending One Notification to Trigger Multiple Waiting Processes



Notifiers work differently than queues in the above scenario. For example, if ten Wait on Notification functions are wired to the same notifier reference, all ten Wait on Notification functions execute when the notification is sent. On the other hand, if ten Dequeue Element functions were wired to the same queue reference, only ONE Dequeue function executes when the message is enqueued.

Handshaking

The handshaking design pattern uses a message source and a message handler. The message source sends a message to the message handler and then waits for the message handler to return a notification that it has finished executing the message. The message source waits for notification that the message has been processed before continuing.

Note As long as you do not need to buffer messages, you can use a notifier for handshaking. If there is a possibility of missing a notification, use queues or a different design pattern.

Table 3-5. Roles of the Message Source and Message Handler

Message Source	Message Handler
1. Sends a message to a message handler. 2. Waits for notification that handler has finished executing.	1. Receives message. 2. Processes message. 3. Sends notification that execution is complete.

To practice the concepts in this section, complete Exercise 3-2.

Self-Review: Quiz

1. Match each type of testing to its description:

Configuration testing	a. Pilot test (alpha and beta) the application in non-development environments.
Performance testing	b. Test the application execution time, memory usage, and file size.
Stress/Load testing	c. Test the application on systems with different hardware and software.
Reliability testing	d. Test the application's behavior beyond the required limits in the specifications.

2. What is the main difference between a consumer loop in a queued message handler design and a producer/consumer design?
   a. The queued message handler uses queues.
   b. The queued message handler uses user events.
   c. The consumer loop of the queued message handler can send messages.
   d. The queued message handler can have more than one consumer loop.

3. Which of the following are differences between a notifier and a queue? (multiple answers)
   a. A notifier can contain only one element at a time.
   b. A notifier allows you to pass complex data types between loops.
   c. A notifier can trigger multiple loops that are waiting for an element.
   d. A notifier allows you to communicate asynchronously between loops.

4. Which data types can adapt to hold any type of data?
   a. Enum
   b. String
   c. Variant
   d. Numeric

Self-Review: Quiz Answers

1. Match each type of testing to its description:

Configuration testing	**c. Test the application on systems with different hardware and software.**
Performance testing	**b. Test the application execution time, memory usage, and file size.**
Stress/Load testing	**d. Test the application's behavior beyond the required limits in the specifications.**
Reliability testing	**a. Pilot test (alpha and beta) the application in non-development environments.**

2. What is the main difference between a consumer loop in a queued message handler design and a producer/consumer design?
   a. The queued message handler uses queues.
   b. The queued message handler uses user events.
   c. The consumer loop of the queued message handler can send messages.
   d. The queued message handler can have more than one consumer loop.

3. Which of the following are differences between a notifier and a queue? (multiple answers)
   a. A notifier can contain only one element at a time.
   b. A notifier allows you to pass complex data types between loops.
   c. A notifier can trigger multiple loops that are waiting for an element.
   d. A notifier allows you to communicate asynchronously between loops.

4. Which data types can adapt to hold any type of data?
   a. Enum
   b. String
   c. Variant
   d. Numeric

Notes

4

Customizing the User Interface

This lesson introduces techniques to improve the way you implement front panels in LabVIEW. You will learn how to use runtime menus, splitter bars, panes, subpanels, and tab controls to customize and extend your user interface. You will also learn the value of creating a user interface prototype and techniques for improving the usability of your application.

Topics

A. User Interface Style Guidelines

B. User Interface Prototypes

C. Customizing a User Interface

D. Extending a User Interface

E. Window Appearance

F. User Documentation

G. User Interface Initialization

H. User Interface Testing

A. User Interface Style Guidelines

If a VI serves as a user interface or dialog box, front panel appearance and layout are important. When designing a front panel for a user interface, choose fonts, colors, and graphics carefully to make the user interface more intuitive and easy to use.

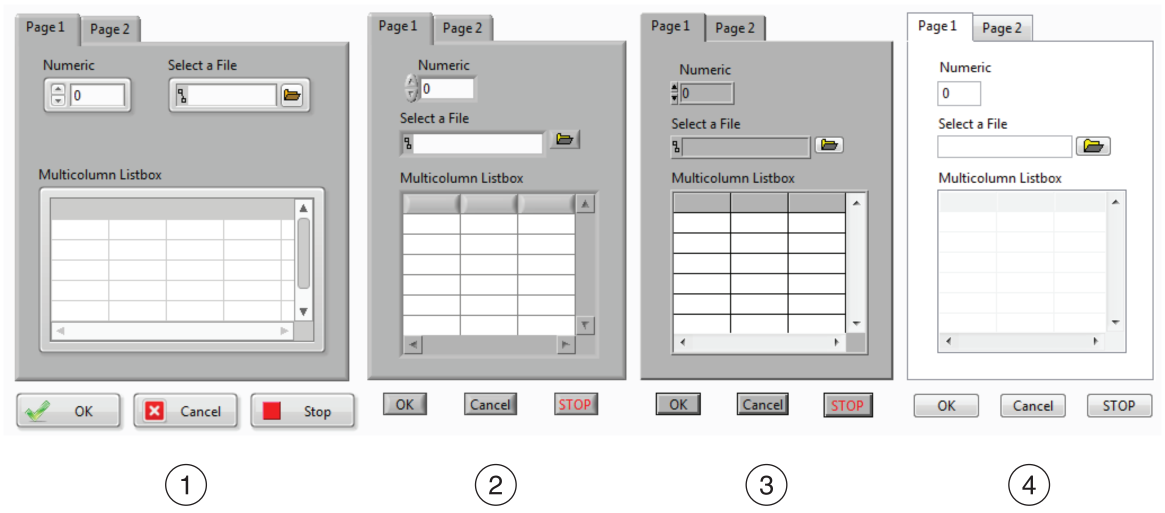
Front Panel Object Styles

You can choose controls and objects from different palettes when you build a user interface. Figure 4-1 shows four examples of the same user interface, each built using controls and objects from different palettes. Front panel controls and indicators can appear in modern, classic, system, or silver style.

Many front panel objects have a high-color appearance. Set the monitor to display at least 16-bit color for optimal appearance of the objects.

The controls and indicators located on the Silver and Modern palettes also have corresponding low-color objects. Use the controls and indicators located on the Classic palette to create VIs for 256-color and 16-color monitor settings.

Figure 4-1. User Interface Control Styles



1	User Interface Built Using Silver Controls	3	User Interface Built Using Classic Controls
2	User Interface Built Using Modern Controls	4	User Interface Build Using System Controls

Use **Silver** controls for customer-facing front panels. These controls have a more modern user-friendly appearance than the other styles.

Use **Modern** controls for most subVI front panels.

Use **Classic** controls for backward compatibility with other code or as a starting point for creating your own custom controls. You also can use Classic controls for VIs that require an efficient user interface.

Use **System** controls and indicators in dialog boxes you create. These controls differ from those that appear on the front panel only in terms of appearance. These controls appear in the colors you have set up for the system.

The system controls change appearance depending on which platform you run the VI. 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.

Fonts and Text Characteristics

Limit the VI to the three standard fonts (application, system, and dialog) unless you have a specific reason to use a different font. For example, monospace fonts, which are fonts that are proportionally spaced, are useful for string controls and indicators where the number of characters is critical. Refer to the *Text Characteristics* topic of the *LabVIEW Help* for more information about setting the default font, using custom fonts, and changing the font style.

The actual font used for the three standard fonts (application, system, and dialog) varies depending on the platform. For example, when working on Windows, preferences and video driver settings affect the size of the fonts. Text might appear larger or smaller on different systems, depending on these factors. To compensate for this, allow extra space for larger fonts and enable the **Size to Text** option on the shortcut menu.

Allow extra space between controls to prevent labels from overlapping objects because of font changes on multiple platforms.

For example, if a label is to the left of an object, justify the label to the right and leave some space to the left of the text. If you center a label over or under an object, center the text of that label as well. Fonts are the least portable aspect of the front panel so always test them on all target platforms.

Colors

Color can distract the user from important information. For instance, a yellow, green, or bright orange background makes it difficult to see a red danger light. Another problem is that some platforms do not have as many colors available. Use a minimal number of colors, emphasizing black, white, and gray. The following are some simple guidelines for using color:

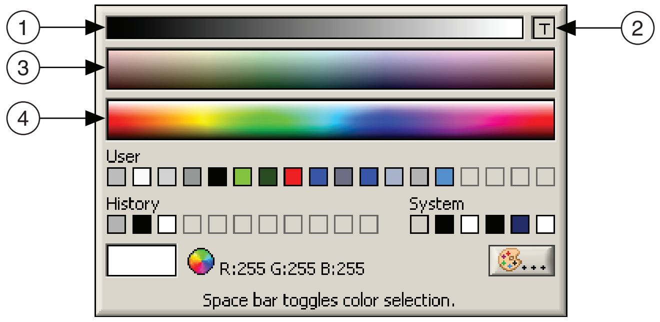
- Never use color as the sole indicator of device state. People with some degree of color-blindness can have problems detecting the change. Also, multiplot graphs and charts can lose meaning when displayed in black and white. Use lines for plot styles in addition to color.
- Consider coloring the pane backgrounds on the front panel background and objects of user interface VIs with the system colors, or symbolic colors, in the color picker. System colors adapt the appearance of the front panel to the system colors of any computer that runs the VI.
- Use light gray, white, or pastel colors for backgrounds. The first row of colors in the color picker contains less harsh colors suitable for front panel backgrounds and normal controls. The second row of colors in the color picker contains brighter colors you can use to highlight

important controls. Select bright, highlighting colors only when the item is important, such as an error notification.

- Always check the VI for consistency on other platforms.

  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 to highlights. Figure 4-2 shows the color picker in LabVIEW.

Figure 4-2. Color Picker



1	Grayscale Spectrum	3	Muted Color Spectrum
2	Transparency Box	4	Highlight Color Spectrum

Keep these things in mind when designing a front panel. For most objects, use complimentary neutral colors that vary primarily in their brightness. Use highlight colors sparingly for objects such as plots, meter needles, company logo, and so on.

Graphics

Use imported graphics to enhance the front panel. You can import graphics and text objects to use as front panel backgrounds, items in picture rings, and parts of custom controls and indicators.

Check how the imported graphics look when you load the VI on another platform. For example, a Macintosh PICT file that has an irregular shape might convert to a rectangular bitmap with a white background on Windows or Linux.

One disadvantage of using imported graphics is that they slow down screen updates. Make sure you do not place indicators and controls on top of a graphic object so that LabVIEW does not have to redraw the object each time the indicator updates. If you must use a large background picture with controls on top of it, divide the picture into several smaller objects and import them separately because large graphics usually take longer to draw than small ones.

User Interface Layouts

Consider the arrangement of controls on front panels. Keep front panels simple to avoid confusing the user. For example, you can use menus to help reduce clutter. For top-level VIs that users see, place the most important controls in the most prominent positions. For subVI front panels, place the controls and indicators of the subVI so that they correspond to the connector pane pattern.

Keep inputs on the left and outputs on the right whenever possible to minimize confusion on the part of the user. Use the **Align Objects**, **Distribute Objects**, and **Reorder Objects** pull-down menus to create a uniform layout.

User Interface Layout Examples

Without proper planning, it is easy to produce a user interface that does not incorporate good front panel design. Figure 4-3 shows an example of a poorly designed user interface.

Figure 4-3. Poorly Designed User Interface

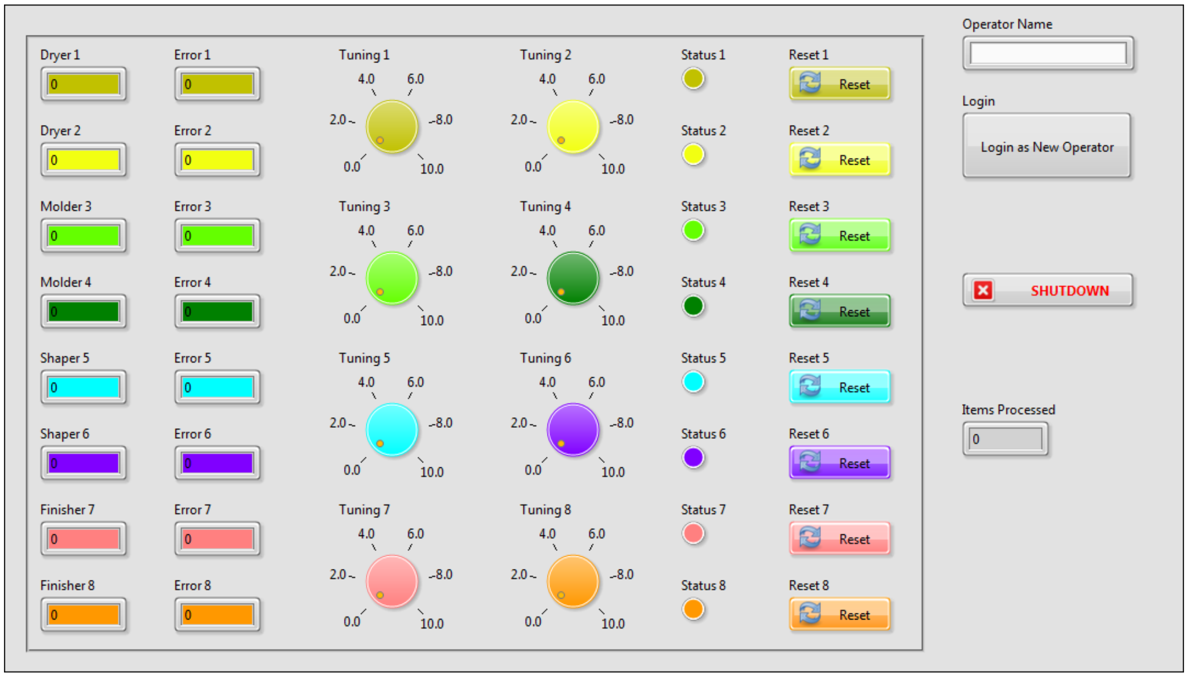
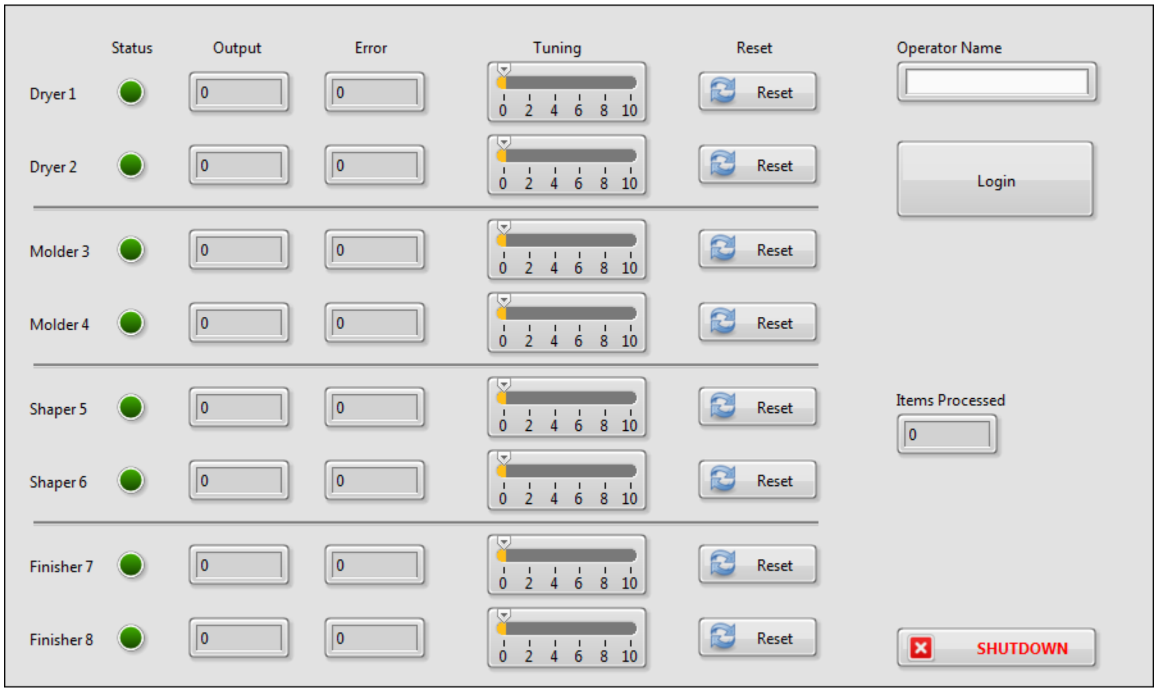


Figure 4-4 shows the same user interface with several improvements, such as reduced use of color, regrouped controls, fewer labels, sliders instead of knobs, and rearranged objects.

Figure 4-4. Improved User Interface



Size and Positioning

Front panels need to fit on a monitor that is the standard resolution for most intended users. Make the window as small as possible without crowding controls or sacrificing a clear layout. If the VIs are for in-house use and everyone is using high-resolution display settings, you can design large front panels. If you are doing commercial development, keep in mind that some displays have a limited resolution, especially LCD displays and touchscreens.

Launch the front panel in a consistent location, either at the upper-left corner or the center of the screen.

Front panels should open in the upper-left corner of the screen for the convenience of users with small screens. Place sets of VIs that are often opened together so the user can see at least a small part of each. Place front panels that open automatically in the center of the screen. Centering the front panels makes the VI easier to read for users on monitors of various sizes. Use the **VI Properties** dialog box to customize the window appearance and size.

From the front panel or block diagram window of a VI, select **File»VI Properties** and select **Window Run-Time Position** from the **Category** pull-down menu to open the **Window Run-Time Position** page. Use this page to customize the run-time front panel window position and size.

You also can use the `Front Panel:Run-Time Position` method to customize the run-time front panel window position and size programmatically.

Icon and Connector Panes

Use good style techniques when you create the icons and connector panes for VIs. Following icon and connector pane style techniques can help users understand the purpose of the VIs and make the VIs easier to use.

Icons

Create a meaningful icon for every VI. The icon represents the VI on a palette and a block diagram. When subVIs have well-designed icons, developers can gain a better understanding of the subVI without the need for excessive documentation.

