LabVIEW™ osnove vol.2

Lesson 1
Common Design Techniques

A. Design Patterns
B. Single Loop Design Patterns
C. Multiple Loop Design Patterns
D. Timing a Design Pattern

Lesson 2
Synchronization Techniques

A. Variables
B. Notifiers
C. Queues

Lesson 3
Event Programming

A. Events
B. Event-Driven Programming
C. Caveats and Recommendations
D. Event-Based Design Patterns

Lesson 4
Error Handling

A. Importance of Error Handling
B. Detect and Report Errors
C. Errors and Warnings
D. Ranges of Error Codes
E. Error Handlers

Lesson 5
Controlling the User Interface

A. Property Nodes
B. Invoke Nodes
C. VI Server Architecture
D. Control References

1

Common Design Techniques

You can develop better programs in LabVIEW and in other programming languages if you follow consistent programming techniques. Design patterns represent techniques that have proved themselves useful time and time again. To facilitate development, LabVIEW provides templates for several common design patterns. This lesson discusses two different categories of programming design patterns—single loop and multiple loops.

Single loop design patterns include the simple VI, the general VI, and the state machine.

Multiple loop design patterns include the parallel loop VI, the master/slave, and the producer/consumer.

Understanding the appropriate use of each design pattern helps you create more efficient LabVIEW VIs.

Topics

A. Design Patterns

B. Single Loop Design Patterns

C. Multiple Loop Design Patterns

D. Timing a Design Pattern

A. Design Patterns

Application design patterns represent LabVIEW code implementations and techniques that are solutions to specific problems in software design. Design patterns typically evolve through the efforts of many developers and are fine-tuned for simplicity, maintainability, and readability. Design patterns represent the techniques that have proved themselves useful over time. Furthermore, as a pattern gains acceptance, it becomes easier to recognize—this recognition alone helps you to read and make changes to your code.

LabVIEW includes several built-in VI templates for several standard design patterns that you will learn about in later lessons. To access the design patterns, select **File»New** to display the **New** dialog box. The design patterns are available in the **VI»From Template»Frameworks»Design Patterns** folder. Listed below are the different kinds of design patterns.

- Master/Slave Design Pattern—Use this template to build a master/slave design pattern. The master loop always executes. It notifies one or more slave loops to execute their code. The slave loop or loops continue executing until they complete, then wait for another notification. Contrast this with the producer/consumer pattern in which the consumer loops execute only when they have data in their queue.
- Producer/Consumer Design Pattern (Data)—Use this template to build a producer/consumer design pattern. Use this template when you need to execute a process, such as data analysis, when a data source, such as a triggered acquisition, produces data at an uneven rate and you need to execute the process when the data becomes available.
- Producer/Consumer Design Pattern (Events)—Use this template to build a producer/consumer design pattern with events to produce queue items. Use this design pattern instead of the user interface event handler pattern for user interfaces when you want to execute code asynchronously in response to an event without slowing the user interface responsiveness.
- Queued Message Handler—Use this template to build a queued message handler design pattern, in which each message handling code can queue any number of new messages.
- Standard State Machine—Use this template to build a standard state machine design pattern. Each state executes code and determines which state to transition to. Contrast this design pattern with the user interface event pattern, in which code executes in response to user actions. Contrast this design pattern also with the queued message handler pattern, in which each message handling code can queue any number of messages.

- User Interface Event Handler—Use this template to build a user interface event handler design pattern. Use this pattern for dialog boxes and other user interfaces in which code executes in response to user actions. You also can create and execute user-defined events that the VI can handle the same way as user interface events.

B. Single Loop Design Patterns

You learned to design three different types of design patterns: the simple architecture, the general architecture, and the state machine.

Simple VI Design Patterns

When performing calculations or making quick lab measurements, you do not need a complicated architecture. Your program might consist of a single VI that takes a measurement, performs calculations, and either displays the results or records them to disk. The simple VI design pattern usually does not require a specific start or stop action from the user. The user just clicks the **Run** button. Use this architecture for simple applications or for functional components within larger applications. You can convert these simple VIs into subVIs that you use as building blocks for larger applications.

Figure 1-1 displays the block diagram of the Determine Warnings VI. This VI performs a single task—it determines what warning to output dependent on a set of inputs. You can use this VI as a subVI whenever you must determine the warning level.

Notice that the VI in Figure 1-1 contains no start or stop actions from the user. In this VI all block diagram objects are connected through data flow. You can determine the overall order of operations by following the flow of data. For example, the Not Equal function cannot execute until the Greater Than or Equal function, the Less Than or Equal function, and both Select functions have executed.

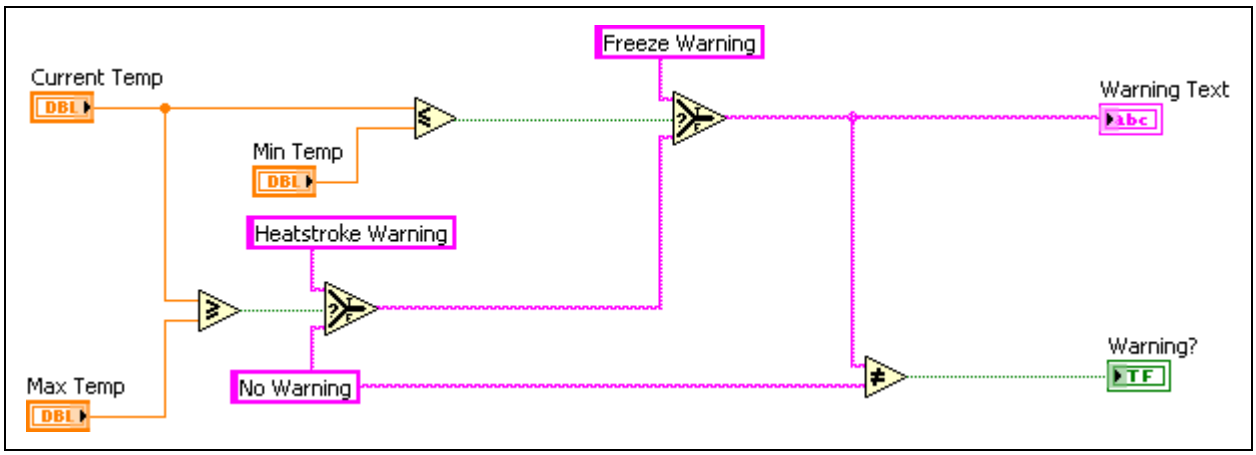


Figure 1-1. Simple VI Architecture

General VI Design Patterns

A general VI design pattern has three main phases—startup, main application, and shutdown. Each of the following phases may contain code that uses another type of design pattern.

- **Startup**—Initializes hardware, reads configuration information from files, or prompts the user for data file locations.
- **Main Application**—Consists of at least one loop that repeats until the user decides to exit the program or the program terminates for other reasons such as I/O completion.
- **Shutdown**—Closes files, writes configuration information to disk, or resets I/O to the default state.

Figure 1-2 shows the general VI design pattern.