Use the following suggestions when creating icons.

- The LabVIEW Icon Editor includes well-designed icons that you can use as prototypes. When you do not have a picture for an icon, text is acceptable. If you localize the application, make sure you also localize the text on the icon. A good size and font choice for text icons is 8 point Small Fonts in all caps.
- Create a unified icon style for related VIs to help users visually understand what subVIs are associated with the top-level VI.
- Always create standard size (32 × 32 pixels) icons. VIs with smaller icons can be awkward to select and wire and might look strange when wired.
- Do not use colloquialisms when making an icon because colloquialisms are difficult to translate. Users whose native language is not English might not understand a picture that does not translate well. For example, do not represent a datalogging VI with a picture of a tree branch or a lumberjack.

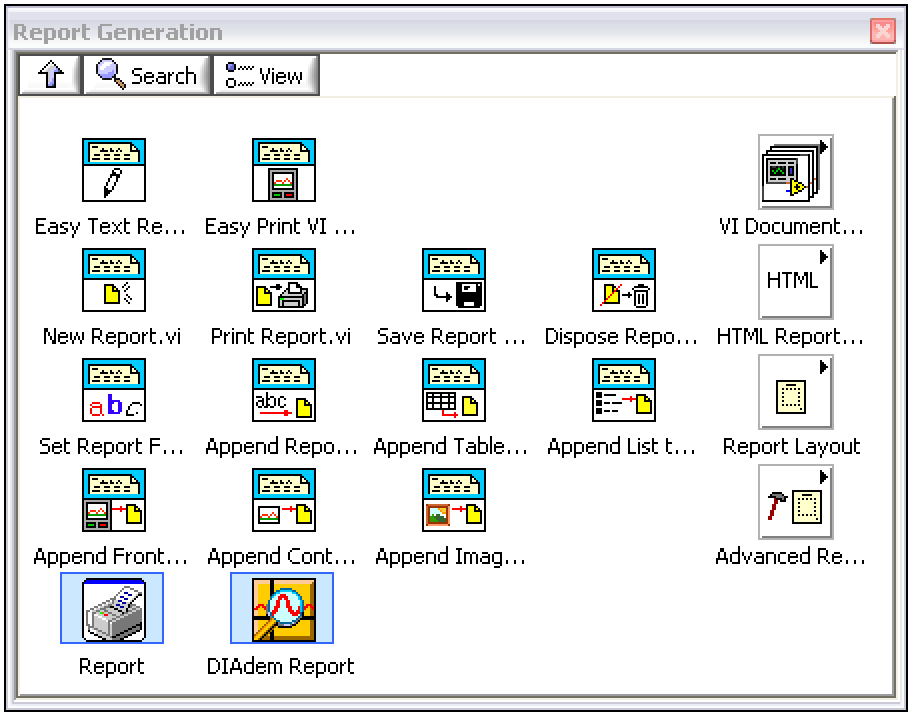
Refer to the *Creating Icons* topic of the *LabVIEW Help* for more information about creating icons.

Tip You can use the graphics available in the **Glyphs** tab of the Icon Editor to create icons.

Examples of Intuitive Icons

The Report Generation VIs are examples of icons designed with good style techniques.

Figure 4-5. Report Generation VIs



The Report Generation VIs use images of a disk, a printer, a pencil, and a trashcan to represent what the VIs do. The image of a piece of paper with text on it represents a report and is a common element in the icons. This consistency unifies the icon designs. Notice that none of the icons are language dependent, thus they are suitable for speakers of any language.

You also can create non-square icons that are similar to the built-in LabVIEW functions, such as the Numeric and Comparison functions. This style of icon can improve readability of the block diagram. For example, you can implement non-square icons to represent control theory functions, such as a summer. For more information about creating non-square icons, refer to the *Creating a Custom-Shaped Icon* topic of the *LabVIEW Help*.

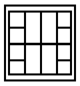
Implementing Appropriate Connector Panes

A connector pane is the set of terminals that correspond to the controls and indicators of a VI. Refer to the *Building the Connector Pane* topic of the *LabVIEW Help* for more information about setting up connector panes.

Use the following suggestions when creating connector panes:

- Keep the default 4 × 2 × 2 × 4 connector pane pattern to leave extra terminals for later development.

Figure 4-6. Example of $4 \times 2 \times 2 \times 4$ Pattern



Using the same pattern ensures that all VIs, even VIs with few inputs, line up correctly and have straight wires connecting them. Always select a connector pane pattern with more terminals than necessary; including extra terminals in the VI allows you to add additional connectors to the VI and makes relinking to the subVI in calling VIs unnecessary.

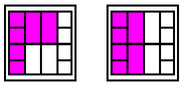
- Use a consistent connector pane layout across related VIs.

  Wire inputs on the left and outputs on the right to follow the standard left-to-right data flow.

  When assigning terminals, keep in mind how the VIs will be wired together. If you create a group of subVIs that you use together often, give the subVIs a consistent connector pane with common inputs in the same location to help you remember where to locate each input. If you create a subVI that produces an output another subVI uses as the input, such as references, task IDs, and error clusters, align the input and output connections to simplify the wiring patterns.

  When assigning terminals as inputs and outputs, make sure to split the terminals of the connector pane consistently. If you need to use the middle four terminals of the 4 × 2 × 2 × 4, divide them either horizontally or vertically. For example, assign the inputs to the top two terminals and the outputs to the bottom two terminals or assign the inputs to the left two terminals and the outputs to the right two terminals.

Figure 4-7. Example of the $4 \times 2 \times 2 \times 4$ Wiring Pattern with Terminals Assigned



- Avoid using connector panes with more than 16 terminals.

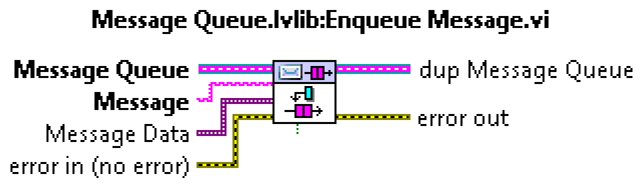
  Although connector pane patterns with more terminals might seem useful, they are very difficult to wire. If you need to pass more data, use clusters.

- The **Required**, **Recommended**, **Optional** setting for connector pane terminals affects the appearance of the inputs and outputs in the **Context Help** window, and prevents users from forgetting to wire subVI connections. Use the **Required** setting for inputs that users must wire for the subVI to run properly. Use the **Optional** setting for inputs that have default values that are appropriate for the subVI most of the time.

- Include **error in** and **error out** clusters in all subVIs, even if the subVI does not process errors. **error in** and **error out** clusters are helpful for controlling execution flow. If a subVI has an incoming error, you can use a Case structure to send the error through the VI without executing any of the subVI code.

Figure 4-8 shows the recommended style for assigning inputs and outputs to a connector pane, with the inputs on the left and the outputs on the right, following the flow of data from left to right.

Figure 4-8. Connector Pane Example



It is important to implement connector panes that provide for scalability and follow a standard for wiring VIs.

UI Localization

The Microsoft Developer Network defines localization as the process of creating an application that can be used in a different locale and culture[1]. Localization can be an important consideration when creating an application because many applications are used worldwide. Even though you might not immediately localize an application, consider the possibility that the application will be used in a different region or locale in the future. When an application is translated to another language, often a brute force translation technique is used to convert all the strings on the front panel to a different language. Use the following guidelines to make localization easier:

- Leave space for localization. Allow for at least 30% growth in short strings and 15% growth for long sentences.
- Do not hardcode user interface strings on the block diagram. Try to move string constants to string controls and hide them. Consider creating a read-only global variable for these strings.
- Avoid using non-international symbols and icons in a VI.
- Avoid using text in icons whenever possible so you do not need to localize the icons.
- Avoid using bitmaps in a VI so you do not need to localize any text in the bitmaps.
- Use a label to define the name of a control but always make the caption visible. Use the caption as the label for a control because you can change captions programmatically.

Refer to the *Localizing VIs* topic of the *LabVIEW Help* for more information about localizing VIs.

B. User Interface Prototypes

User interface prototypes provide insight into the organization of the program. Assuming the program is user-interface intensive, you can attempt to create a mock interface that represents what the user sees.

To create a prototype, create only the controls and indicators you think you need. Focus on the front panel and avoid implementing block diagrams in the early stages of creating prototypes. Leaving

[1]. Microsoft Corporation, *Localization Planning*, 2003, `http://msdn.microsoft.com`, MSDN.

the block diagram empty at first avoids the code-and-fix trap while you prototype. As you create buttons, listboxes, and rings, think about what needs to happen as the user makes selections. Ask yourself questions such as the following:

- Should the button lead to another front panel?
- Should some controls on the front panel be hidden or replaced by other controls?

If new options are presented, follow those ideas by creating new front panels to illustrate the results. This kind of prototyping helps to define the requirements for a project and gives you a better idea of its scope.

Systems with many user interface requirements are perfect for prototyping. Determining the method you use to display data or prompt the user for settings is difficult on paper. Instead, consider designing VI front panels with the controls and indicators you need. Leave the block diagram empty and figure out how the controls work and how various actions require other front panels. For more extensive prototypes, tie the front panels together. However, do not get carried away with this process.

If you are bidding on a project for a client, using front panel prototypes is an extremely effective way to discuss with the client how you can satisfy his or her requirements. Because you can add and remove controls quickly, especially if the block diagrams are empty, you help customers clarify requirements.

Limit the amount of time you spend prototyping before you begin. Time limits help to avoid overdoing the prototyping phase. As you incorporate changes, update the requirements and the current design.

To practice the concepts in this section, complete Exercise 4-1.

C. Customizing a User Interface

If a VI serves as a user interface or dialog box, front panel appearance and layout are important. A good user interface also organizes data in a VI, such as in an array or a cluster, to improve the readability of the VI as well as help reduce development time.

Run-Time Menus

To help reduce clutter, use menus. You can create custom menus for every VI you build, and you can configure VIs to show or hide menu bars.

Note Custom menus appear only while the VI runs.

You can build custom menus or modify the default LabVIEW menus statically when you edit the VI or programmatically when you run the VI.

Static Menus

To add a custom menu bar to a VI rather than the default menu bar, select **Edit»Run-Time Menu** and create a menu in the **Menu Editor** dialog box. LabVIEW creates a run-time menu (`.rtm`) file. After you create and save the `.rtm` file, you must maintain the same relative path between the VI and the `.rtm` file. You also can create a custom run-time shortcut menu by right-clicking a control and selecting **Advanced»Run-Time Shortcut Menu»Edit**. This option opens the **Shortcut Menu Editor**. When the VI runs, it loads the menu from the `.rtm` file.

Menu items can be the following three types:

- **User Item**—Allows you to enter new items that must be handled programmatically on the block diagram. A user item has a name, which is the string that appears on the menu, and a tag, which is a unique, case-sensitive string identifier. The tag identifies the user item on the block diagram. When you type a name, LabVIEW copies it to the tag. You can edit the tag to be different from the name. For a menu item to be valid, its tag must have a value. The **Item Tag** text box displays question marks for invalid menu items. LabVIEW ensures that the tag is unique to a menu hierarchy and appends numbers when necessary.
- **Separator**—Inserts a separation line on the menu. You cannot set any attributes for this item.
- **Application Item**—Allows you to select default menu items. To insert a menu item, select **Application Item** and follow the hierarchy to the items you want to add. Add individual items or entire submenus. LabVIEW handles application items automatically. These item tags do not appear in block diagrams. You cannot alter the name, tag, or other attributes of an application item. LabVIEW begins all of its application item tags with the prefix `APP_`.

Click the blue plus button, shown below, on the toolbar to add more items to the custom menu.

Click the red X button, shown below to delete items.

You can arrange the menu hierarchy in the **Menu Editor** dialog box by clicking the arrow buttons on the toolbar, using the hierarchy manipulation options in the **Edit** menu, or by dragging and dropping.

Menu Selection Handling

There are two possible approaches to handling run-time menu selections:

- Use menu functions
- Add an event handler case

Use Menu Functions

Use the **Menu** palette functions to modify the menus in LabVIEW applications. Use the functions located on the top row of the palette to handle menu selections.

When you create a custom menu, you assign each menu item a unique, case-insensitive string identifier called a tag. When the user selects a menu item, you retrieve its tag programmatically using the Get Menu Selection function. LabVIEW provides a handler on the block diagram for each menu item based on the tag value of each menu item. The handler is a While Loop and Case structure combination that allows you to determine which, if any, menu is selected and to execute the appropriate code.

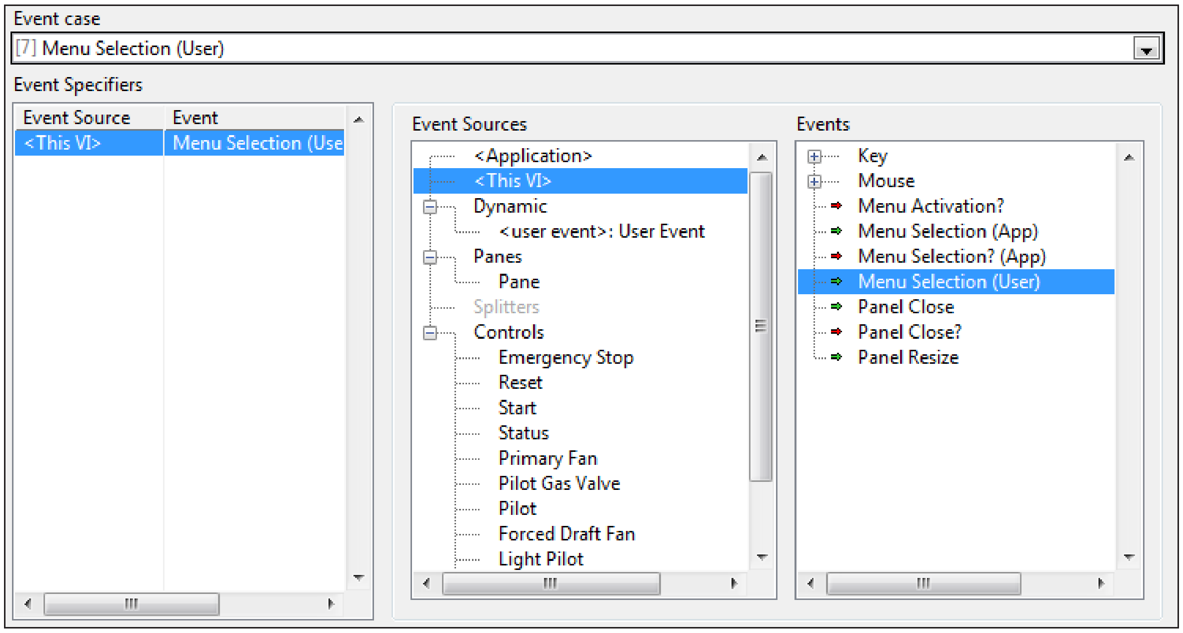
After you build a custom menu, build a Case structure on the block diagram that executes, or handles, each item in the custom menu. This process is called menu selection handling. LabVIEW handles all application items implicitly.

Use the Get Menu Selection and Enable Menu Tracking functions to define what actions to take when users select each menu item.

Add an Event Handler Case

To handle the selection in an Event structure, add an event case and edit the events that it handles, as shown in Figure 4-9.

Figure 4-9. Edit Events for a New Event Case


Event case
[7] Menu Selection (User)

Event Specifiers

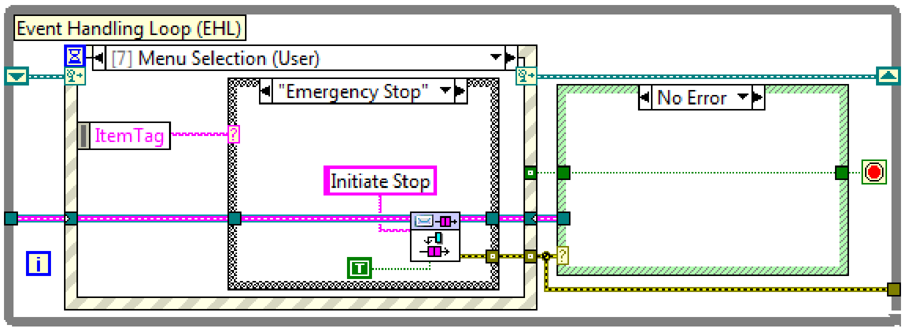
Event Source	Event
<This VI>	Menu Selection (Use

Event Sources
- <Application>
- <This VI>
- Dynamic
  - <user event>: User Event
- Panes
  - Pane
- Splitters
- Controls
  - Emergency Stop
  - Reset
  - Start
  - Status
  - Primary Fan
  - Pilot Gas Valve
  - Pilot
  - Forced Draft Fan
  - Light Pilot

Events
- Key
- Mouse
- Menu Activation?
- Menu Selection (App)
- Menu Selection? (App)
- Menu Selection (User)
- Panel Close
- Panel Close?
- Panel Resize

In the new event case, use the Item Tag for the selection to determine how the event is handled as shown in Figure 4-10.

Figure 4-10. Use Item Tag in New Event Case



Run-Time Shortcut Menus

You can customize the run-time shortcut menu for each control you include in a VI. To customize a shortcut menu, right-click a control and select **Advanced»Run-Time Shortcut Menu»Edit** from the shortcut menu to display the **Shortcut Menu Editor** dialog box. Use the **Shortcut Menu Editor** dialog box to associate the default shortcut menu or a customized shortcut menu file (`.rtm`) with the control. You can customize shortcut menus programmatically.

You also can add shortcut menus to front panels. To add a shortcut menu to the front panel, use the Shortcut Menu Activation and Shortcut Menu Selection pane events.

You also can disable the run-time shortcut menu on a control.

Note Custom run-time shortcut menus appear only while the VI runs.

Splitter Bars and Panes

You can use splitter bars such as toolbars or status bars to create professional user interfaces in the front panel window. You can create a splitter bar to separate the front panel into multiple regions, called panes. Each pane acts similarly to a unique front panel, with its own sets of pane coordinates and controls and indicators on it. You can scroll each pane individually. The splitter bar separates the controls on one pane from those on another pane, yet the terminals for all controls are on the same block diagram.

Refer to the *Creating Splitter Bars and Panes* topic of the *LabVIEW Help* for more information about using splitter bars and panes.

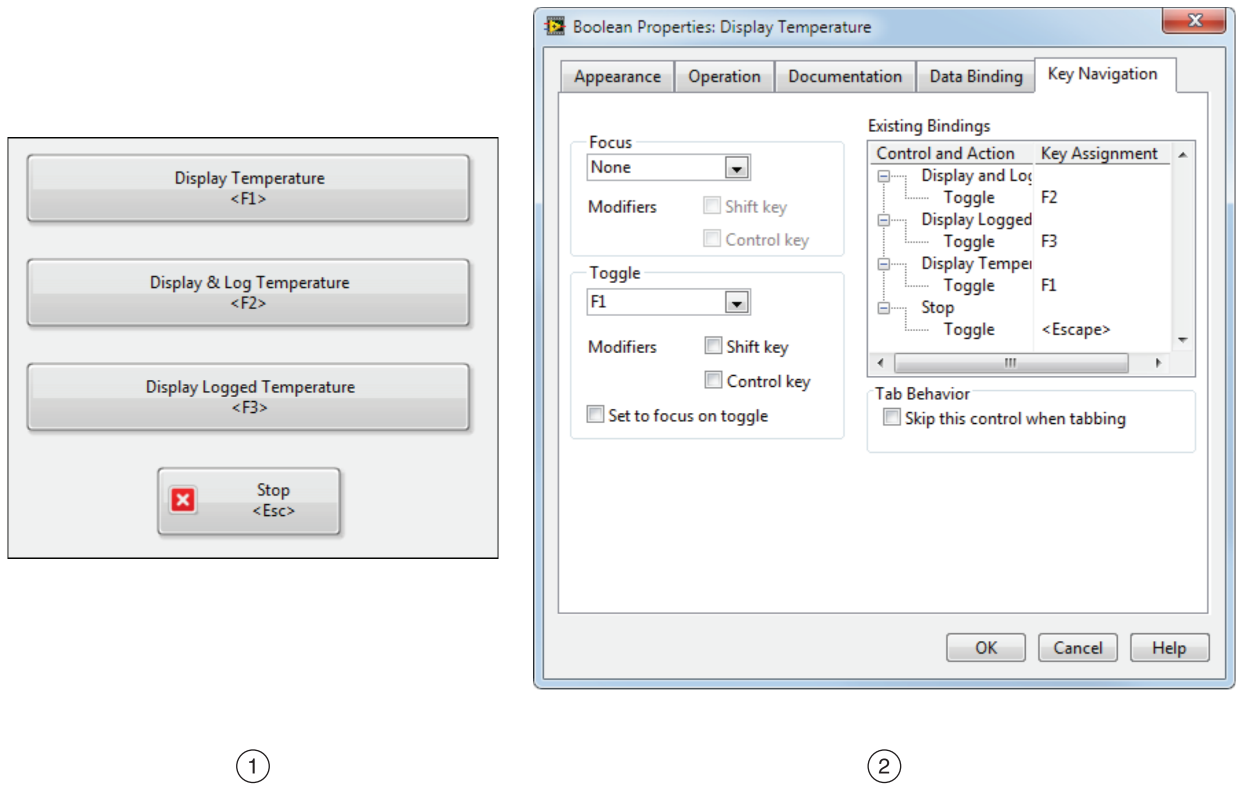
Key Navigation

Some users prefer to use the keyboard instead of a mouse. In some environments, such as a manufacturing plant, only a keyboard is available. Consider including keyboard shortcuts for VIs even if the use of a mouse is available because keyboard shortcuts add convenience to a VI.

Pay attention to the key navigation options for objects on the front panel and set the tabbing order for the objects to read left to right and top to bottom. Set the <Enter> key as the keyboard shortcut for the front panel default control, which is usually the **OK** button. However, if you have a multiline string control on the front panel, you might not want to use the <Enter> key as a shortcut.

If the front panel has a **Cancel** button, set the <Esc> key to be the keyboard shortcut. You also can use function keys as navigation buttons to move from screen to screen. If you do this, be sure to use the shortcuts consistently. Select **Edit»Set Tabbing Order** to arrange controls in a logical sequence when the user needs to tab between the controls. For controls that are off-screen, use the **Key Navigation** tab of the **Properties** dialog box, as shown in Figure 4-11, to skip over the controls when tabbing or to hide the controls.

Figure 4-11. Key Navigation



Focus: None
Modifiers: Shift key [ ], Control key [ ]
Toggle: F1
Modifiers: Shift key [ ], Control key [ ]
Set to focus on toggle [ ]

Existing Bindings

Control and Action	Key Assignment
Display and Log	
Toggle	F2
Display Logged	
Toggle	F3
Display Temper	
Toggle	F1
Stop	
Toggle	<Escape>

Tab Behavior: Skip this control when tabbing [ ]

1 Front panel buttons with corresponding key navigation indicated
2 Key Navigation tab of Properties dialog box

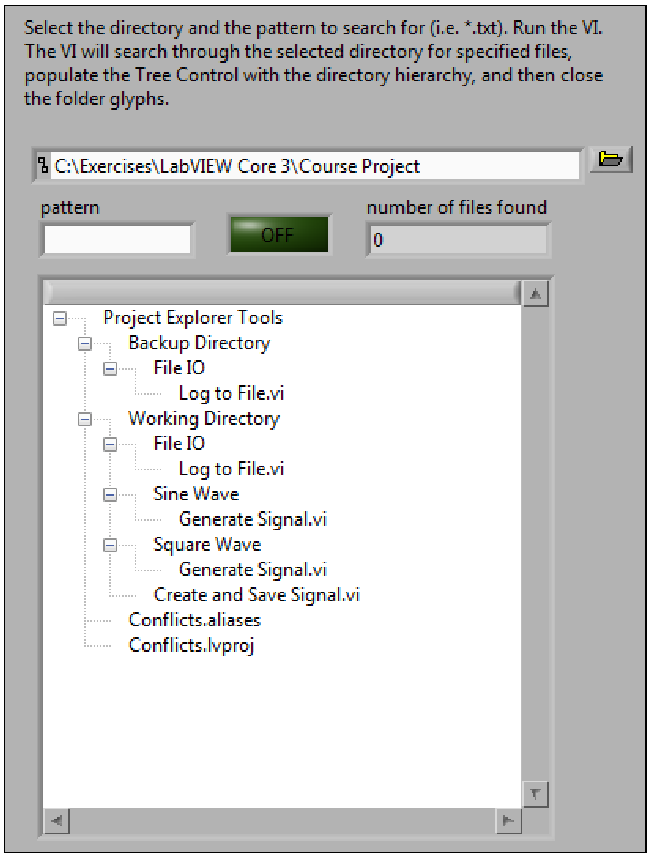
Also consider using the key focus property to set the focus programmatically to a specific control when the front panel opens.

Tree Controls

Use the tree control to give users a hierarchical list of items from which to select. You organize the items you enter in the tree control into groups of items, or nodes. Click the expand symbol next to a node to expand it and display all the items in that node. You also click the symbol next to the node to collapse the node.

Figure 4-12 shows the Directory Hierarchy in Tree Control VI from the *NI Example Finder*, which uses a tree control.

Figure 4-12. VI with Tree Control


Select the directory and the pattern to search for (i.e. *.txt). Run the VI. The VI will search through the selected directory for specified files, populate the Tree Control with the directory hierarchy, and then close the folder glyphs.

C:\Exercises\LabVIEW Core 3\Course Project

pattern

number of files found: 0

```
Project Explorer Tools
  Backup Directory
    File IO
      Log to File.vi
  Working Directory
    File IO
      Log to File.vi
    Sine Wave
      Generate Signal.vi
    Square Wave
      Generate Signal.vi
    Create and Save Signal.vi
  Conflicts.aliases
  Conflicts.lvproj
```

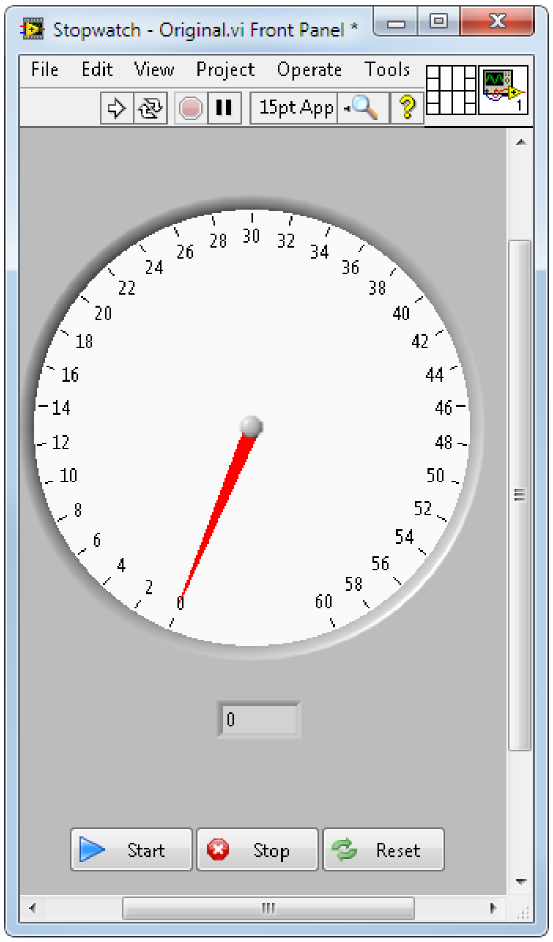
Creating a Non-LabVIEW-Looking UI

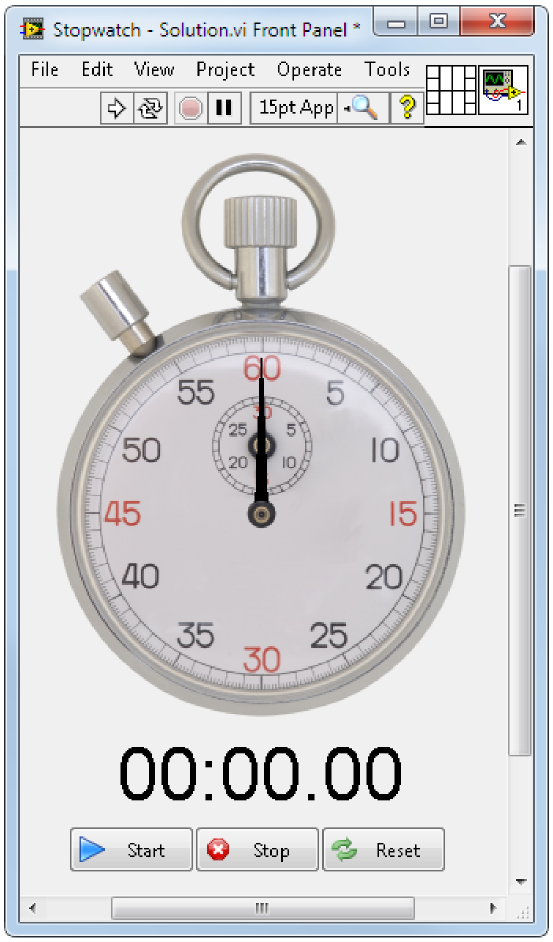
The simplest way to enhance the look of your UI is to use something other than the default gray LabVIEW front panel background and the silver controls palette. By merely changing the background color of the front panel, populating your UI with images from external sources, or even using the system controls palette, you can achieve a unique look without investing much time in customization. Plus, the system controls and indicators are typically more familiar to most users because they're designed to mimic OS style. This provides instant familiarity and therefore increased usability.

Customizing controls adds a layer of complexity and flexibility. Using the LabVIEW control editor, you can dissect each control to isolate and modify the individual lower level graphic components that compose every control. This technique makes individual controls and indicators more stylish, recognizable, or representative of the real-world signals they portray. From adding decals to buttons to changing the image of a gauge background, control customization is one of the most popular ways to improve the cosmetic appeal of your LabVIEW UI.

The stopwatch applications shown in Figure 4-13 are functionally equivalent, but the one on the right is more appealing and is more recognizable as a stopwatch.

Figure 4-13. Stopwatch Application Example





D. Extending a User Interface

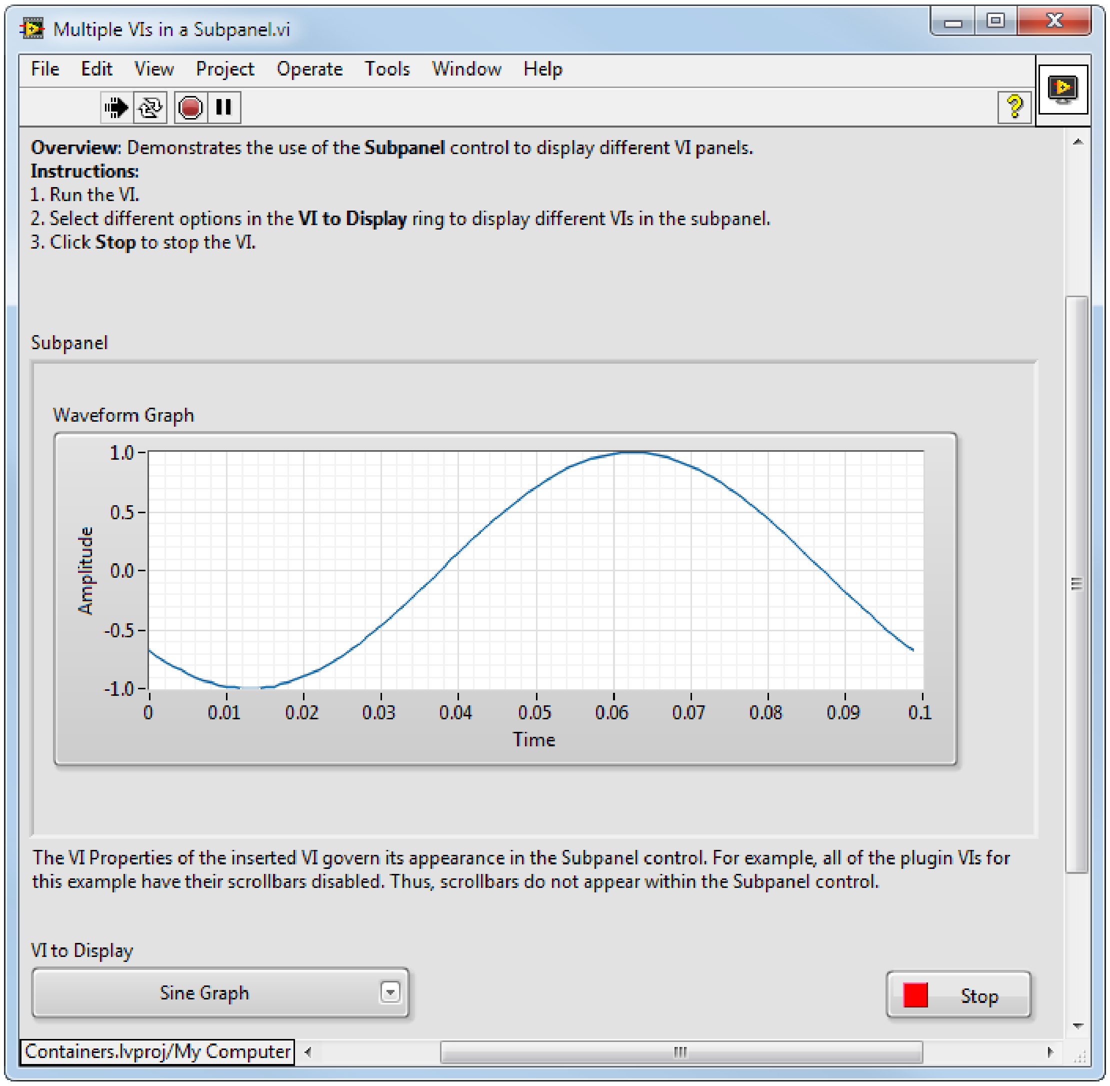
Use subpanels and tab controls to extend the available set of front panel objects.

Subpanels

Use the subpanel control to display the front panel of another VI on the front panel of the current VI. For example, you can use a subpanel control to design a user interface that behaves like a wizard. Place the **Back** and **Next** buttons on the front panel of the top-level VI and use a subpanel control to load different front panels for each step of the wizard.

Figure 4-14 shows an example of a VI that uses a subpanel control.

Figure 4-14. VI with Subpanel Control



Overview: Demonstrates the use of the **Subpanel** control to display different VI panels.
Instructions:
1. Run the VI.
2. Select different options in the **VI to Display** ring to display different VIs in the subpanel.
3. Click **Stop** to stop the VI.

The VI Properties of the inserted VI govern its appearance in the Subpanel control. For example, all of the plugin VIs for this example have their scrollbars disabled. Thus, scrollbars do not appear within the Subpanel control.

Refer to the *Front Panel Controls and Indicators* topic of the *LabVIEW Help* for more information about subpanel controls. In addition, the *Advanced Architectures in LabVIEW* course contains more examples of using subpanels in user interfaces.