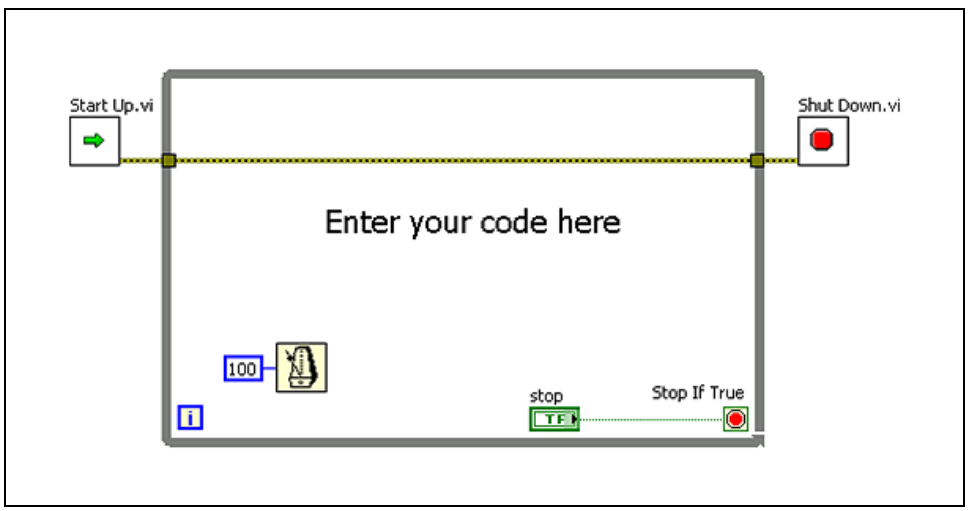


Figure 1-2. General VI Design Pattern

In Figure 1-2, the error cluster wires control the execution order of the three sections. The While Loop does not execute until the Start Up VI finishes running and returns the error cluster data. Consequently, the Shut Down VI cannot run until the main application in the While Loop finishes and the error cluster data leaves the loop.

Tip Most loops require a Wait function, especially if that loop monitors user input on the front panel. Without the Wait function, the loop might run continuously and use all of the computer system resources. The Wait function forces the loop to run asynchronously even if you specify 0 milliseconds as the wait period. If the operations inside the main loop react to user inputs, you can increase the wait period to a level acceptable for reaction times. A wait of 100 to 200 ms is usually good because most users cannot detect that amount of delay between clicking a button on the front panel and the subsequent event execution.

For simple applications, the main application loop is obvious and contains code that uses the simple VI design pattern. When the application incudes complicated user interfaces or multiple tasks such as user actions, I/O triggers, and so on, the main application phase gets more complicated.

State Machine Design Pattern

The state machine design pattern is a modification of the general design pattern. It usually has a start up and shut down phase. However, the main application phase consists of a Case structure embedded in the loop. This architecture allows you to run different code each time the loop executes, depending upon some condition. Each case defines a state of the machine, hence the name, state machine. Use this design pattern for VIs that are easily divided into several simpler tasks, such as VIs that act as a user interface.

A state machine in LabVIEW consists of a While Loop, a Case structure, and a shift register. Each state of the state machine is a separate case in the Case structure. You place VIs and other code that the state should execute within the appropriate case. A shift register stores the state that should execute upon the next iteration of the loop. The block diagram of a state machine VI with five states appears in Figure 1-3. Figure 1-4 shows the other cases, or states, of the state machine.

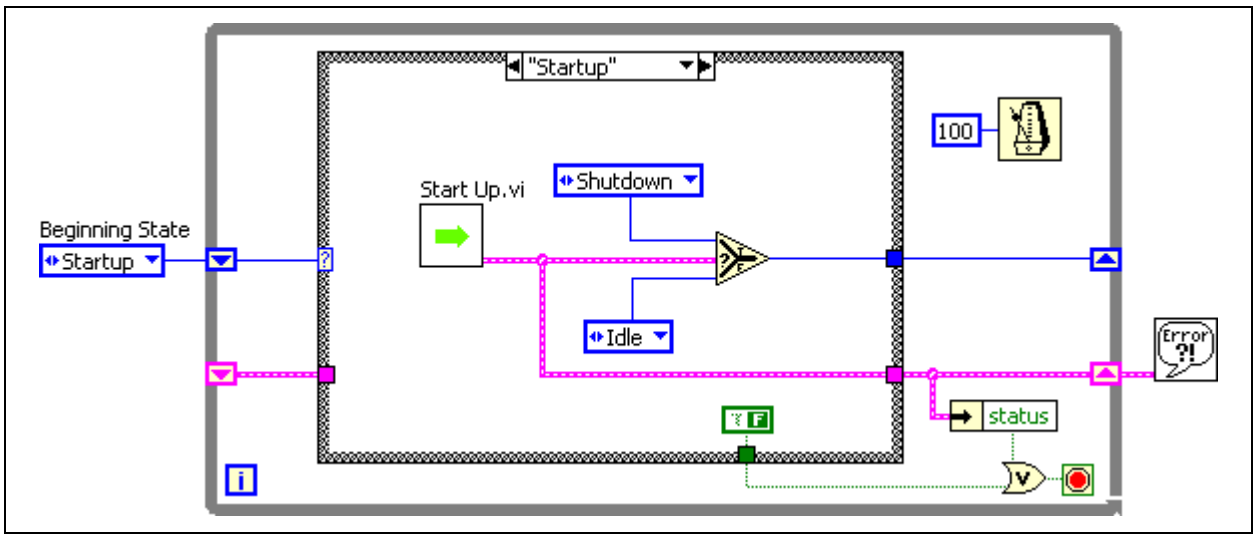


Figure 1-3. State Machine with Startup State

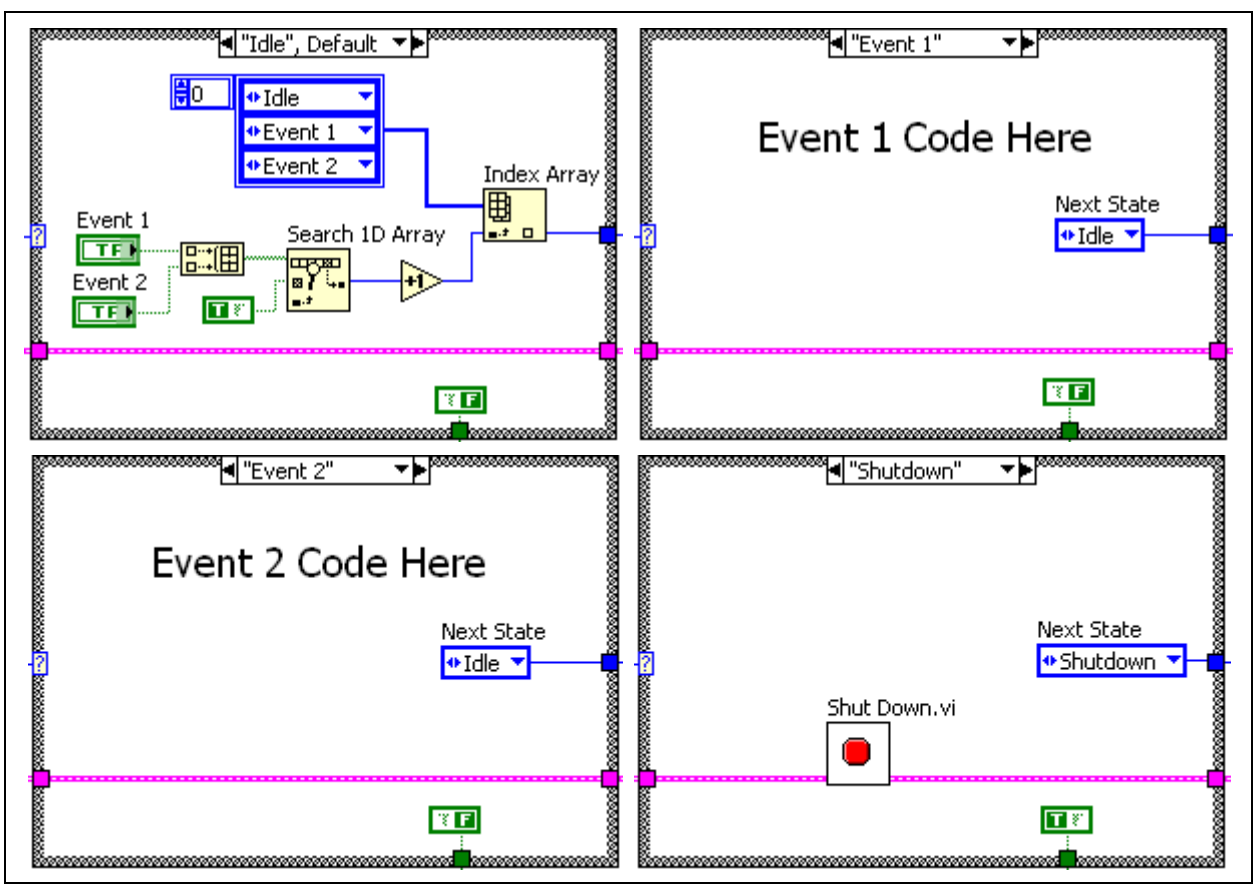


Figure 1-4. Idle (Default), Event 1, Event 2, and Shutdown States

In the state machine design pattern, you design the list of possible tasks, or states, and then map them to each case. For the VI in the previous example, the possible states are Startup, Idle, Event 1, Event 2, and Shutdown. An enumerated constant stores the states. Each state has its own case in the Case

structure. The outcome of one case determines which case to execute next. The shift register stores the value that determines which case to execute next.