Tab Controls

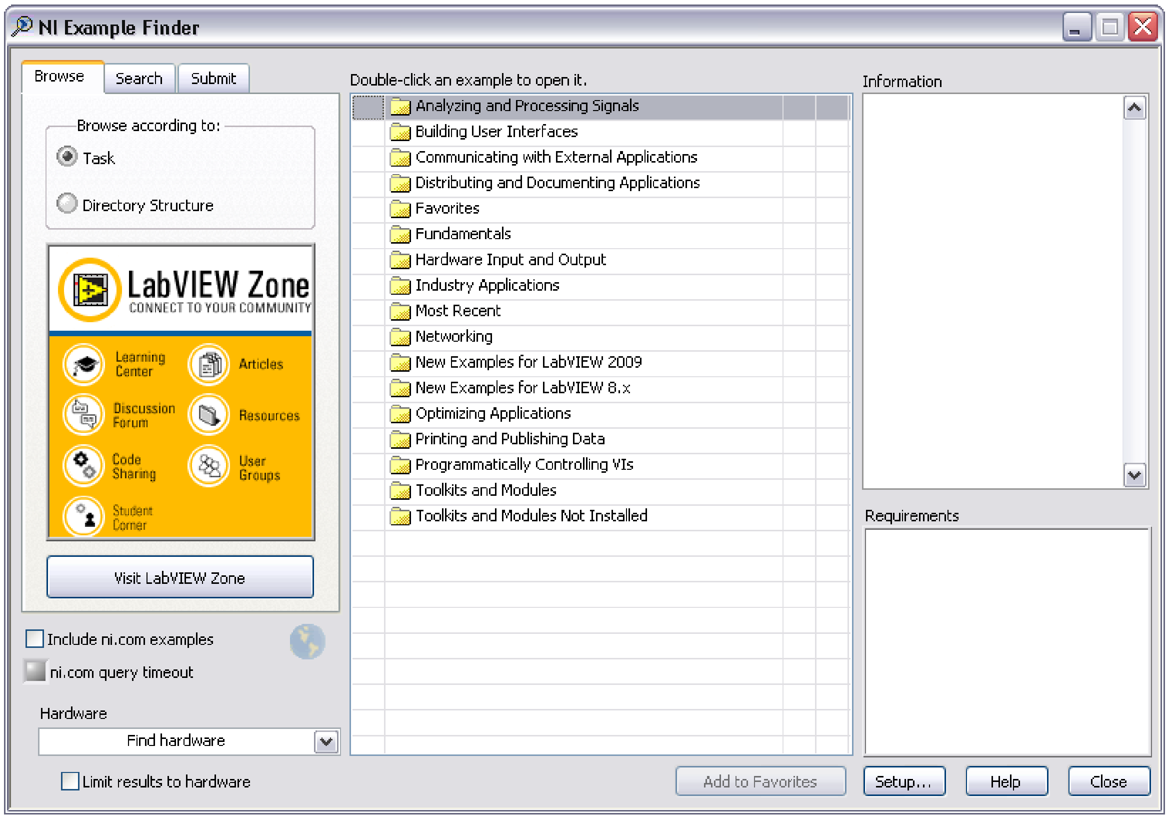
Use tab controls to overlap front panel controls and indicators in a smaller area. A tab control consists of pages and tabs. Place front panel objects on each page of a tab control and use the tab as the selector for displaying different pages.

Tab controls are useful when you have several front panel objects that are used together or during a specific phase of operation. For example, you might have a VI that requires the user to first configure several settings before a test can start, then allows the user to modify aspects of the test as it progresses, and finally allows the user to display and store only pertinent data.

On the block diagram, the tab control is an enumerated type control. Terminals for controls and indicators placed on the tab control appear as any other block diagram terminal.

The *NI Example Finder*, shown in Figure 4-15, is a VI that uses tab controls to organize the user interface.

Figure 4-15. NI Example Finder



Browse according to: [✓ Task] Directory Structure

Include ni.com examples

ni.com query timeout

Hardware: Find hardware

Limit results to hardware

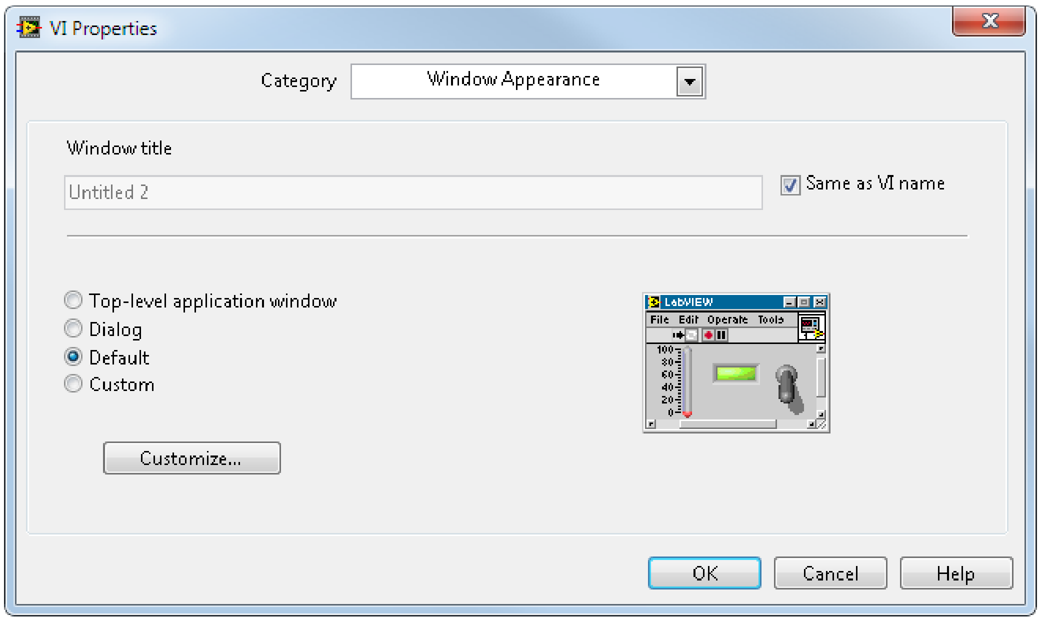
Double-click an example to open it.

- Analyzing and Processing Signals
- Building User Interfaces
- Communicating with External Applications
- Distributing and Documenting Applications
- Favorites
- Fundamentals
- Hardware Input and Output
- Industry Applications
- Most Recent
- Networking
- New Examples for LabVIEW 2009
- New Examples for LabVIEW 8.x
- Optimizing Applications
- Printing and Publishing Data
- Programmatically Controlling VIs
- Toolkits and Modules
- Toolkits and Modules Not Installed

E. Window Appearance

You can customize the appearance for your user interface VIs from the Window Appearance page of the VI Properties dialog box. The options you configure on this page, shown in Figure 4-16 apply to the VI when it is running.

Figure 4-16. VI Properties Dialog Box—Window Appearance Page



Category: Window Appearance

Window title: Untitled 2

[✓] Same as VI name

( ) Top-level application window
( ) Dialog
[✓ Default]
( ) Custom

Use the options on this page to change how the user interacts with the application. You can restrict access to LabVIEW features and change the way the window looks and behaves.

You can choose from the following style options:

- **Top-level application window**—Use for the main UI for the application. This option affects the UI in the following ways:
  - Hides scrollbars and the toolbar
  - Does not allow window resizing
  - Shows the front panel when the VI is called
- **Dialog**—Use for subVIs that you want to launch dynamically as dialog windows. This option affects the UI in the following ways:
  - Does not allow you to interact with other LabVIEW windows
  - Hides the menu bar, scrollbars, and tool bar
  - Does not allow window resizing
  - Shows the front panel when the VI is called
- **Default**—This is the same style as the LabVIEW development environment.
- **Custom**—Use this option to create your own window configurations.

Tip Select another style before selecting **Custom** to use that style as a starting point.

F. User Documentation

It is important to provide meaningful documentation to support users of the VI and other developers who may inherit the VI. Documentation should be systematically organized and include three types of information: concepts, procedures, and reference material. Different users have different documentation needs.

Labels and Captions

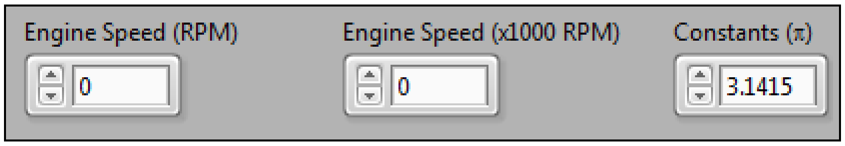
Effective use of labels and captions document the user interface and can improve the usability of front panels that serve as user interfaces.

Labels

The name of a control or indicator should describe its function. The **Context Help** window displays labels as part of the connector pane. If the default value is essential information, place the value in parentheses next to the name in the label. Include the units of the value if applicable.

For Boolean controls, use the name to give an indication of which state (TRUE or FALSE) corresponds to the function (for example, ON) and indicate the default state in parenthesis. For checkboxes and radio buttons, the user can click the Boolean text of the control and the value of the Boolean control changes. Free labels next to a Boolean control can help clarify the meaning of each position on a switch. For example, use free labels like **Cancel**, **Reset**, and **Initialize** that describe the action taken.

Figure 4-17. Labels with Units and Default Values



Captions

Front panel objects also can have captions. You can use captions instead of labels to localize a VI without breaking the VI. Unlike a label, a caption does not affect the name of the object, and you can use it as a more descriptive object label. The caption appears only in the front panel window.

If the control is visible to the user, use captions to display a long description and add a short label to prevent using valuable space on the block diagram. For example, when you label a ring control or slide control that has options for volts, ohms, or amperes, select an intuitive name for the control. A caption such as “Select units for display” is a better choice than “V/O/A”.

If you assign the object to a connector pane terminal, the caption appears in a tip strip when you use the Wiring tool to move the cursor over the terminal on the block diagram. The caption also appears next to the terminal in the **Context Help** window if you move the cursor over the connector pane or VI icon.

Right-click the object and select **Visible Items»Caption** from the shortcut menu to show or hide the caption.

Note When the text of the caption changes, LabVIEW does not need to recompile the VI or its callers.

Descriptions for Controls and Indicators

Include descriptions for each control and indicator. The user must understand how to use each control and the information conveyed by each indicator. A good description includes the following information:

- Functionality
- Default value
- Valid range

To add documentation for a control or indicator, right-click the control/indicator and select **Properties**. In the **Properties** dialog box, select the **Documentation** tab.

VI Properties

Create and edit VI descriptions by selecting **File»VI Properties** and selecting **Documentation** from the **Category** pull-down menu. The VI description is often the only source of information about a VI available to the user. The VI description appears in the **Context Help** window when you move the cursor over the VI icon and in any VI documentation you generate.

Include the following items in a VI description:

- An overview of the VI
- Instructions for using the VI
- Descriptions of the inputs and outputs

You also can link from the VI to an HTML file or compiled help file. Refer to the *Documenting and Printing VIs* topic of the *LabVIEW Help* for more information about creating help files.

Print VI Documentation

Print or save VIs and VI documentation to keep a record you can refer to later. Select **File»Print** to print more comprehensive information about a VI, including information about the front panel, block diagram, subVIs, controls, VI history, and so on.

You can print VI documentation or save it to HTML, RTF, or text files by selecting **File»Print**. You can select whether to print documentation for a single VI or for multiple VIs. You also can select a built-in documentation style or create a custom style for documentation.

The documentation you create can include the following items:

- Icon and connector pane
- Front panel and block diagram
- Controls, indicators, and data type terminals
- Labels and captions for controls and indicators
- VI and object descriptions
- VI hierarchy
- List of subVIs
- Revision history

Note The documentation you create for certain types of VIs cannot include all the previous items. For example, a polymorphic VI does not have a front panel or a block diagram, so you cannot include those items in the documentation you create for a polymorphic VI.

Help Files

Use the **Print** dialog box to help you create HTML source material for the help documents. You can edit the HTML file as needed using any text editor. You can then ship the HTML with the application.

You also can link to the help files directly from a VI. Link VIs to the **Help** menu using the **Documentation** page of the **VI Properties** dialog box.

Refer to the *Documenting and Printing VIs* topic of the *LabVIEW Help* for more information about creating help files.

To practice the concepts in this section, complete Exercise 4-2.

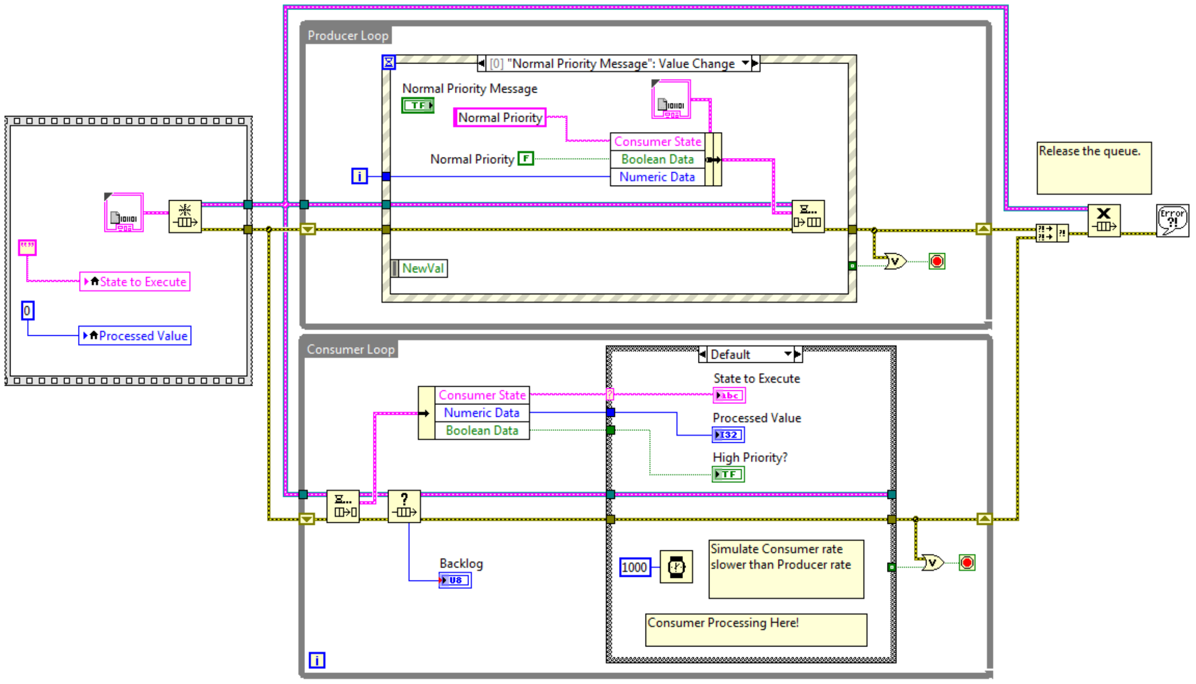
G. User Interface Initialization

You must initialize the user interface controls and indicators for your application. This helps to ensure that the application is in a known good state when it begins. There are several techniques you can use to initialize a design pattern.

Initializing with a Single Frame Sequence Structure

One way to initialize a design pattern uses a single frame Sequence structure that executes before the design pattern executes. For example, the producer/consumer design pattern shown in Figure 4-18 uses a single frame Sequence structure to initialize the queue and the front panel controls. Notice that the Sequence structure uses a local variable to initialize front panel controls. This is an acceptable use of local variables.

Figure 4-18. Producer/Consumer with Initialization



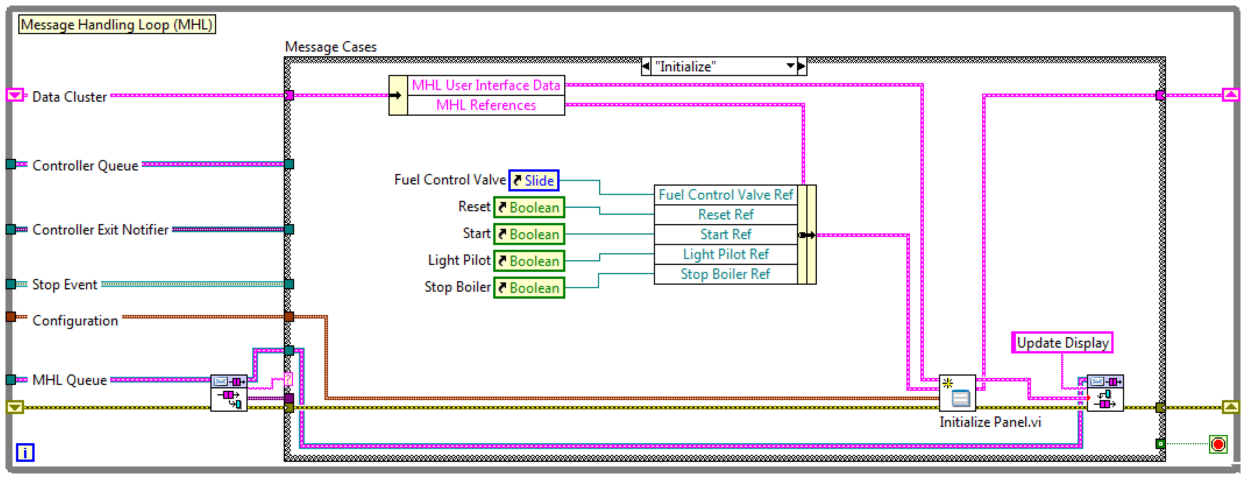
The initialization Sequence structure shown in Figure 4-18 consists of readable code that you can scale as more objects require initialization. The Sequence structure guarantees that the flow of data controls the order of execution. The producer/consumer design pattern executes only after the Sequence structure completes all the initialization steps. When you use a single frame Sequence structure to initialize a design pattern, you can control the execution of objects that do not have dataflow control, such as local variables. In Figure 4-18, the **Timing Control** local variable must initialize to 0 before the producer/consumer design pattern can execute. If the **Timing Control** local variable was not enclosed in a structure, you could not guarantee when the local variable would execute.

If the sequence structure gets too large, consider creating a subVI that contains the initialization data.

Initializing with an Initialization State

Many applications are based on the state machine design pattern. In the typical flow of a state machine, the application initializes, performs some work, and performs cleanup operations. To initialize a state machine design pattern, create an initialization state. Use the initialization state to set up files, open file references, open data acquisition devices or instruments, or perform any other initialization necessary for execution to begin. Figure 4-19 shows an initialization state for a state machine design pattern.

Figure 4-19. Message Handling Loop with Initialization State



Initializing from Disk

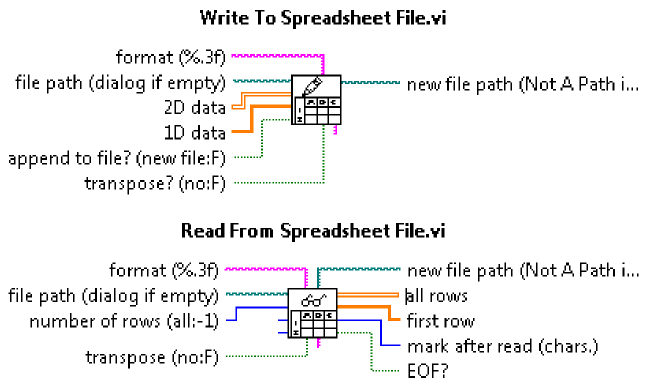
Reading initialization constants from disk is especially helpful if you want to be able to load different values for the same constants as a result of switching out system hardware or software.