The state machine design pattern can make the block diagram much smaller, and therefore, easier to read and debug. Another advantage of the state machine architecture is that each case determines the next state, unlike Sequence structures that must execute every frame in sequence.

A disadvantage of the state machine design pattern is that with the approach in the previous example, it is possible to skip states. If two states in the structure are called at the same time, this model handles only one state, and the other state does not execute. Skipping states can lead to errors that are difficult to debug because they are difficult to reproduce. More complex versions of the state machine design pattern contain extra code that creates a queue of events, or states, so that you do not miss a state. Refer to Lesson 2, *Synchronization Techniques*, for more information about queue-based state machines.

C. Multiple Loop Design Patterns

This section describes the following multiple loop design patterns—parallel loop, master/slave, and producer/consumer data.

Parallel Loop Design Pattern

Some applications must respond to and run several tasks concurrently. One way to improve parallelism is to assign a different loop to each task. For example, you might have a different loop for each button on the front panel and for every other kind of task, such as a menu selection, I/O trigger, and so on. Figure 1-5 shows this parallel loop design pattern.

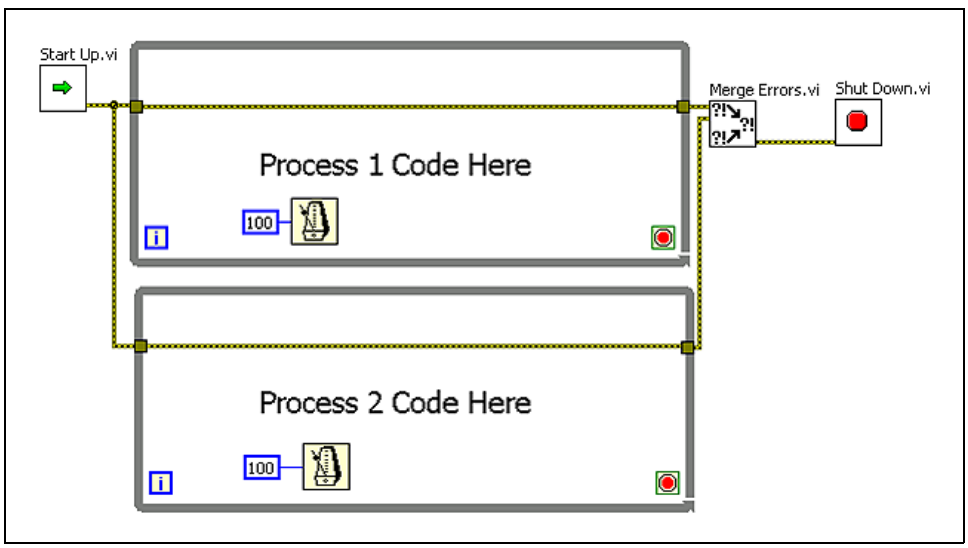


Figure 1-5. Parallel Loop Design Pattern

This structure is straightforward and appropriate for some simple menu VIs, where you expect a user to select from one of several buttons that perform different actions. The parallel loop design pattern lets you handle multiple, simultaneous, independent tasks. In this design pattern, responding to one action does not prevent the VI from responding to another action. For example, if a user clicks a button that displays a dialog box, parallel loops can continue to respond to I/O tasks.

However, the parallel loop design pattern requires you to coordinate and communicate between different loops. You cannot use wires to pass data between loops because doing so prevents the loops from running in parallel. Instead, you must use a messaging technique for passing information among processes. Refer to Appendix A, *Using Variables*, for more information about using local variables to message among parallel loops. Refer to Lesson 2, *Synchronization Techniques*, for messaging techniques using notifiers and queues.

Master/Slave Design Pattern

The master/slave design pattern consists of multiple parallel loops. Each of the loops may execute tasks at different rates. One loop acts as the master, and the other loops act as slaves. The master loop controls all the slave loops and communicates with them using messaging techniques, as shown in Figure 1-6.

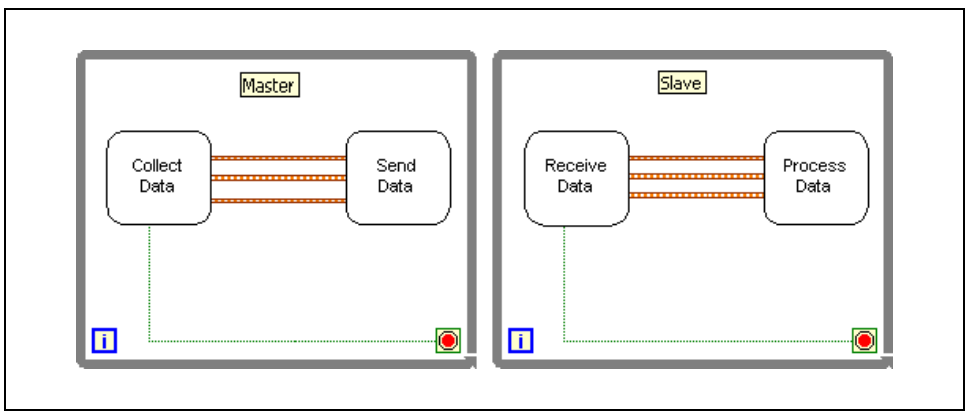


Figure 1-6. Master/Slave Design Pattern

Use the master/slave design pattern when you need a VI to respond to user interface controls while simultaneously collecting data. For example, you want to create a VI that measures and logs a slowly changing voltage once every five seconds. The VI acquires a waveform from a transmission line and displays it on a graph every 100 ms. The VI also provides a user interface that allows the user to change parameters for each acquisition. The master/slave design pattern is well suited for this acquisition application. For this application, the master loop contains the user interface. The voltage

acquisition occurs in one slave loop, while the graphing occurs in another slave loop.

Using the standard master/slave design pattern approach to this VI, you would put the acquisition processes in two separate While Loops, both of them driven by a master loop that receives inputs from the user interface controls. This ensures that the separate acquisition processes do not affect each other, and that any delays caused by the user interface, such as displaying a dialog box, do not delay any iterations of the acquisition processes.

VIs that involve control also benefit from the use of master/slave design patterns. Consider a VI where a user controls a free-motion robotic arm using buttons on a front panel. This type of VI requires efficient, accurate, and responsive control because of the physical damage to the arm or surroundings that might occur if control is mishandled. For example, if the user instructs the arm to stop its downward motion, but the program is occupied with the arm swivel control, the robotic arm might collide with the support platform. Apply the master/slave design pattern to the application to avoid these problems. In this case, the master loop handles the user interface, and each controllable section of the robotic arm has its own slave loop. Because each controllable section of the arm has its own loop and its own piece of processing time, the user interface has more responsive control of the robotic arm.

With a master/slave design pattern, it is important that no two While Loops write to the same shared data. Ensure that no more than one While Loop may write to any given piece of shared data. Refer to Lesson 2, *Synchronization Techniques*, for more information about implementing an application based on the master/slave design pattern.

The slave must not take too long to respond to the master. If the slave is processing a signal from the master and the master sends more than one message to the slave, the slave receives only the latest message. This use of the master/slave architecture could cause a loss of data. Use a master/slave architecture only if you are certain that each slave task takes less time to execute than the master loop.

Producer/Consumer Design Pattern

The producer/consumer design pattern is based on the master/slave design pattern and improves data sharing among multiple loops running at different rates. Similar to the master/slave design pattern, the producer/consumer design pattern separates tasks that produce and consume data at different rates. The parallel loops in the producer/consumer design pattern are separated into two categories—those that produce data and those that consume the data produced. Data queues communicate data among the

loops. The data queues also buffer data among the producer and consumer loops.

Tip A buffer is a memory device that stores temporary data among two devices, or in this case, multiple loops.

Use the producer/consumer design pattern when you must acquire multiple sets of data that must be processed in order. Suppose you want to create a VI that accepts data while processing the data sets in the order they were received. The producer/consumer pattern is ideal for this type of VI because queuing (producing) the data occurs much faster than the data can be processed (consumed). You could put the producer and consumer in the same loop for this application, but the processing queue could not receive additional data until the first piece of data was completely processed. The producer/consumer approach to this VI queues the data in the producer loop and processes the data in the consumer loop, as shown in Figure 1-7.

Tip Queue functions allow you to store a set of data that can be passed among multiple loops running simultaneously or among VIs. Refer to Lesson 2, *Synchronization Techniques*, for more information about queues and implementing applications using the producer/consumer design pattern.

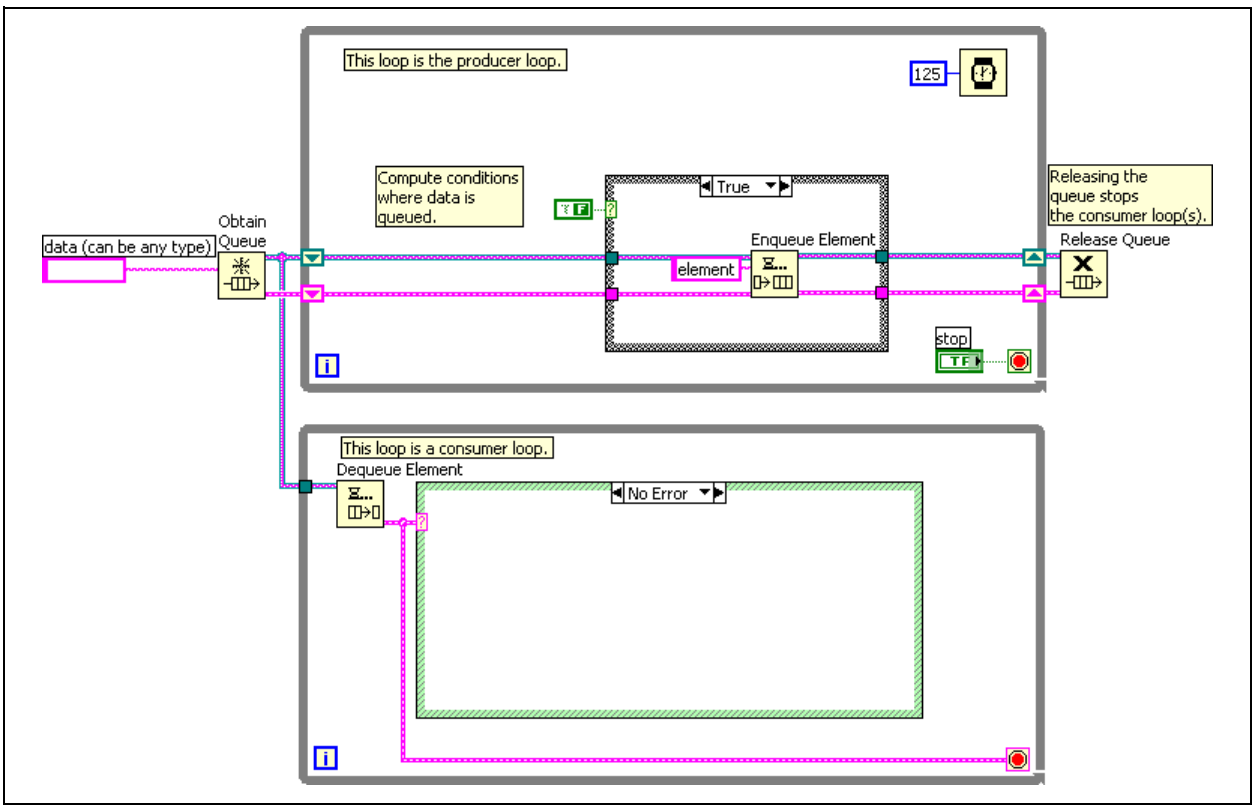


Figure 1-7 Producer/Consumer Design Pattern

This design pattern allows the consumer loop to process the data at its own pace, while the producer loop continues to queue additional data.

You also can use the producer/consumer design pattern to create a VI that analyzes network communication. This type of VI requires two processes to operate at the same time and at different speeds. The first process constantly polls the network line and retrieves packets. The second process analyzes the packets retrieved by the first process.

In this example, the first process acts as the producer because it supplies data to the second process, which acts as the consumer. The producer/consumer design pattern is an effective architecture for this VI. The parallel producer and consumer loops handle the retrieval and analysis of data off the network, and the queued communication between the two loops allows buffering of the network packets retrieved. Buffering can become important when network communication is busy. With buffering, packets can be retrieved and communicated faster than they can be analyzed.

D. Timing a Design Pattern

This section discusses two forms of timing—execution timing and software control timing. Execution timing uses timing functions to give the processor time to complete other tasks. Software control timing involves timing a real-world operation to perform within a set time period.

Execution Timing

Execution timing involves timing a design pattern explicitly or based on events that occur within the VI. Explicit timing uses a function that specifically allows the processor time to complete other tasks, such as the Wait Until Next ms Multiple function. When timing is based on events, the design pattern waits for some action to occur before continuing and allows the processor to complete other tasks while it waits.

Use explicit timing for design patterns such as the master/slave, producer/consumer, and state machine. These design patterns perform some type of polling while they execute.

Tip Polling is the process of making continuous requests for data from another device. In LabVIEW, this generally means that the block diagram continuously asks if there is data available, usually from the user interface.

For example, the master/slave design pattern shown in Figure 1-8 uses a While Loop and a Case structure to implement the master loop. The master executes continuously and polls for an event of some type, such as the user clicking a button. When the event occurs, the master sends a message to the slave. You need to time the master so it does not take over the execution of

the processor. In this case, you typically use the Wait (ms) function to regulate how frequently the master polls.

Tip Always use a timing function such as the Wait (ms) function or the Wait Until Next ms Multiple function in any design pattern that continually executes and needs to be regulated. If you do not use a timing function in a continuously executing structure, LabVIEW uses all the processor time, and background processes may not run.