The initialization file should contain constants associated with UI values as well as other application data.

CSV Files

Comma separated values (CSV) files are stored as text files. To read a specific value from a CSV file, you must either parse through the entire file and search for the text you are looking for or know the specific row and column of the data. To create CSV files, use spreadsheet file functions to read and write. You can use tabs or commas as delimiters in CSV files.

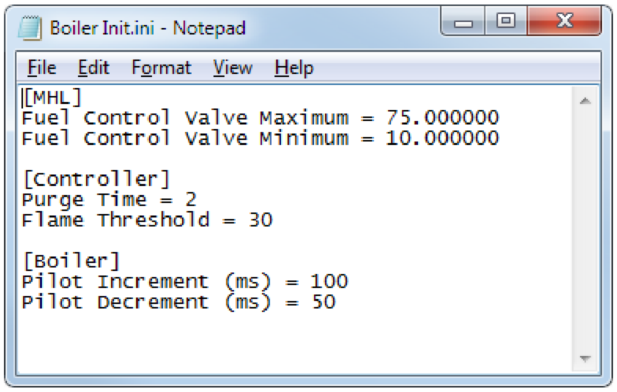
Figure 4-20. Write to Spreadsheet and Read From Spreadsheet VIs Used with CSV Files


Write To Spreadsheet File.vi
format (%.3f)
file path (dialog if empty)
2D data
1D data
append to file? (new file:F)
transpose? (no:F)
new file path (Not A Path i...
Read From Spreadsheet File.vi
format (%.3f)
file path (dialog if empty)
number of rows (all:-1)
transpose (no:F)
new file path (Not A Path i...
all rows
first row
mark after read (chars.)
EOF?

INI Files

Initialization (INI) files are organized into sections and keys. A key is a string associated with a constant value. A section is a group of related keys. You specify the section and key that you want to write or read. In the INI file shown in Figure 4-21, MHL is the section and Fuel Control Valve Minimum and Fuel Control Valve Maximum are the keys.

Figure 4-21. INI File Example



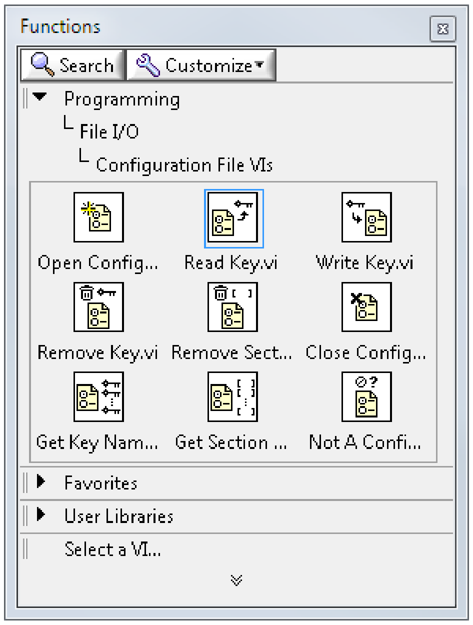
```
[MHL]
Fuel Control Valve Maximum = 75.000000
Fuel Control Valve Minimum = 10.000000

[Controller]
Purge Time = 2
Flame Threshold = 30

[Boiler]
Pilot Increment (ms) = 100
Pilot Decrement (ms) = 50
```

To read data from an INI file, use the Configuration File VIs palette, shown in Figure 4-22.

Figure 4-22. Configuration File VIs Palette



To practice the concepts in this section, complete Exercise 4-3.

H. User Interface Testing

You should test user interfaces for usability throughout the development process. Usability measures the potential of a system to accomplish the goals of the user. Some factors that determine system usability are ease-of-use, visual consistency, and a clear, defined process for evolution.

Systems that are usable enable users to concentrate on their tasks and do real work rather than focusing on the tools they use to perform tasks.

Usable VIs have the following characteristics:

- Easy to learn
- Efficient to use
- Provide quick recovery from errors
- Easy to remember
- Enjoyable to use
- Visually pleasing

Usability applies to every aspect of a VI with which a user interacts, including hardware, software, menus, icons, messages, documentation, and training. Every design and development decision made throughout the VI life cycle has an effect on usability. As users depend more and more on software to get their jobs done and become more critical consumers of software, usability can be a critical factor that ensures that VIs are used.

Usability Testing

Usability testing is a cyclic process, resulting in several iterations of design and test. A typical usability study includes three iterations and a final meeting. Each iteration includes three steps—meeting, testing, and report.

- **Meeting**—Gather information, consider the different issues encountered, and determine guidelines. Set goals and purposes during this step.
- **Testing**—Test, analyze, or evaluate the product. Depending on the iteration of usability testing, conduct a usability evaluation, cognitive walkthrough, or formal study of the software.
- **Report**—Present the test results. Issues are typically listed by priority. Reports increase in detail and complexity with each stage.

To practice the concepts in this section, complete Exercise 4-4.

Self-Review: Quiz

1. Match each front panel object style to when it should be used

Silver	a. Use for most subVI front panels
Modern	b. Use when the VI will run on different operating systems
Classic	c. Use when VI will need to run on low-color monitors
System	d. Use for customer-facing front panels

2. Which of the following is *not* true about creating a user interface prototype?
   a. Helps get customer buy-in early in the development process
   b. Fill in as much block diagram functionality as possible
   c. Time spent creating the prototype should be limited
   d. Customers could get the wrong idea about how far along development has come

3. Match each initialization file format to the description that best fits it.

INI File	a. A comma or tab-delimited spreadsheet file
CSV	b. Divided into sections and keys for easy look-up

Self-Review: Quiz Answers

1. Match each front panel object style to when it should be used

Silver	**d. Use for customer-facing front panels**
Modern	**a. Use for most subVI front panels**
Classic	**c. Use when VI will need to run on low-color monitors**
System	**b. Use when the VI will run on different operating systems**

2. Which of the following is *not* true about creating a user interface prototype?
   a. Helps get customer buy-in early in the development process
   b. Fill in as much block diagram functionality as possible
   c. Time spent creating the prototype should be limited
   d. Customers could get the wrong idea about how far along development has come

3. Match each initialization file format to the description that best fits it.

INI File	**b. Divided into sections and keys for easy look-up**
CSV	**a. A comma or tab-delimited spreadsheet file**

Notes

5 Managing and Logging Errors

No matter how confident you are in the VI you create, you cannot predict every problem a user can encounter.

This lesson describes several approaches to developing software that gracefully responds to different types of errors. You will learn how to determine whether an error should be handled locally or globally and when you should log error data to disk for later analysis.

Topics

A. Error Testing

B. Local Error Handling

C. Global Error Handling

D. Error Logging

A. Error Testing

Without a mechanism to check for errors, you know only that the VI does not work properly. It is important to consider how your system will respond to different types of errors when they occur.

Mission-critical applications require more complete error management than simpler LabVIEW applications. Comprehensive error management may increase development time, but it will save troubleshooting time in the long run.

When you develop your error test strategy, the first step is to identify the different errors that are likely to occur. Then determine how the system should respond to each type of error. Finally, develop tests to ensure that the system behaves as expected for each error.

While LabVIEW provides basic tools for error and exception handling, implementing a comprehensive error handling strategy is challenging and requires significant programming effort. A comprehensive error handling strategy requires both the ability to respond to specific error codes, for example: ignoring an error that does not affect the system, and the ability to take general actions based upon the type of error that occurs, for example: log all warnings to a file.

When creating software, you can develop separate error handling code for two areas of the error handling strategy—development and deployment. During development, the error handling code should be noticeable and should clearly indicate where errors occur. This helps you determine where any bugs might exist in a VI. During deployment, however, you want the error handling system to be unobtrusive to the user. The error handling system should allow a clean exit and provide very clear prompts to the user.

Force Errors

Error tests help you determine if the error handling strategy you designed works correctly and make sure the error handling code performs as you expect. Test the error handling code by creating test plans that force errors. The error test should verify that the proper errors are reported and recovered. The error test plan also should verify that the error handling code handles errors gracefully.

Verify System Behavior

The error test should verify that the proper errors are reported and recovered. The error test plan should also verify that the error handling code handles errors gracefully.

The behavior of the system will be different for different error classifications. Critical errors occur when the system cannot proceed safely. These errors should result in a safe system shutdown. Non-critical errors should be handled and the system should continue execution. A warning may only require user notification.

B. Local Error Handling

By default, as a VI runs, LabVIEW tests for errors at each execution node. If no error occurs, the node executes normally. If LabVIEW detects an error, the node does not execute. If the error out terminal of that node is unwired, LabVIEW suspends execution, highlights the node and displays an error dialog. If the error out terminal of that node is wired, then LabVIEW passes the error to the next node. The next node will repeat the check, skip execution, and either suspend execution or pass the error to the next node. At the end of execution, if the error passes all the way through the application, LabVIEW displays an error dialog.

This method is not always the best way to handle errors. For example, you may have some code that must execute regardless of an error (hardware shutdown, for example).

To disable automatic error handling, follow these steps:

1. Navigate to **Tools»Options»Block Diagram**.
2. In the Error Handling section, disable **Enable automatic error handling in new VIs** and **Enable automatic error handling dialogs**.
3. Click OK.

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. 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.

You can design corrective error processing that tries to determine and fix the error. This allows each module to implement error correcting code. For example you could develop a Read File I/O VI that can fix errors which might have occurred with the Open/Create/Replace File VI.

Handling errors locally is the primary approach for minor errors that can be prevented or controlled and that should not result in system shutdown. Whenever possible, correct errors in the VI where the errors occur. If you cannot correct an error within the module, return the error to the calling VI so that the calling VI can correct, ignore, or display the error to the user. When you implement a module, make sure it handles any errors that it can. Include error clusters to pass error information into and out of the module.

Preventing Errors

You can design a system that either ensures an error cannot occur or proactively determines if an error will occur.

For example:

- It might be necessary to catch any problems that are in a configuration that is being passed to a data acquisition driver. The system can prevent the data acquisition driver from even executing if there are errors.

- If a control value will be used as the denominator of a divide operation, develop code that checks that the control value is non-zero before the divide operation.
- If a control value must remain between -10 and 10, set control limits on the front panel.

The goal of this technique is to prevent most errors from occurring in the first place.

Handling Specific Errors

Handling specific errors involves performing some sort of action because a specific error occurs. The first step in determining how to handle errors for a VI is to answer the following questions:

- What could go wrong?
- Can I do anything about it?
- Does responding to this error require higher-level code?
- How will the error affect subsequent code?
- How should the error affect subsequent code?

Only handle errors that you can do something about. Any errors that should result in system shutdown should be passed to the top-level application. Once you identify the errors that you can address, there are three main approaches to handling that error locally:

- Ignore the error—Some libraries or functions may generate errors that you expect. For example, the File Dialog VI returns error 43 if the user cancels the dialog. If you want to allow the user to cancel that window, then you should clear and ignore this error.
- Retry the operation that caused the error—If the error is caused by an intermittent communication or connectivity problem, retrying the operation may result in success. In general, it is best to limit the number of retry attempts. If you reach the retry limit, try another method for handling the error.
- Correct the error.

Document any assumptions that you make and errors that you decide to ignore.

Place error handling code as close to the error-generating code as you can to minimize the propagation of the error. This also avoids confusion about which piece of code generates the error. Additionally, some error handling methods, like retrying the operation that caused the error, are only effective when the error is detected and handled immediately.

Clearing Errors

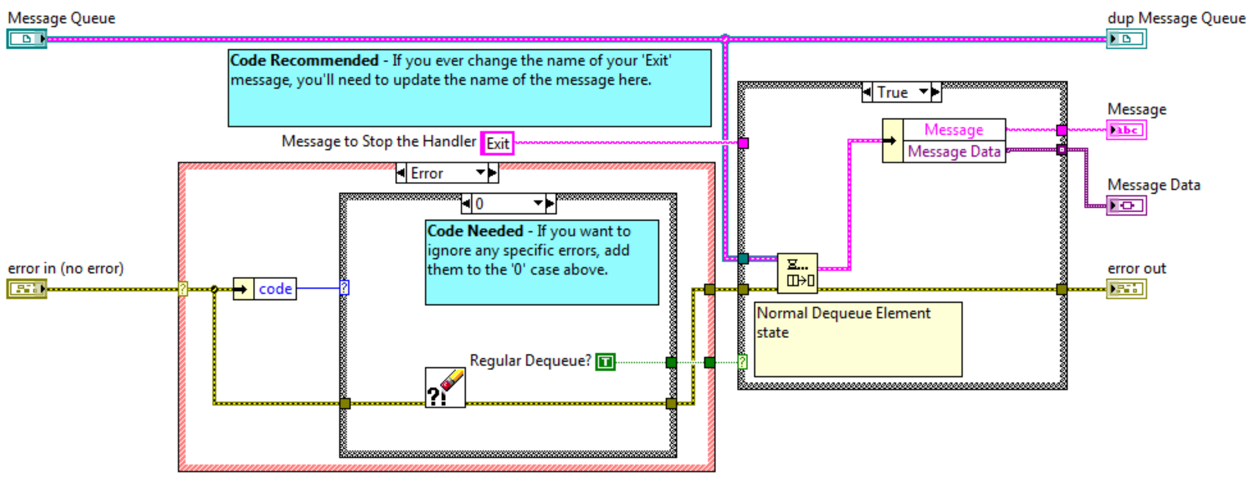
Clear errors only after handling the error locally or passing the information to a global error handler. Methods for clearing errors include using the Clear Errors VI or an empty error cluster constant.

Use the Clear Errors VI if there is additional code that executes after the error is handled. If the error handling is the last piece of code to execute, you can use an empty error cluster constant instead.

Queued Message Handler Local Error Handling

The Dequeue Message VI, shown in Figure 5-1, can ignore errors when it reads data from the message queue. The Error case reads the error code from the **error in** cluster. The case structure reads the error code and clears specific errors. By default, the only error that it clears is error 0 (no error). Modify this case to include any other errors that you want to ignore/clear when this VI executes. The errors to ignore depend on your application. For example, if you are reading the queue over a network, you may want to ignore timeout errors.

Figure 5-1. Dequeue Message VI


Message Queue
dup Message Queue
Code Recommended - If you ever change the name of your 'Exit' message, you'll need to update the name of the message here.
Message to Stop the Handler
Exit
True
Message
Message Data
Message
Message Data
Error
0
Code Needed - If you want to ignore any specific errors, add them to the '0' case above.
error in (no error)
code
error out
Normal Dequeue Element state
Regular Dequeue?

C. Global Error Handling

Some errors require action from a higher-level VI. For example, critical errors, which occur when the application cannot continue operating, require a safe system shutdown. You can design a system that passes error data out of the VI to the calling VI or a global handler. If you choose to pass the error out to the calling VI, the caller will have to either handle the error or pass it along to its caller. Errors can come from one of the following two sources:

- LabVIEW functions
- Custom errors defined by the developer

Custom Error Codes

VIs and functions in LabVIEW can return numeric error codes. Each product or group of VIs defines a range of error codes. Refer to the *Ranges of LabVIEW Error Codes* topic of the *LabVIEW Help* for error code tables listing the numeric error codes and descriptions for each product and VI grouping.

In addition to defining error code ranges, LabVIEW reserves some error code ranges for you to use in your application. You can define custom error codes in the range of –8999 through –8000, 5000 through 9999, or 500,000 through 599,999.

Determine where possible errors can occur in the VI and define error codes and messages for those errors. You can create custom error messages that are meaningful for your VIs.

National Instruments recommends that you define custom error codes in the range of 5000 to 9999.

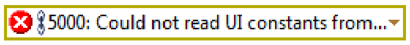
Define Custom Error Codes

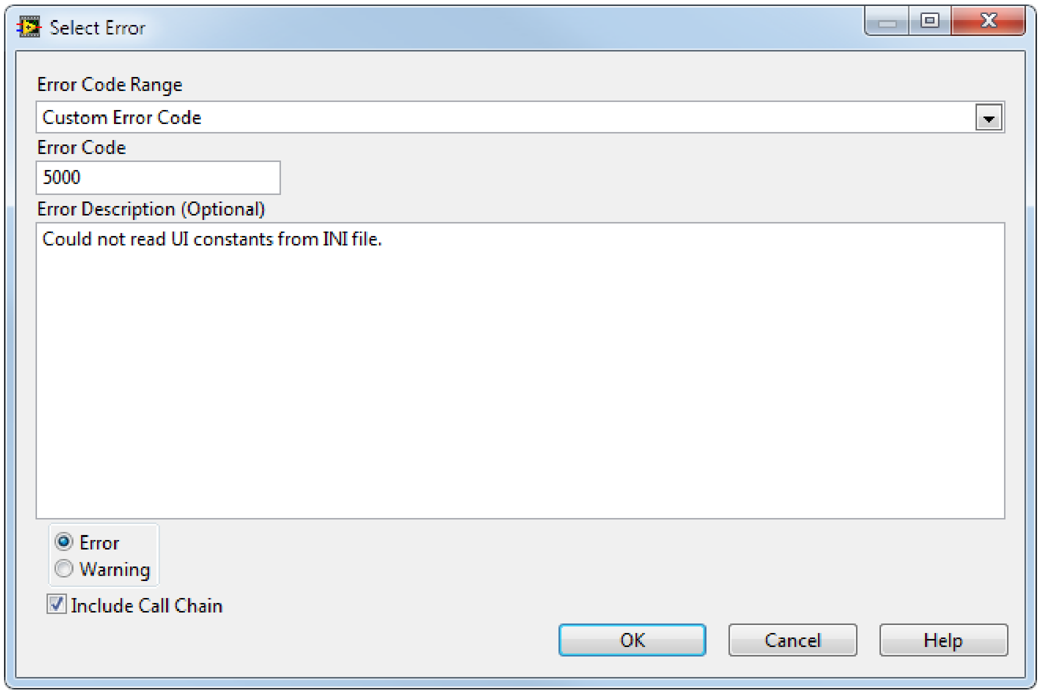
The approach that you choose for defining custom error codes depends on how those custom codes need to be distributed.