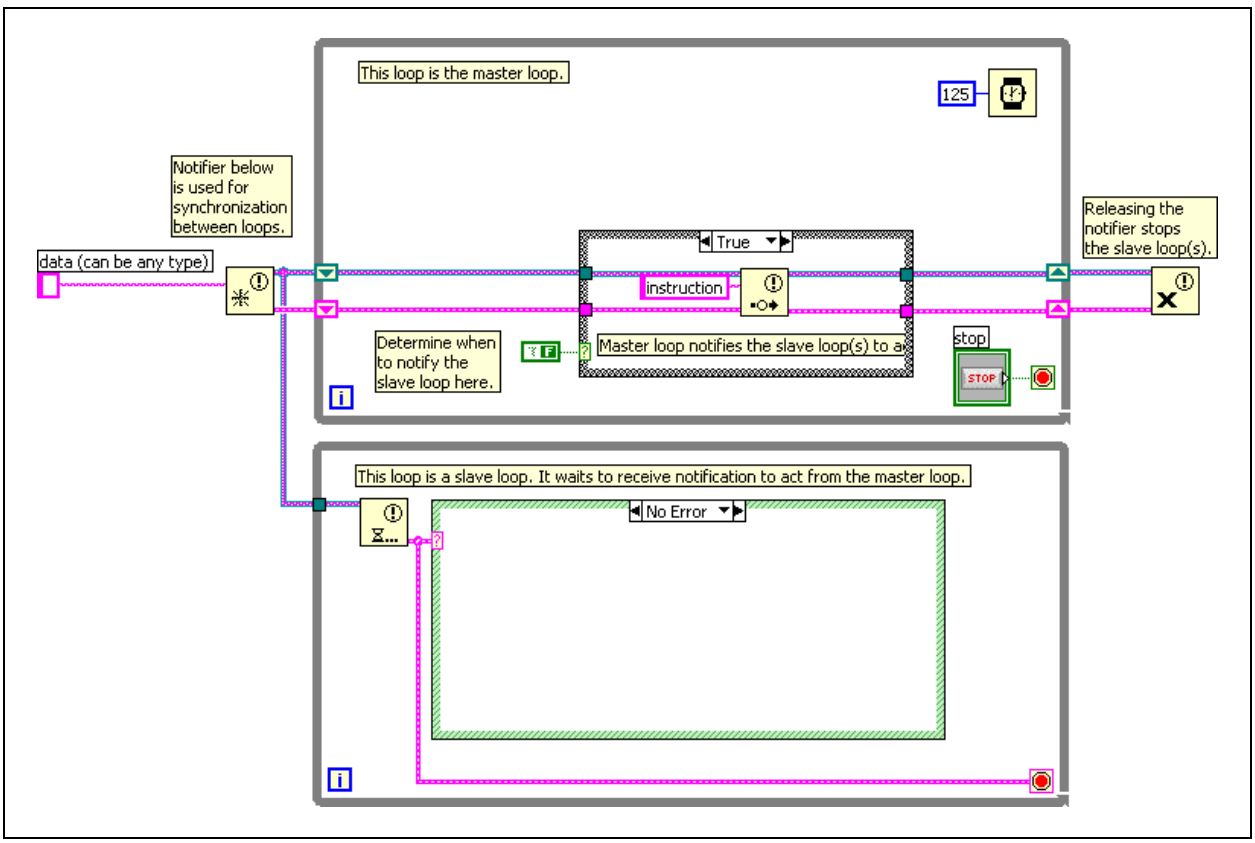


Figure 1-8. Master/Slave Design Pattern

Notice that the slave loop does not contain any form of timing. The use of Synchronization functions, such as queues and notifiers, to pass messages provides an inherent form of timing in the slave loop because the slave loop waits for the Notifier function to receive a message. After the Notifier function receives a message, the slave executes on the message. This creates an efficient block diagram that does not waste processor cycles by polling for messages. This is an example of execution timing by waiting for an event.

When you implement design patterns where the timing is based on the occurrence of events, you do not have to determine the correct timing frequency because the execution of the design pattern occurs only when an

event occurs. In other words, the design pattern executes only when it receives an event.

Software Control Timing

Many applications that you create must execute an operation for a specified amount of time. Consider implementing a state machine design pattern for a temperature data acquisition system. If the specifications require that the system acquire temperature data for 5 minutes, you could remain in the acquisition state for 5 minutes. However, during that time you cannot process any user interface actions such as stopping the VI. To process user interface actions, you must implement timing so that the VI continually executes for specified time. Implementing this type of timing involves keeping the application executing while monitoring a real-time clock.

In vol. 1, you implemented software control timing to monitor the time until the VI should acquire the next piece of data, as shown in Figure 1-9. Notice the use of the Elapsed Time Express VI to keep track of a clock.

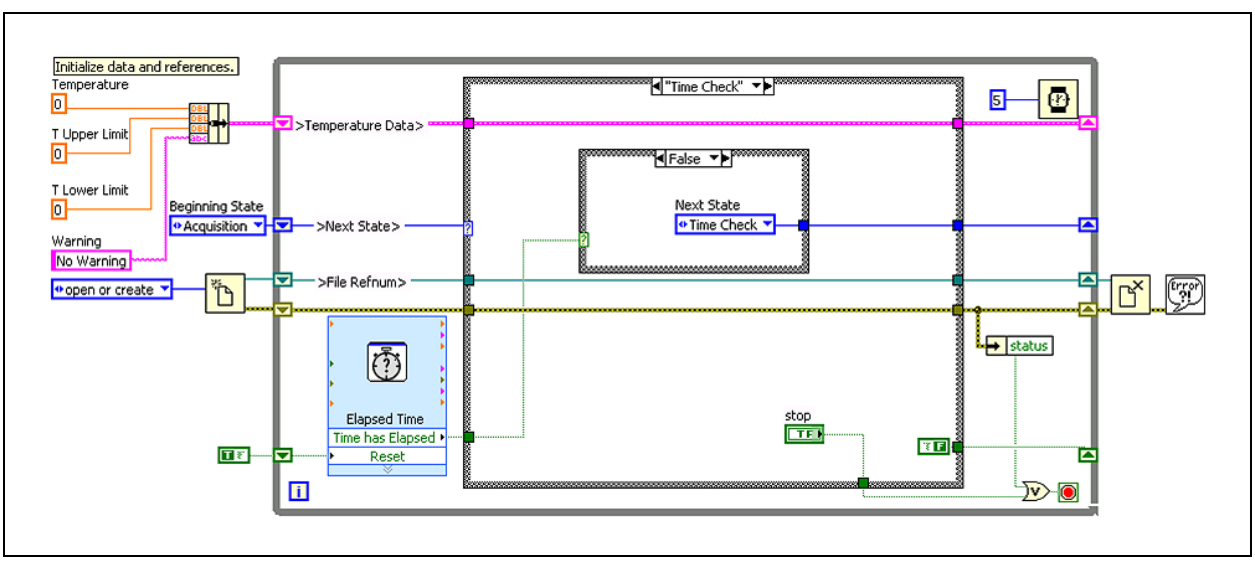


Figure 1-9. Use of the Elapsed Time Express VI

If you use the Wait (ms) function or the Wait Until Next ms Multiple function to perform software timing, the execution of the function you are timing does not occur until the wait function finishes. These timing functions are not the preferred method for performing software control timing, especially for VIs where the system must continually execute. A better method for software control timing utilizes the Get Date/Time In Seconds function to get the current time and track it using shift registers.

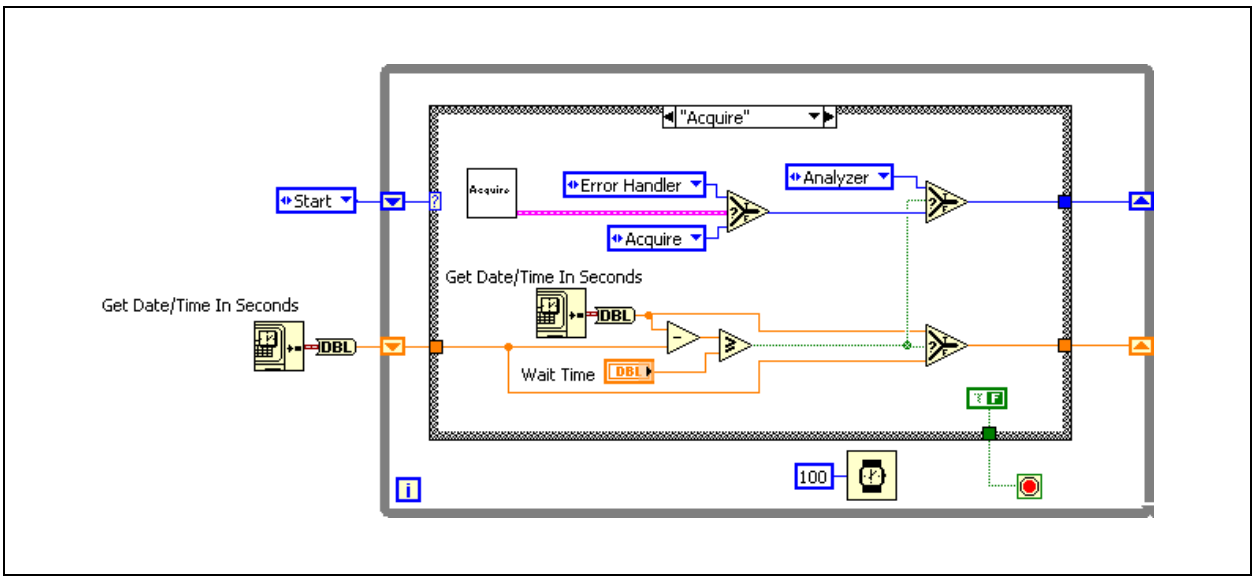


Figure 1-10. Software Timing Using the Get Date/Time In Seconds Function

The Get Date/Time In Seconds function, connected to the left terminal of the shift register, initializes the shift register with the current system time. Each state uses another Get Date/Time In Seconds function and compares the current time to the start time. If the difference in these two times is greater or equal to the wait time, the state finishes executing and the rest of the application executes.

Tip Always use the Get Date/Time In Seconds function instead of the Tick Count function for this type of comparison because the value of the Tick Count function can rollover to 0 during execution.

Refer to Appendix A, *Using Variables*, for more information on creating a timing functional global variable to make the timing functionality modular and reusable.

Self-Review: Quiz

1. The state machine is an example of a design pattern.
   a. True
   b. False

2. Which of the following are reasons for using a multiple loop design pattern?
   a. Execute multiple tasks concurrently
   b. Execute different states in a state machine
   c. Execute tasks at different rates
   d. Execute start up code, main loop, and shutdown code

3. Software control timing allows the processor time to complete other tasks.
   a. True
   b. False

Self-Review: Quiz Answers

1. The state machine is an example of a design pattern.
   a. True
   b. False

2. Which of the following are reasons for using a multiple loop design pattern?
   a. Execute multiple tasks concurrently
   b. Execute different states in a state machine
   c. Execute tasks at different rates
   d. Execute start up code, main loop, and shutdown code

3. Software control timing allows the processor time to complete other tasks.
   a. True
   b. False

Notes

2

Synchronization Techniques

Variables are useful in LabVIEW for passing data between parallel processes. However, when using variables it is often difficult to synchronize data transfers and you must take care to avoid race conditions. This lesson introduces notifiers and queues as alternative methods for passing data between parallel processes. Notifiers and queues have advantages over using variables because of the ability to synchronize the transfer of data.

Topics

A. Variables

B. Notifiers

C. Queues

A. Variables

For parallel loops to communicate, you must use some form of globally available shared data. Using a variable breaks the LabVIEW dataflow paradigm, allows for race conditions, and incurs more overhead than passing the data by wire. Refer to Appendix A, *Using Variables*, for more information about using variables for communicating among multiple loops.

The example shown in Figure 2-1 is a less effective implementation of a master/slave design pattern. This example uses a variable, which causes two problems—there is no timing between the master and the slave, and the variable can cause race conditions. The master cannot signal the slave that data is available, so the slave loop must continually poll the variable to determine if the data changes.

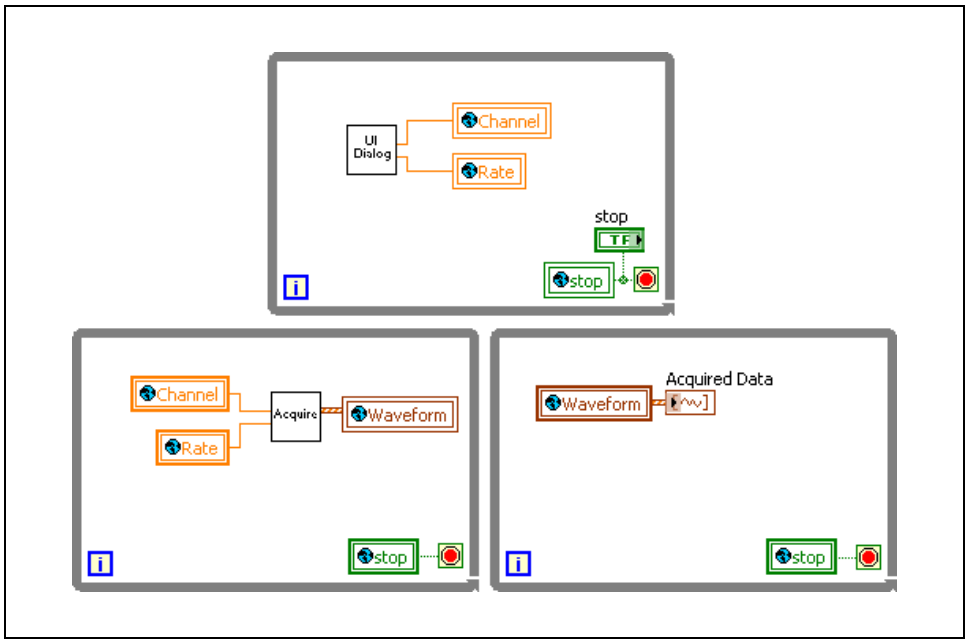


Figure 2-1. Master/Slave Architecture Using Global Variables

B. Notifiers

A more effective implementation of the master/slave design pattern uses notifiers to synchronize data transfer. A notifier sends data along with a notification that the data is available. Using a notifier to pass data from the master to the slave removes any issues with race conditions. Using notifiers also provides a synchronization advantage because the master and slave are timed when data is available, providing for an elegant implementation of the master/slave design pattern. Figure 2-2 shows the master/slave design pattern using notifiers.