Error Ring

An Error Ring constant defines a single custom error code for one particular instance in your application. The Error Ring constant allows you to choose an existing LabVIEW error cluster (code and description) or create your own custom error code, as shown in Figure 5-2.

Figure 5-2. Define Errors with the Error Ring Constant


5000: Could not read UI constants from...



Error Code Range: Custom Error Code

Error Code: 5000

Error Description (Optional): Could not read UI constants from INI file.

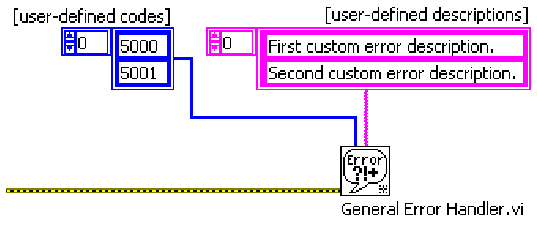
[✓ Error] Warning

[✓ Include Call Chain]

General Error Handler

Use the General Error Handler VI to define multiple custom error codes that your application does not widely use and you will not distribute in built applications or shared libraries. For this VI, create arrays of custom error codes and descriptions for your custom error codes. If an incoming error matches one in user-defined codes, the General Error Handler VI returns the corresponding description from the user-defined description input, as shown in Figure 5-3.

Figure 5-3. General Error Handler VI

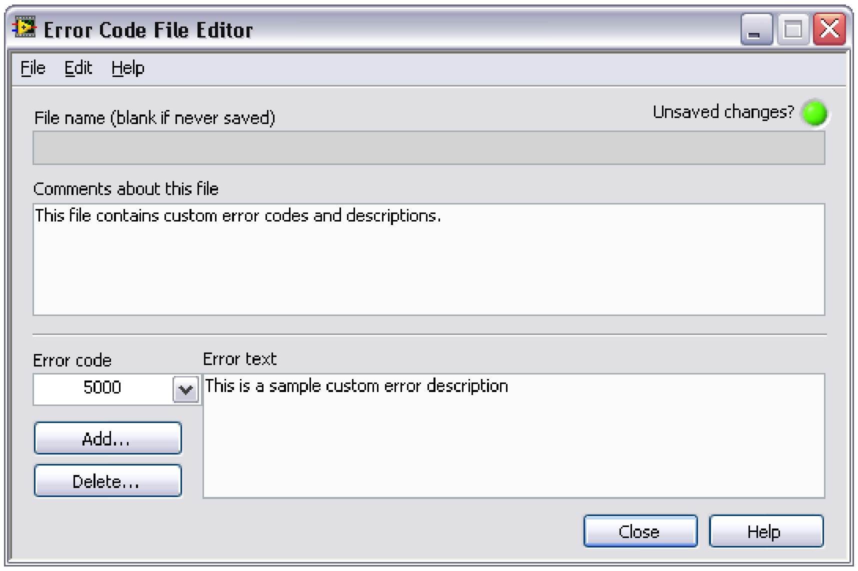

[user-defined codes]
0
5000
5001
[user-defined descriptions]
0
First custom error description.
Second custom error description.
Error ?!+
General Error Handler.vi

Refer to the *Defining Custom Error Codes Using the General Error Handler VI* topic of the *LabVIEW Help* for more information on using this VI.

Error Code Editor

Use the Error Code Editor to define multiple error codes that you use throughout your application and will distribute in built applications or shared libraries. Error Code File Editor creates an XML-based text file in the `<labview>\user.lib\errors` directory. To launch this dialog, shown in Figure 5-4, select **Tools»Advanced»Edit Error Codes** to launch the Error Code File.

Figure 5-4. Error Code File Editor



File name (blank if never saved)

Comments about this file
This file contains custom error codes and descriptions.

Error code: 5000
Error text: This is a sample custom error description

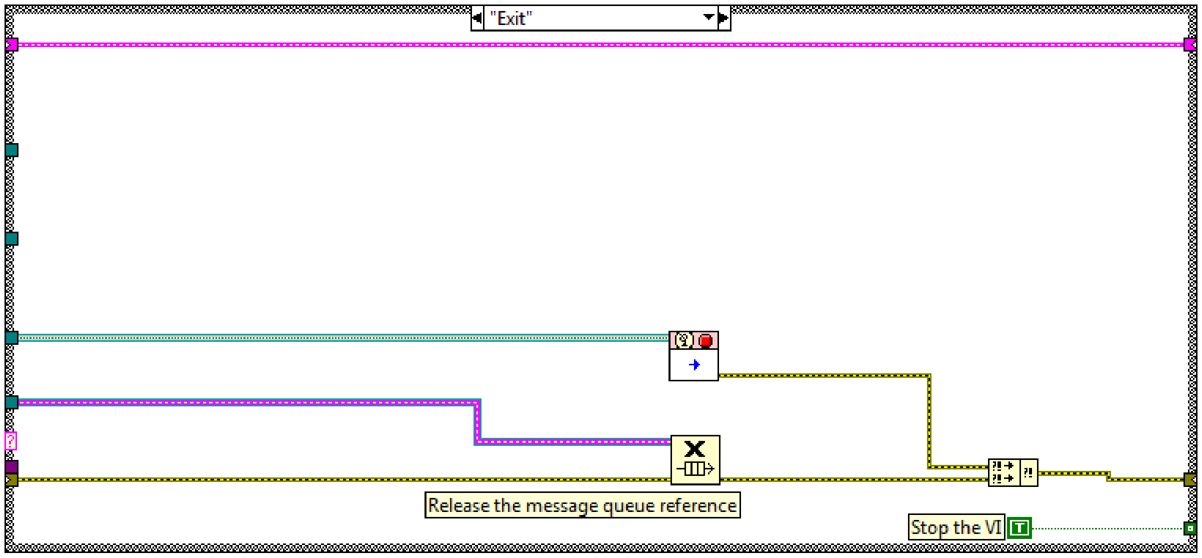
Refer to the *Defining Custom Error Codes to Distribute Throughout Your Application* topic of the *LabVIEW Help* for more information on using this tool.

After you create an error code file, refer to the *LabVIEW Error Code File Locations* Knowledgebase at `ni.com` for more information on how to deploy this file or distribute it to other development systems.

Queued Message Handler Critical Error Handling

If an error is detected in the event handling loop or `Dequeue Message.vi` in the message handling loop, the Exit case of the message handling loop, shown in Figure 5-5, executes.

Figure 5-5. QMH Critical Error Handling



The Exit message is passed out of `Dequeue Message.vi` unless you specify that the error code is a specific one that should be ignored. If you want your MHLs to shut down on a message other than Exit, you must make that change in this VI.

The Exit case of the message handling loop fires the Stop user event to stop the EHL and releases the MHL queue.

Global Error Handler VI

A global error handler consists of high-level code that checks for errors in an entire system. The global error handler should use error classification information (critical, non-critical, warning) to determine which actions to take. The global error handler should also handle the case of an unclassified error or an error with an unrecognized classification and react accordingly. Examples of actions a global error handler might take to respond to error classification (or the lack thereof) include logging the error, displaying error information in an error dialog, placing the system outputs in a safe state, and/or initiating a system shutdown/reboot.

The goal of the global error handler is to handle errors that could come from multiple locations. If an error can come only from one specific subVI, handle that error locally.

Functional global variables are commonly used to implement global error handlers that share the error response across parallel loops. The functional global variable should include at least three cases:

- Initialize—Initialize references and error data. Call this case when the application begins execution.
- Handle Errors—Handle any errors that have been passed to the functional global variable. Pass a clear error cluster out.
- Report Errors—Pass all errors that have occurred to the error out terminal for the functional global variable.

For the global error handler to be effective, you must select a method for communicating errors to that VI. If your global error handler is a functional global variable, you can call the Handle Errors case from any location in your application that generates errors that must be handled globally. For other approaches, you may use a queue to pass error data.

To practice the concepts in this section, complete Exercise 5-1.

D. Error Logging

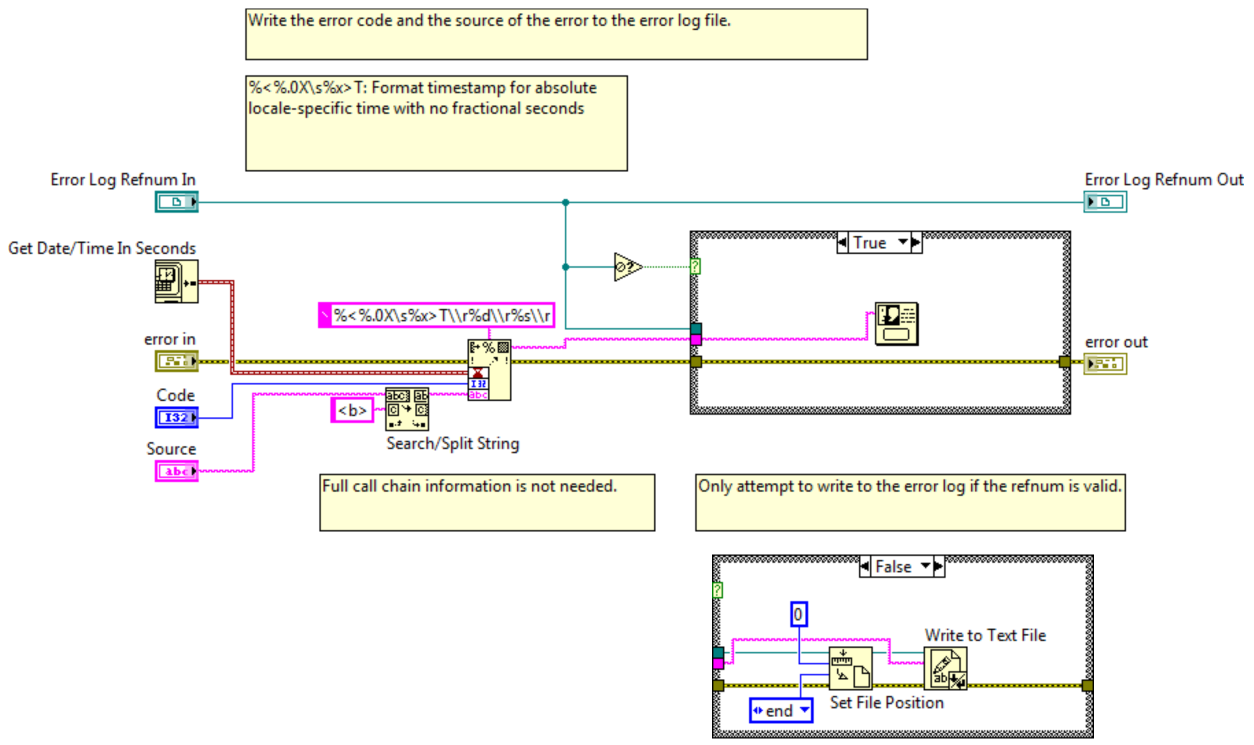
You should log information about critical errors to disk.

Error logging is a benefit for a user and a developer. Users have a historical record of what happened in case of an error and a developer can hopefully discern more about what caused an error. It is logical for the log to be readable by humans (ASCII characters). Because errors may be logged several times during an application's execution or in a quick amount of time (for example, before a operating system crashes), you want your error logging code to be very efficient and high performing. A lot of embedded systems don't have a lot of system resources to hold a large error log so you will want to either clean out the error log every once in a while or just log the most important things.

At minimum, you want to log the time of the error, the error code, and the error source. Depending on your application, you may want to log additional information.

The VI shown in Figure 5-6 shows an example of error logging. The input string input of the Format Into String function describes how each piece of information is written to the string. If the Error Log Refnum is invalid, write to the end of the file. If not, then the string is displayed to the user. The reference is only invalid if the application is running from a location where it does not have write access, such as a CD.

Figure 5-6. Error Logging Example


Write the error code and the source of the error to the error log file.
%<%.0X\s%x>T: Format timestamp for absolute locale-specific time with no fractional seconds
Error Log Refnum In
Error Log Refnum Out
Get Date/Time In Seconds
True
%<%.0X\s%x>T\\r%d\\r%s\\r
error in
error out
Code
<b>
Source
Search/Split String
Full call chain information is not needed.
Only attempt to write to the error log if the refnum is valid.
False
0
Write to Text File
end
Set File Position

Regardless of whether the error information is logged or displayed in a dialog, the information should be understandable, localizable, and indicate any actions that the user should take.

To practice the concepts in this section, complete Exercise 5-2.

Self-Review: Quiz

1. Which of the following methods are valid ways to generate a custom error code? (multiple answer)
   a. Manually create an errors.txt file
   b. Error Ring constant
   c. Simple Error Handler
   d. General Error Handler

2. What information should be logged for a critical error?
   a. Time of error
   b. Error status
   c. Error code
   d. Error source

Self-Review: Quiz Answers

1. Which of the following methods are valid ways to generate a custom error code? (multiple answer)
   - **a. Manually create an errors.txt file**
   - **b. Error Ring constant**
   - c. Simple Error Handler
   - **d. General Error Handler**

2. What information should be logged for a critical error?
   - **a. Time of error**
   - b. Error status
   - **c. Error code**
   - **d. Error source**

Notes

6

Creating Modular Code

Best practices for large application development include creating modular code. This lesson describes how to use modular code in a large application and guidelines for making large applications more maintainable. You will learn several approaches for testing code modules and integrating them into your top-level application architecture.

Topics

A. Designing Modular Applications

B. Code Module Testing

C. Integration Testing

A. Designing Modular Applications

Designing modular applications involves designing individual VIs or LabVIEW modules that perform specific functions and coordinating those modules into a larger application. In LabVIEW, a module can be a single frame of a Case structure, a single VI, or a set of VIs.

Creating modules code involves building distinct LabVIEW code, such as events or subVIs, when there is a logical division of labor or the potential for code reuse. Modular code can solve more general problems along with the specific ones. Module code also facilitates testing because you can test the VIs as you write them.

Modular Code

Modular code is divided into separate function components. It is easier to test, reuse, and maintain modular code. Creating modular code generally involves developing subVIs and project libraries to group these subVIs locally. Object-oriented code is inherently modular.

Refer to the *Object-Oriented Design and Programming in LabVIEW* course for more information about designing and implementing object-oriented code.

Coupling and Cohesion

Coupling and cohesion are the primary characteristics you use to evaluate the design quality of code modules or VIs. A modular application has low coupling and high cohesion.

Coupling

A module rarely stands by itself. A VI is called by other VIs. Top-level VIs can become plug-ins to other VIs or utilities for the operating system. Coupling refers to the connections that exist between modules. Any module that relies on another module to perform some function is coupled to that module. If one module does not function correctly, other modules to which it is coupled do not function correctly. The more connections among modules, the harder it is to determine what module causes a problem. Coupling measures how many dependencies a system of modules contains.

The fewer outputs a module exposes, the more you can change the code inside the module without breaking other modules. Low coupling also helps make bug tracking more efficient. For example, a data acquisition bug cannot occur in a VI that performs only GPIB instrument control. If a data acquisition bug appears in a GPIB VI, you know the code is probably too tightly coupled.

The dataflow model of LabVIEW automatically encourages low coupling among modules. The dataflow model means each VI has only the information it needs to do its job. A VI cannot change data in another VI. You can introduce other communication paths, such as global variables, queues, I/O connections, and so on, that can increase coupling, but a typical VI is self-sufficient and couples only to the subVIs it calls. The goal of any hierarchical architecture is to make sure that the implementation uses low or loose coupling.

Cohesion

In a well-designed VI, each module fulfills one goal and thus is strongly cohesive. If a single module tries to fulfill more than one goal, you should probably divide it into several modules. When a module tries to accomplish multiple goals, the block diagram becomes harder to read, and the code for one goal can affect how the code for another goal works. In this case, fixing a bug in one goal might break the code of another goal.

A VI with strong cohesion has one goal per module. Strong cohesion decreases overall development time. You can more easily understand a single module if it focuses on one task. You can read a clean, uncomplicated block diagram quickly and make deliberate changes with confidence.

The Mathematics VIs and functions that ship with LabVIEW demonstrate strong cohesion. For example, the Sine function performs a single goal, and performs it well. It is a perfect example of a function that has strong cohesion. The name of the function clearly indicates what the function accomplishes. Other VIs that have strong cohesion are the File I/O VIs and functions. The File I/O VIs and functions perform sequential cohesion because one module must execute before another module can execute. The Open File, Read File, Write File, and Close File functions each perform a single goal within each module.

B. Code Module Testing

It is essential to test each module as you complete it. But how do you know what to test and if the tests you perform are appropriate? Because there are so many possible inputs for a VI, it can be incredibly difficult to determine if a VI has been built correctly and works as you intended. Testing every possible combination of inputs is an inefficient and ineffective strategy. The excessive computation time such a test would require is impractical. A more effective strategy is to develop test cases that can identify the largest number of errors in the application. The goal of a good test plan is to find the majority of the errors.

Creating a Test Plan for Individual VIs

The individual component VI undergoing testing is referred to as the unit under test (UUT). Unit testing is necessary before system integration testing. You should design a test plan for each module before you build any VIs. Developing a test plan before you build a VI helps you know exactly what the VI should accomplish. The test plan also helps you understand the expected inputs and outputs to each VI.

To develop a test plan, you must determine what you need to test. Use the following list to help identify what you need to test.

- Test all requirements in the requirements document
- Identify areas that need usability testing, such as the user interface
- Test the error handling and reporting capabilities of the VI
- Test areas of the VI that have performance requirements

It is much easier to develop a test plan when you have a requirements document because you know you need to test each requirement in the document. You can begin by writing a test plan that tests each specification or requirement. However, testing the requirements does not always guarantee that the software functions based on the requirements. To develop a complete test plan, include different forms of testing for your VIs.