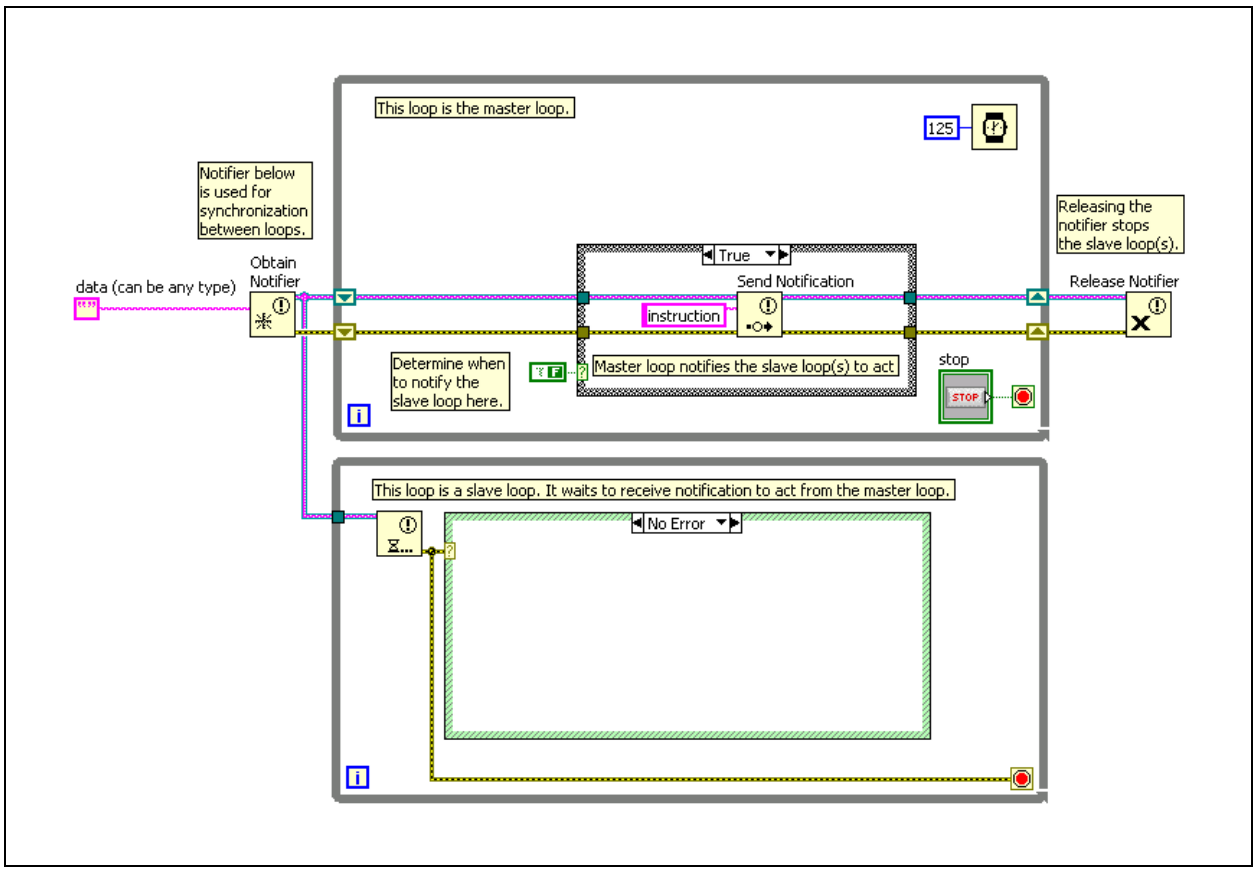


Figure 2-2. Master/Slave Design Pattern Using Notifiers

The notifier is created before the loops begin using the Obtain Notifier function. The master loop uses the Send Notification function to notify the slave loop through the Wait on Notification function. After the VI has finished using the notifiers, the Release Notifier function releases the notifiers.

The following benefits result from using notifiers in the master/slave design pattern:

- Both loops are synchronized to the master loop. The slave loop only executes when the master loop sends a notification.
- You can use notifiers to create globally available data. Thus, you can send data with a notification. For example, in Figure 2-2, the Send Notification function sends the string `instruction`.
- Using notifiers creates efficient code. You need not use polling to determine when data is available from the master loop.

However, using notifiers can have drawbacks. A notifier does not buffer the data. If the master loop sends another piece of data before the slave loop(s) reads the first piece of data, that data is overwritten and lost.

C. Queues

Queues are similar to notifiers, except that a queue can store multiple pieces of data. By default, queues work in a first in, first out (FIFO) manner. Therefore, the first piece of data inserted into the queue is the first piece of data that is removed from the queue. Use a queue when you want to process all data placed in the queue. Use a notifier if you want to process only the current data.

When used, the producer/consumer design pattern, queues pass data and synchronize the loops.

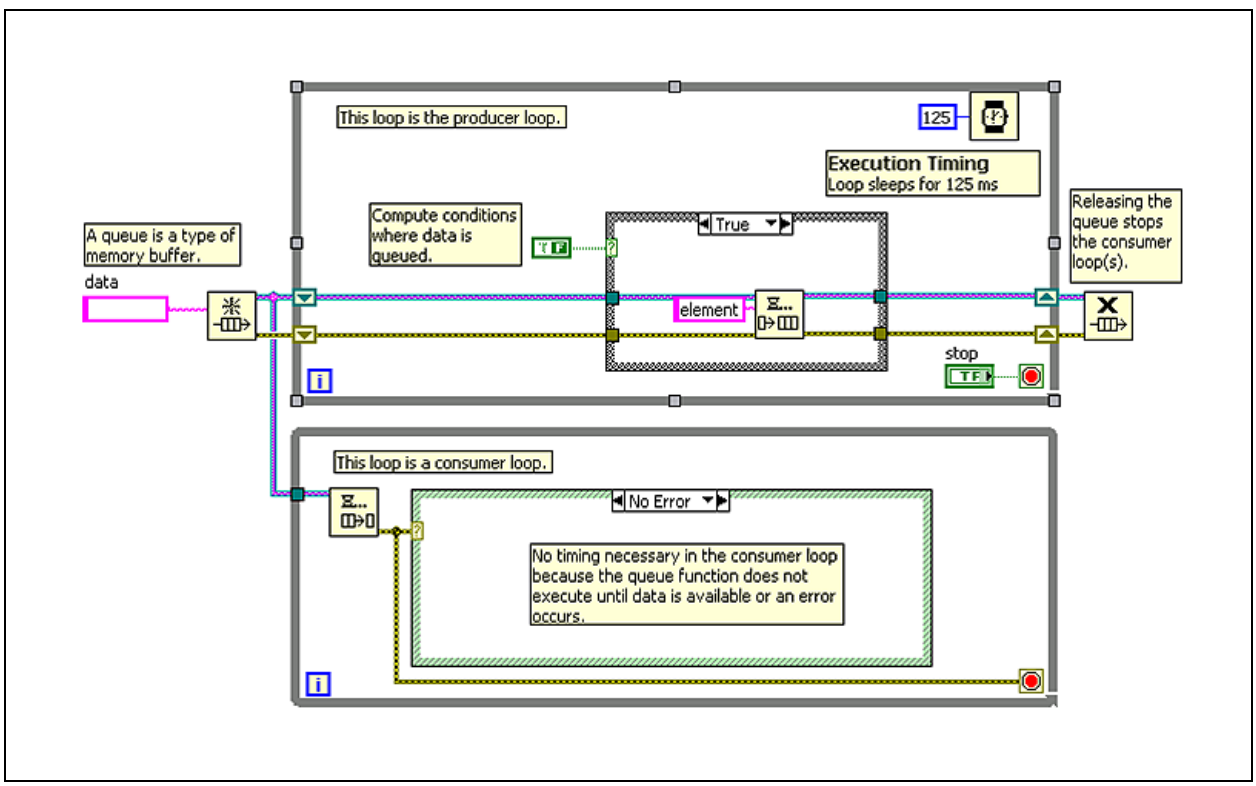


Figure 2-3. Producer/Consumer Design Pattern Using Queues

The queue is created before the loops begin using the Obtain Queue function. The producer loop uses the Enqueue Element function to add data to the queue. The consumer loop removes data from the queue using the Dequeue Element function. The consumer loop does not execute until data is available in the queue. After the VI has finished using the queues, the Release Queue function releases the queues. When the queue releases, the Dequeue Element function generates an error, effectively stopping the consumer loop. This eliminates the need to use a variable to stop the loops.

The following benefits result from using queues in the producer/consumer design pattern:

- Both loops are synchronized to the producer loop. The consumer loop only executes when data is available in the queue.
- You can use queues to create globally available data that is queued, removing the possibility of losing the data in the queue when new data is added to the queue.
- Using queues creates efficient code. You need not use polling to determine when data is available from the producer loop.

Queues are also useful for holding state requests in a state machine. In the implementation of a state machine that you have learned, if two states are requested simultaneously, you might lose one of the state requests. A queue stores the second state request and executes it when the first has finished.

Case Study: Weather Station Project

The weather station project acquires temperature and wind speed data, and analyzes it to determine if the situation requires a warning. If the temperature is too high or too low, it alerts the user to a danger of heatstroke or freezing. It also monitors the wind speed to generate a high wind warning when appropriate.

The block diagram consists of two parallel loops, which are synchronized using queues. One loop acquires data for temperature and wind speed and the other loop analyzes the data. The loops in the block diagram use the producer/consumer design pattern and pass the data through the queue. Queues help process every reading acquired from the DAQ Assistant.