Functional Testing

Functional tests focus on the most important aspects of a VI. Use functional tests to determine if a VI works and functions as expected. A functional test involves more than passing simple test data to a VI and checking that the VI returns the expected data. Although this is one way to test a VI, you should develop a more robust test.

A functional test should test for expected functionality as noted in the requirements document. An easy way to create a functional test plan is to create a table with three columns. The first column indicates the action that the VI should perform, as noted in the requirements document. The second column indicates the expected result, and the third column indicates if the VI generates the expected result. Using this table is a good way to create a usable test plan for the listed requirements.

Functional tests can check for the following common problems.

- Boundary conditions for each input, such as empty arrays and empty strings, or 0 for a size input. Be sure floating-point parameters deal with `Inf` and `NaN`.
- Invalid values for each input, such as `-3` for a size input.
- Strange combinations of inputs.
- Missing files and bad path names.
- What happens when the user clicks the **Cancel** button in a file dialog box?
- What happens if the user aborts the VI?

Testing Boundary Conditions

A boundary condition is a value that is above or below a set maximum. Values at or just outside of boundaries are common sources of function errors. To test for boundary conditions, if a requirement calls for a maximum value, you should test (`maximum value +1`), (`maximum value -1`), and `maximum value`. Creating tests this way exercises each boundary condition. Perform the same kind of tests for minimum values.

Hand Checking

Another common form of functional testing is simple hand checking. This type of testing occurs during the development of the VI but can help during the testing phase. If you know the value that a VI should return when a certain set of inputs are used, use those values as a test. For example, if you build a VI that calculates the trigonometric sine function, you could pass an input of 0 to the VI and expect the VI to generate a result of 0. Hand checking works well to test that a VI functions.

Be careful that hand checks do not become too difficult to work with. For example, if you pass the value 253.4569090 to the sine function, it would be difficult to determine if the VI generated the correct results. You increase the possibility of incorrectly determining the expected value of a VI when you hand check with values that appear to add complexity to the test. In general, passing more complex numbers to a VI does not provide better test results than passing simple values such as 0 or pi for the sine function. Make sure to pass data that is of the same data type but do not complicate the test plan by using numbers that make it difficult for you to calculate the expected result.

Functional Testing Tools

The LabVIEW Unit Test Framework Toolkit provides tools you use to check VIs for functional correctness. Use the **Project Explorer** window to create, configure, manage, and execute tests interactively in a LabVIEW project. You also can execute tests programmatically by using the Unit Test Framework VIs.

Refer to the *Managing Software Engineering for LabVIEW* course for more information about the Unit Test Framework Toolkit.

To practice the concepts in this section, complete Exercise 6-1.

C. Integration Testing

Use integration testing to test the individual VIs as you integrate them into a larger system. You perform integration testing on a combination of units. Unit testing usually finds most bugs, but integration testing can reveal unanticipated problems. Modules might not work together as expected. They can interact in unexpected ways because of the way they manipulate shared data.

You also can perform integration testing in earlier stages before you put the whole system together. For example, if a developer creates a set of VIs that communicates with an instrument, he can develop unit tests to verify that each subVI correctly sends the appropriate commands. He also can develop integration tests that use several of the subVIs in conjunction with each other to verify that there are no unexpected interactions.

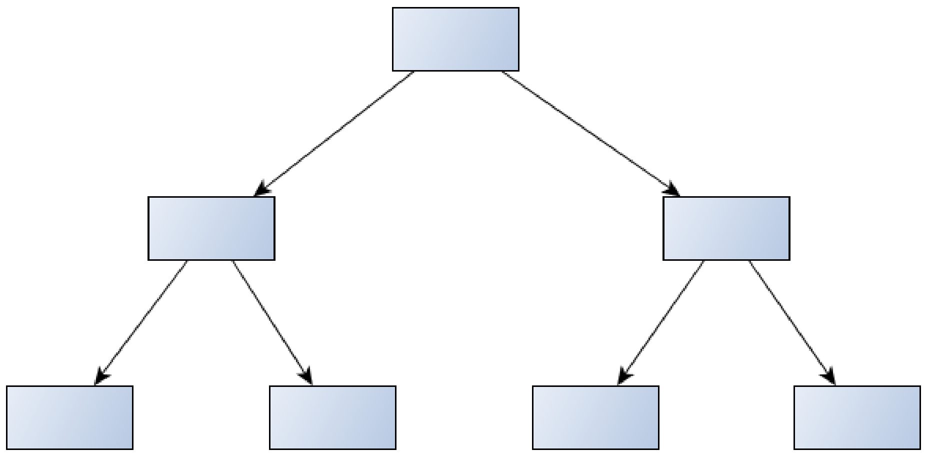
You can use a top-down, bottom-up, or sandwich integration testing technique. Do not perform integration testing as a comprehensive test in which you combine all the components and try to test the top-level program, an approach known as big bang testing. This method can be expensive because it is difficult to determine the specific source of problems within a large set of VIs. Instead, consider testing incrementally with a top-down or bottom-up testing approach.

Top-Down Integration Testing

Top-down integration testing involves testing VIs that are on the top of the hierarchy and then working down the hierarchy levels. In order to test a top-level VI, you must build stub VIs that can respond to the top-level VI. A stub VI is a non-functional prototype that represents a future subVI. Create an icon and create a front panel with the necessary inputs and outputs for each stub VI. As you continue integrating VIs, you replace the stub VIs with actual VI components. Conduct tests as you integrate each VI. As the tests move downward in the hierarchy, remove the stub VIs.

Top-down integration testing allows you to identify major control errors with the VI early in the testing process. The biggest disadvantages to top-down testing are the time and effort required to build stub VIs and delayed testing of interactions between low-level modules. Figure 6-1 shows an example of top-down testing.

Figure 6-1. Top-Down Testing Process



Regression Testing

Regardless of the approach you take, you must perform regression testing at each step to verify that the previously tested features still work. Regression testing consists of repeating some or all previous tests. If you need to perform the same tests numerous times, consider developing representative subsets of tests to use for frequent regression tests. You can run these subsets of tests at each stage. You can run the more detailed tests to test an individual set of modules if problems arise or as part of a more detailed regression test that periodically occurs during development.

Regression testing enables you to find out about newly introduced problems earlier in the integration process. This type of testing may add significant time to the development process, especially for large applications.

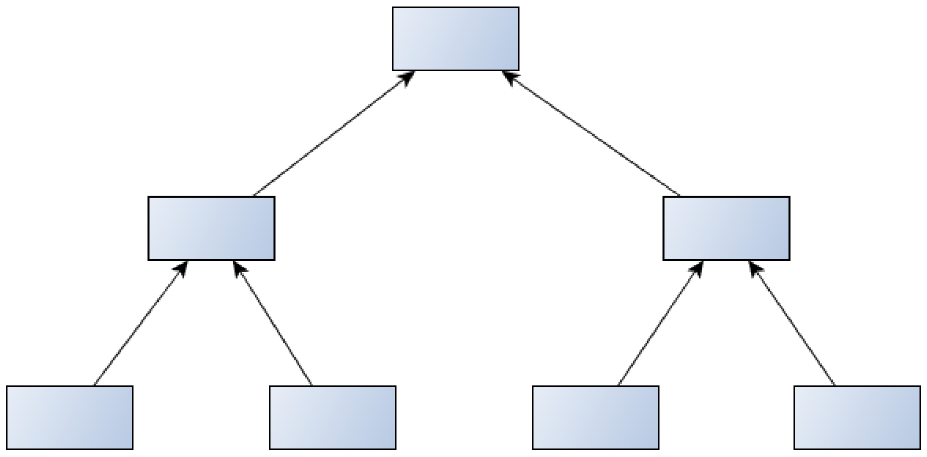
To practice the concepts in this section, complete Exercise 6-2.

Bottom-Up Integration Testing

Bottom-up integration testing involves integrating and testing the lower-level VIs in the hierarchy and moving up the hierarchy. This form of integration testing requires you to first test the individual VI, then create a driver VI or a control program to coordinate the inputs and outputs of the VI you are integrating. As you continue this process, you can replace the control program with VIs that are higher in the hierarchy.

The advantage to bottom-up testing is that you interact with the lowest level components in the system early on in the testing phase. For example, you can test the VI that interacts with the data acquisition device early on to make sure that the device works as required. Another advantage is that as you add a VI to the system, you know if that VI caused an error because you are testing a single VI in the integration of the system. A disadvantage to this integration testing method is that you do not test the top-level control and decision algorithms until the end of testing phase. Figure 6-2 illustrates the process of bottom-up testing.

Figure 6-2. Bottom-Up Testing Process



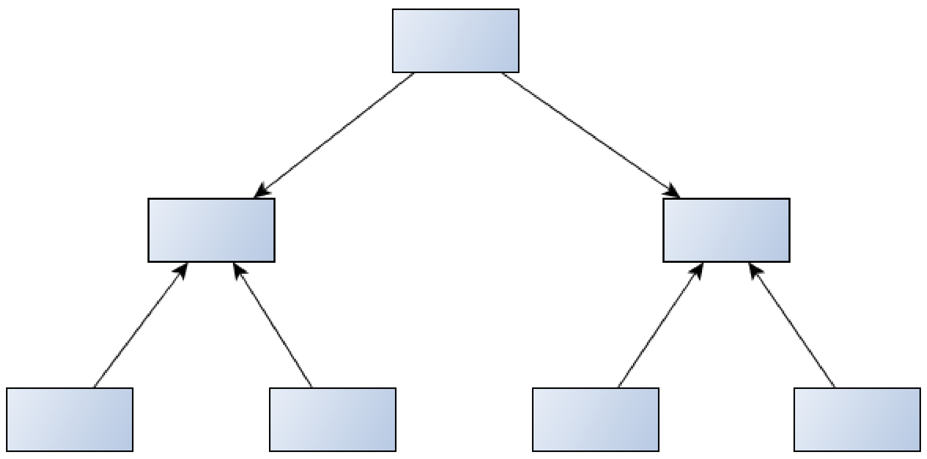
To practice the concepts in this section, complete Exercise 6-3.

To practice the concepts in this section, complete Exercise 6-4.

Sandwich Testing

Sandwich testing is a combination of bottom-up and top-down integration testing. This method uses top-down tests to test the upper-level control. Bottom-up tests are used to test the low-level hardware and system functionality. This method takes advantage of the benefits that both integration testing methods use, while minimizing their disadvantages. Figure 6-3 shows an example of sandwich testing.

Figure 6-3. Sandwich Testing Process



To practice the concepts in this section, complete Exercise 6-5.

Self-Review: Quiz

1. Match each integration testing method to its primary benefit.

Top-down testing	a. Verify functionality of low-level components early
Bottom-up testing	b. Discover problems introduced to existing code by new components
Regression testing	c. Identify major control problems early

Self-Review: Quiz Answers

1. Match each integration testing method to its primary benefit.

Top-down testing	**c. Identify major control problems early**
Bottom-up testing	**a. Verify functionality of low-level components early**
Regression testing	**b. Discover problems introduced to existing code by new components**

Notes

A

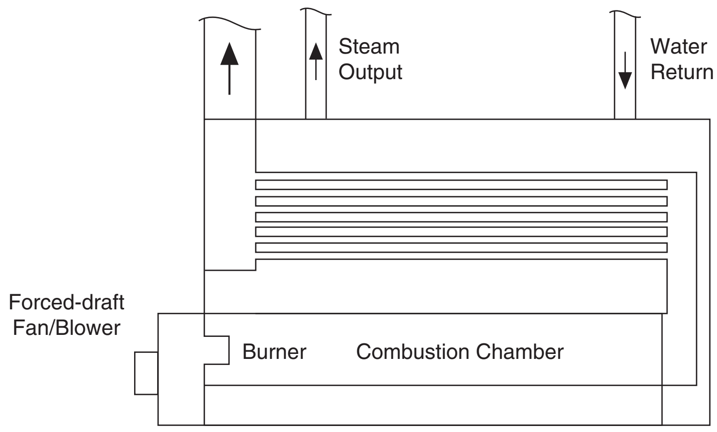
Boiler Controller Requirements

A. Boiler Fundamentals

A boiler is a closed vessel in which a liquid is heated and changed into a vapor. Most boilers heat water into steam by applying heat resulting from the combustion of fuel (for example, natural gas), electrical resistance, or the recovery and conversion of normally unused energy. The steam is used to heat buildings or for various processes, such as running a steam turbine, refining oil, or drying paper.

There are two basic types of boilers-watertube, a boiler in which tubes contain water and steam, with heat being applied to the outside surface; and firetube, a boiler with straight tubes, which are surrounded by water and steam and through which hot gases from combustion pass.

Figure A-1. Fire Tube Boiler



A boiler is almost always part of a closed loop system, whereby the boiler receives feed water that consists of varying proportions of recovered condensed water (return water) and fresh/purified water. The steam, which escapes from the boiler, consists mainly of pure water vapor.

In the case of a gas-burning boiler (for example, natural gas or propane), heat energy is applied to the process by burning fuel. A positive pressure line and control valve supply fuel to the boiler. A pilot flame controls the flow of fuel through the valve. A separate control valve supplies fuel for the pilot flame. A thermocouple provides a positive indication of an established pilot flame (proving the pilot). When the pilot flame is established, the main fuel valve may be opened and the burner ignited as fuel comes in contact with the pilot.

In order to obtain combustion, air must be available to the combustion chamber. To ensure that sufficient airflow (draft) is provided to the combustion chamber, a forced-draft air fan or blower may be used. This forced-draft fan/blower is often integrated with the burner assembly.

Small boilers often cycle on and off to maintain steam pressure. When the burner starts, it goes through a purge cycle where the primary fan blows fresh air through the boiler and stack to blow out any combustible gasses that may have accumulated. When the burner shuts off, there is a similar post purge cycle. Both the pre- and the post-purge cycles cool down the inside of the boiler somewhat, but they are necessary for safety reasons.

Modern boilers may have several built-in safety devices to ensure safe operation. One example of a safety device is the pilot assembly itself. If, for any reason, fuel is not available to the pilot, the main fuel valve closes and fuel cannot flow to the burner. This ensures that an unimpeded flow of unburned fuel cannot occur. In addition to the prove pilot mechanism, a boiler may include sensors to measure fuel flow, airflow, steam temperature, steam pressure, and so on. A boiler controller may provide a “safety interlock” mechanism, which ensures the boiler is in a safe state before beginning the startup sequence.

B. General Application Requirements

The application should fulfill the following requirements.

- Function as specified in Section III: Application Requirements of this document.
- Conform to LabVIEW coding style and documentation standards found in LabVIEW documentation. Refer to the *Development Guidelines* section of the *LabVIEW Help*.
- Be hierarchical in nature. All major functions should be performed in subVIs.
- Be easily scalable to add more states or features without having to manually update the hierarchy.
- Minimize the use of excessive structures, local or global variables, and Property Nodes.
- Respond to front panel controls within 100 ms and not use 100% of CPU time.
- Close all opened references and handles where used.
- Be well documented and include the following documentation features.
  - Labels on appropriate wires within the main VI and subVIs
  - Descriptions for each algorithm
  - Documentation in VI Properties»Documentation for both main VI and subVIs
  - Tip strips and descriptions for front panel controls and indicators
  - Labels for constants

C. Application Requirements

Definitions

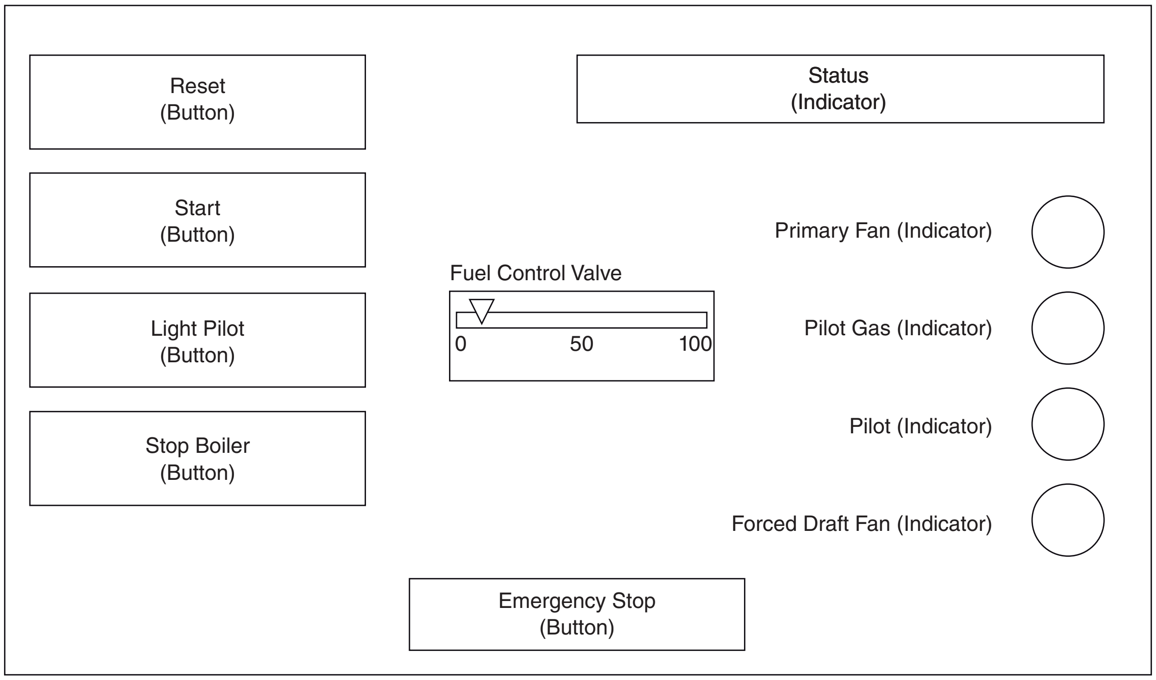
This section defines the terminology for the project.

- Forced draft fan—Provides fresh air to the pilot flame for combustion while the boiler is running. A sensor on the boiler indicates when the forced draft fan is on.
- Fuel control valve—Controls the flow of fuel to the boiler while it is running. This control has a maximum and minimum allowed value while the boiler is running.
- Fuel level—Sensor on the boiler that indicates the flow of fuel to the boiler as a percentage of its maximum.
- Pilot decrement—Time (in milliseconds) for the pilot flame level to decrease by 1% while the boiler is shutting down.
- Pilot flame— Ignites fuel to start combustion. A sensor on the boiler indicates when the pilot light has been ignited.
- Pilot flame level—Sensor on the boiler that indicates the size of the pilot flame as a percentage of its maximum.
- Pilot gas valve—Supplies natural gas as fuel to ignite the pilot flame. A sensor on the boiler indicates when the pilot gas valve is open.
- Pilot increment— Time (in milliseconds) for the pilot flame level to increase by 1% while it is being proved.
- Primary Fan—Generates air flow during purge cycles. A sensor on the boiler that indicates when the primary fan is on.
- Proving the pilot—Process for ensuring that the pilot flame level reaches a pre-defined safety threshold before turning on the forced draft fan and starting the flow of fuel to the boiler.
- Purge cycle—A safety process for removing any combustible gasses that may have accumulated in the combustion chamber. The purge cycle typically runs before starting the boiler and during shutdown.
- Purge time—Duration (in seconds) of the purge cycle. During this time, the primary fan remains on.
- Run interlock—A safety mechanism that ensures the boiler is in a safe state before beginning the startup sequence. A sensor on the boiler indicates when the run interlock requirements have been satisfied.

Task

Develop a boiler controller using LabVIEW. The front panel of the controller should look similar to the front panel shown in Figure A-2.

Figure A-2. Boiler Controller Front Panel Example

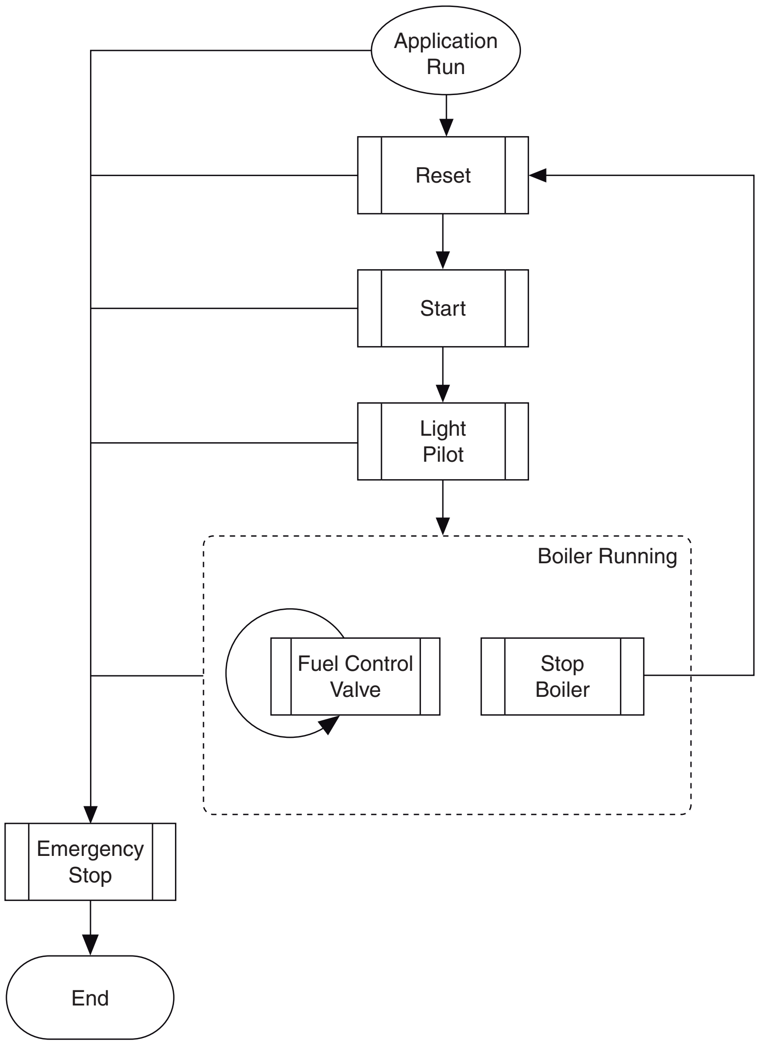


General Operation

The boiler controller allows a user to start up and shut down a boiler. The user interacts with controls on the front panel to start up and shut down the boiler and simulate conditions in the system. Indicators on the front panel display the status and the current step in the startup and shutdown process. The controller also logs events as they occur during the process.
Sequence of Operation

This section describes the sequence the operator follows to operate the boiler controller, as shown in Figure A-3.

Figure A-3. Boiler Sequence of Operations



Application Run

When the application starts, the Status string indicates Lockout and all indicator LEDs are OFF. Initialize controls as indicated in Table A-1

Table A-1. Initial Values for Boiler Controls

Control	Initialize Value
Fuel Control Valve	**`0`**
Fuel Control Valve Minimum	**`10`**
Fuel Control Valve Maximum	**`75`**
Status	**`Lockout`**
All Boolean LEDs	OFF

The **Reset** and **Emergency Stop** buttons and all indicators are enabled. All other controls are disabled to ensure that the operator starts the boiler properly.

The application reads configuration data from a user-specified INI file. These values initialize the boiler controller and the boiler. You can use different INI files for different boilers.

When initialization completes, log the following data to the status log file:

- Time string: Absolute date and time at event
- Event string: **`Boiler Initialized`**
- Event data string: — (There is no relevant data to log for this process)

Note Refer to the *File Specifications* section for file format and update policies for the status log file and the INI file.

Reset

Click the **Reset** button to activate the run interlock on the boiler to ensure that the boiler is in a safe state before continuing through start-up.

When the run interlock completes, the **Status** string indicates **`Ready`**. Log the following data to the status log file.

- Time string: Absolute date and time at event
- Event string: **`Boiler Ready`**
- Event data string: — (There is no relevant data to log for this process)

Enable the **Start** button and disable the **Reset** button to ensure that the operator starts the boiler properly.

Start

Click the **Start** button to execute the pre-purge process to clear combustible gasses from the combustion chamber prior to starting the boiler for safety reasons.

When this step begins, log the following data to the status log file.

- Time string: Absolute date and time at event
- Event string: **`Start Pre-Purge`**
- Event data string: **`0`**

Disable the **Start** button so that the operator does not click it a second time during the purge cycle.

During pre-purge, the Primary Fan turns on. This step lasts for the purge time specified in the INI file. During this time, the Status string indicates Pre-Purge.

After the pre-purge completes, the **Status** string indicates **`Pre-Purge Complete`** and the Primary Fan turns off. Log the following data to the status log file.

- Time string: Absolute date and time at event
- Event string: Pre-Purge Complete
- Event data string: The purge time value, as specified by the INI file.

Enable the **Light Pilot** button to ensure that the operator starts the boiler properly.

Light Pilot

Click the **Light Pilot** button to ignite the pilot, prove the pilot, and start the boiler.

Disable the Light Pilot button to ensure that the operator starts the boiler properly.

Ignition

This step opens the pilot gas valve (turn on the **Pilot Gas Valve** LED) to start the flow of natural gas to the pilot and simultaneously creates a spark to ignite the pilot (turn on the **Pilot** LED).

When the pilot ignites, change the **Status** string to indicate **`Pilot On`** and log the following data to the status log file.

- Time string: Absolute date and time at event
- Event string: **`Pilot ON`**
- Event data string: **`True`**

Prove the Pilot

This step ensures that the pilot flame reaches an adequate level before opening the fuel control valve to start the flow of natural gas to the boiler.

When the **Pilot Flame Level** on the boiler is greater than the flame threshold value read from the INI file, the **Status** string indicates **`Boiler Ready`**. Log the following data to the status log file.

- Time string: Absolute date and time at event
- Event string: `Pilot Proved`
- Event data string: The flame threshold value, as specified by the INI file

Start the Boiler

This step starts the flow of air and natural gas to the combustion chamber and turns off the pilot flame.

Start the flow of air to the combustion chamber by turning on the forced draft fan (turn on the **Forced Draft Fan** LED).

Start the flow of natural gas to the combustion chamber by opening the fuel control valve (enable the **Fuel Control Valve**). While the boiler is running, the fuel control valve should remain between the maximum/minimum values specified in the INI file.

When the boiler is running, the **Status** string indicates **`Boiler Running`**. Enable the **Stop Boiler** button so that the operator can safely shut down the boiler during normal operation. Log the following data to the status log file.

- Time string: Absolute date and time at event
- Event string: **`Forced Draft Fan ON`**
- Event data string: **`True`**

Now that the boiler is running, close the pilot gas valve (turn off the **Pilot Gas Valve** LED), which turns off the pilot flame (turn off the **Pilot** LED). The boiler now begins monitoring its **Simulate Failure** button. Log the following data to the status log file.

- Time string: Absolute date and time at event
- Event string: **`Boiler Running`**
- Event data string: Value of Fuel Control Valve Position

Stop Boiler

Stop the flow of natural gas to the combustion chamber to stop combustion and initiate a purge cycle to clear any remaining combustible gasses from the combustion chamber.

Either of the following conditions can shut down the boiler when it is running.

- Click the **Stop Boiler** button.
- Click the **Simulate Failure** button on the Boiler.

Set the Status string to **`Shutdown`**. Close the fuel control valve (set the value of **Fuel Control Valve** to zero and disable it) to stop the flow of natural gas to the combustion chamber. This results in the **Pilot Flame Level** of the boiler decreasing to zero. Turn off the forced draft fan (turn off the **Forced Draft Fan** LED).

When combustion stops, initiate a purge cycle. As with the pre-purge cycle, the **Primary Fan** turns on. This step lasts for the purge time specified in the INI file. During this time, the **Status** string indicates **`Purge`**. Log the following data to the status log file.

- Time string: Absolute date and time at event
- Event string: **`Start Shutdown Purge`**
- Event data string: **`0`**

After the purge cycle completes, the **Status** string indicates **`Lockout`**. The **Primary Fan** and **Run Interlock** LEDs turn OFF.

Log the following data to the status log file.

- Time string: Absolute date and time at event
- Event string: **`Shutdown Purge Complete`**
- Event data string: **`Purge elapsed time`**

Emergency Stop

This button shuts down the application completely. Click this button to halt execution of all parallel loops, close all open references, and exit.

Handle Errors

Depending on the type of error, the system should respond differently.

- Critical errors—Result from problems that prevent the application from executing safely. These errors halt the application in the same manner as the Emergency Stop button. Log critical errors to disk.

Note Refer to the *File Specifications* section for file format and update policies.

- Non-critical errors—Glitches or minor problems that do not affect safety and do not prevent normal execution. Handle these errors locally instead of shutting down the system.

Description of Controls and Indicators

Table A-2. Description of Controls

Control Name	Description	Function
Reset	Button	Resets Boiler
Start	Button	Initiates the pre-purge step
Light Pilot	Button	Initiates the ignition step
Fuel Control Valve	Numeric	Controls the amount of fuel flowing into the boiler
Stop Boiler	Button	Initiates the boiler shutdown
Emergency Stop	Button	Stops the application within the specified time

Table A-3. Description of Indicators

Indicator Name	Description	Function
Status	String	Indicates boiler status (current step)
Primary Fan	LED	Indicates ON/OFF status of the primary fan

Table A-3. Description of Indicators (Continued)

Indicator Name	Description	Function
Pilot Gas Valve	LED	Indicates ON/OFF status of the natural gas valve
Pilot	LED	Indicates ON/OFF status of the pilot
Forced Draft Fan	LED	Indicates ON/OFF status of the forced draft fan

File Specifications

This section describes the three files the boiler controller uses to initialize configuration data, log status information, and log error information.

INI File Specification

- File Name: `Boiler Init.ini`
- File Location: Relative—same location as the main VI
- Format: Configuration file
  - Sections—Message Handling Loop (MHL), Controller, Boiler
  - Keys:
    - MHL
      - Fuel Control Valve Maximum
      - Fuel Control Valve Minimum
    - Controller
      - Purge Time
      - Flame Threshold
    - Boiler
      - Pilot Increment (ms)
      - Pilot Decrement (ms)
- File Creation and Modification: Create the file and add configuration data as needed for the application.

Status Log File Specification

- File Name: `Status Log.txt`
- File Location: Relative - same location as the main VI
- Format: Tab-delimited text file
- File Header: Timestamp, Event, Event Data
- Event Data string format:
  - Absolute date and time string
  - Event string
  - Event data string
- File Creation and Modification: If file does not exist, create new, write header and log status data. If file exists, append status data to the end of the file.

Error Log File Specification

- File Name: `Error Log.txt`
- File Location: Relative - same location as the main VI
- Format: Tab-delimited text file
- Error Data string format:
  - Absolute date and time string
  - Error Code
  - Error Source (the full call chain information is not needed)
- File Creation and Modification: If file does not exist, create new and log error data. If file exists, append error data to the end of the file.

B

Boiler Controller User Stories

Refer to the following user stories when you develop the *LabVIEW Core 3* Boiler Controller course project.

- As the boiler operator, I want the application to be flexible enough that I can replace the boiler in my system or request that the application include additional boiler features, so that these updates occur with a relatively quick turnaround and minimal down-time.
- As a boiler operator, I want the user interface to remain responsive while boiler operations occur, so that I can always shut down the boiler in the event of an emergency.
- As a boiler operator, I want to ensure that when I shut down the system, the boiler controller does not suspend execution until the boiler finishes shutting down, so that the boiler controller is always aware of the current state of the boiler.
- As a boiler operator, I want to test the functionality of the application by running it with code that simulates the behavior of a boiler, so that I can verify correct operation before I implement the system.
- As a boiler operator, I want the system to include sufficient user documentation, so that new users can easily learn to operate the boiler controller.
- As a boiler operator, I want to control the boiler start-up process from the user interface and observe the current state of the boiler, so that I can readily identify if the boiler behaves correctly at each step.
- As a boiler operator, I want to specify the configuration settings that the boiler controller uses, so that I can reuse the controller for boilers with different configuration settings.
- As a boiler operator, I want the boiler controller to prevent the boiler from entering an unsafe state as a result of a critical error.
- As a boiler operator, I want the system to be robust enough that minor errors do not result in system shut down, so that the boiler only shuts down if the error prevents safe operation.
- As a boiler operator, I want information about critical errors to be logged to disk, so that I can identify and troubleshoot the cause of the error at a later date.
- As a boiler operator, I want to click a button to reset the boiler to a known good state, so that the run interlock (safety control) on the boiler indicates that conditions have been satisfied for the boiler to run safely.
- As a boiler operator, I want the user interface to let me modify only the controls that I am allowed to use at that stage of boiler operation so that I do not accidentally execute any start-up steps out of sequence.
- As a boiler operator, I want the boiler controller to write status information when the boiler transitions between operating states, so that I can review the state information at a later date if an error occurs.

- As a boiler operator, I want to click a button to execute the pre-purge process, which runs the primary fan for a pre-set period of time to purge gases from the combustion chamber prior to lighting the pilot, so that the system does not explode.
- As a boiler operator, I want to click a button to start the flow of gas, ignite the pilot, prove the pilot flame, and start the boiler so that these steps always occur in the proper order to ensure safe operation.
- As a boiler operator, I want to adjust the rate at which fuel enters the boiler furnace, so that I can adjust the temperature of the boiler and the fuel level is always within safe levels.
- As a boiler operator, when the boiler is running, I want to click a button to shut down the boiler without exiting the system, so that the system always follows the boiler shutdown procedure and the boiler safely shuts down.

C

IEEE Requirements Documents

This appendix describes the IEEE standards for software engineers.

Topics

A. Institute of Electrical and Electronic Engineers (IEEE) Standards

A. Institute of Electrical and Electronic Engineers (IEEE) Standards

IEEE defined a number of standards for software engineering. IEEE Standard 730, first published in 1980, is a standard for software quality assurance plans. This standard serves as a foundation for several other IEEE standards and gives a brief description of the minimum requirements for a quality plan in the following areas:

- Purpose
- Reference documents
- Management
- Documentation
- Standards, practices, conventions, and metrics
- Reviews and audits
- Test
- Problem reporting and corrective action
- Tools, techniques, and methodologies
- Code control
- Media control
- Supplier control
- Records collection, maintenance, and retention
- Training
- Risk management

As with the ISO standards, IEEE 730 is fairly short. It does not dictate how to meet the requirements but requires documentation for these practices to a specified minimum level of detail.

In addition to IEEE 730, several other IEEE standards related to software engineering exist, including the following:

- IEEE 610—Defines standard software engineering terminology.
- IEEE 829—Establishes standards for software test documentation.
- IEEE 830—Explains the content of good software requirements specifications.
- IEEE 1074—Describes the activities performed as part of a software lifecycle without requiring a specific lifecycle model.
- IEEE 1298—Details the components of a software quality management system; similar to ISO 9001.

Your projects may be required to meet some or all these standards. Even if you are not required to develop to any of these specifications, they can be helpful in developing your own requirements, specifications, and quality plans.

D

Additional Information and Resources

This appendix contains additional information about National Instruments technical support options and LabVIEW resources.

National Instruments Technical Support Options

Log in to your National Instruments `ni.com` User Profile to get personalized access to your services. Visit the following sections of `ni.com` for technical support and professional services:

- **Support**—Technical support at `ni.com/support` includes the following resources:
  - **Self-Help Technical Resources**—For answers and solutions, visit `ni.com/support` 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. Registered users also receive access to the NI Discussion Forums at `ni.com/forums`. NI Applications Engineers make sure every question submitted online receives an answer.
  - **Standard Service Program Membership**—This program entitles members to direct access to NI Applications Engineers via phone and email for one-to-one technical support, as well as exclusive access to self-paced online training modules at `ni.com/self-paced-training`. All customers automatically receive a one-year membership in the Standard Service Program (SSP) with the purchase of most software products and bundles including NI Developer Suite. NI also offers flexible extended contract options that guarantee your SSP benefits are available without interruption for as long as you need them. Visit `ni.com/ssp` for more information.

    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

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. Visit `zone.ni.com/devzone/cda/tut/p/id/5072` for more information.