Code for acquiring temperature and wind speed is placed in the producer loop. Code containing the state machine for analysis of temperature-weather conditions is within the no error case of the consumer loop. The code using a queue is more readable and efficient than the code using only state machine architecture. The Obtain Queue function creates the queue reference. The producer loop uses the Enqueue Element function to add data obtained from the DAQ Assistant to the queue. The consumer loop uses the Dequeue Element function to get the data from the queue and provide it to the state machine for analysis. The Release Queue function marks the end of queue by destroying it. The use of queues also eliminates the need for a shared variable to stop the loops because the Dequeue Element function stops the consumer loop when the queue is released.

Figure 2-4 shows the block diagram consisting of a producer and a consumer loop. Data transfer and synchronization between the loops is achieved by the queue functions.

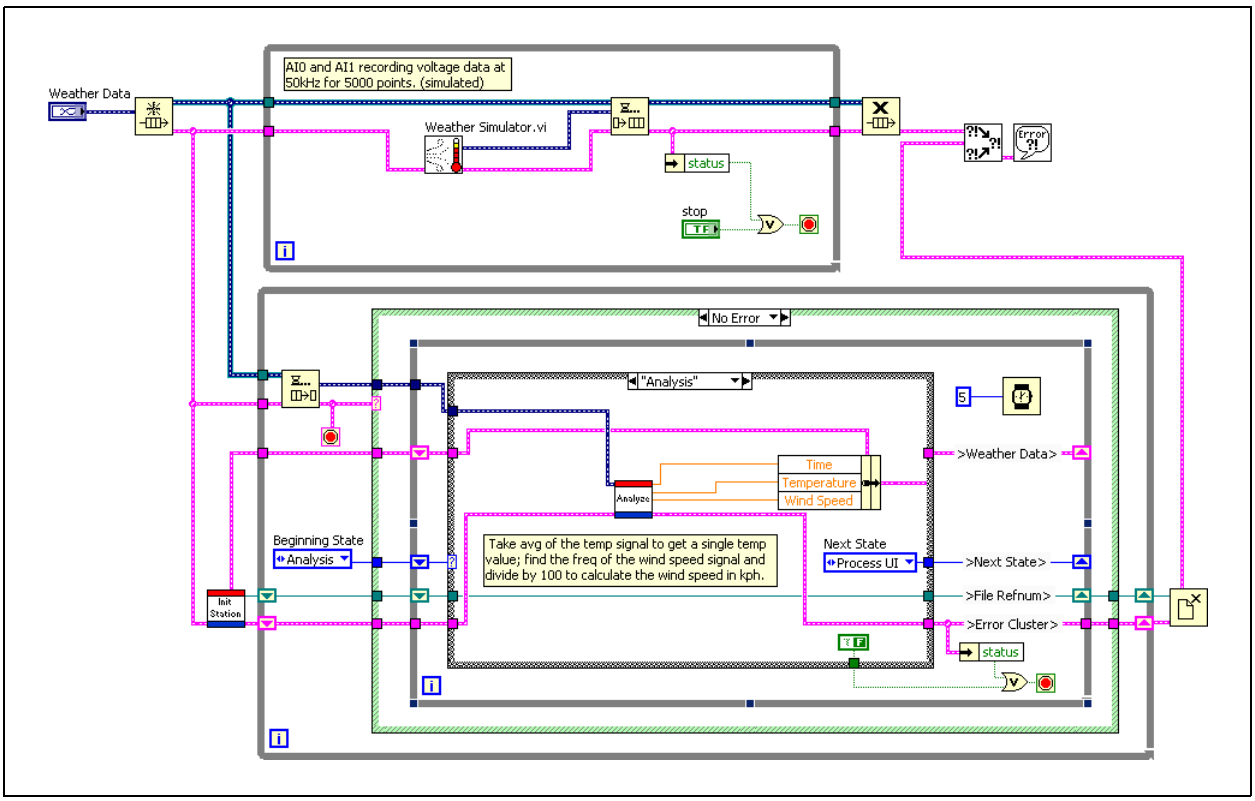


Figure 2-4. Data Transfer and Synchronization of Parallel Loops Using Queues

Self-Review: Quiz

1. Which of the following buffer data?
   a. Notifiers
   b. Queues
   c. Global Variables
   d. Local Variables

2. Match the following:

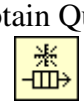
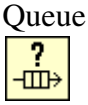
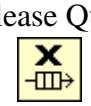
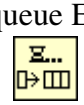
Obtain Queue 	Destroys the queue reference
Get Queue Status 	Assigns the data type of the queue
Release Queue 	Adds an element to the back of the queue
Enqueue Element 	Determines the number of elements currently in the queue

3. Which of the following are valid data types for Queues and Notifiers?
   a. String
   b. Numeric
   c. Enum
   d. Array of Booleans
   e. Cluster of a String and a Numeric

Self-Review: Quiz Answers

1. Which of the following buffer data?
   a. Notifiers
   b. Queues
   c. Global Variables
   d. Local Variables

2. Match the following:

Obtain Queue 	**Assigns the data type of the queue**
Get Queue Status 	**Determines the number of elements currently in the queue**
Release Queue 	**Destroys the queue reference**
Enqueue Element 	**Adds an element to the back of the queue**

3. Which of the following are valid data types for Queues and Notifiers?
   a. String
   b. Numeric
   c. Enum
   d. Array of Booleans
   e. Cluster of a String and a Numeric

Notes

3

Event Programming

Event-based design patterns allow you to create more efficient and flexible applications. Event-based design patterns use the Event structure to respond directly to the user or other events. This lesson describes event-driven programming using the Event structure and design patterns that use the Event structure.

Topics

A. Events

B. Event-Driven Programming

C. Caveats and Recommendations

D. Event-Based Design Patterns

A. Events

LabVIEW is a dataflow programming environment where the flow of data determines the execution order of block diagram elements. Event-driven programming features extend the LabVIEW dataflow environment to allow the user's direct interaction with the front panel and other asynchronous activity to further influence block diagram execution.

Note Event-driven programming features are available only in the LabVIEW Full and Professional Development Systems. You can run a VI built with these features in the LabVIEW Base Package, but you cannot reconfigure the event-handling components.

What Are Events?

An event is an asynchronous notification that something has occurred. Events can originate from the user interface, external I/O, or other parts of the program. User interface events include mouse clicks, key presses, and so on. External I/O events include hardware timers or triggers that signal when data acquisition completes or when an error condition occurs. Other types of events can be generated programmatically and used to communicate with different parts of the program. LabVIEW supports user interface and programmatically generated events. LabVIEW also supports ActiveX and .NET generated events, which are external I/O events.

In an event-driven program, events that occur in the system directly influence the execution flow. In contrast, a procedural program executes in a pre-determined, sequential order. Event-driven programs usually include a loop that waits for an event to occur, executes code to respond to the event, and reiterates to wait for the next event. How the program responds to each event depends on the code written for that specific event. The order in which an event-driven program executes depends on which events occur and on the order in which they occur. Some sections of the program might execute frequently because the events they handle occur frequently, and other sections of the program might not execute at all because the events never occur.

Why Use Events?

Use user interface events in LabVIEW to synchronize user actions on the front panel with block diagram execution. Events allow you to execute a specific event-handling case each time a user performs a specific action. Without events, the block diagram must poll the state of front panel objects in a loop, checking to see if any change has occurred. Polling the front panel requires a significant amount of CPU time and can fail to detect changes if they occur too quickly. By using events to respond to specific user actions, you eliminate the need to poll the front panel to determine which actions the user performed. Instead, LabVIEW actively notifies the block diagram each

time an interaction you specified occurs. Using events reduces the CPU requirements of the program, simplifies the block diagram code, and guarantees that the block diagram can respond to all interactions the user makes.

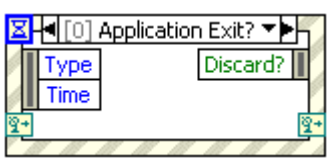
Use programmatically generated events to communicate among different parts of the program that have no dataflow dependency. Programmatically generated events have many of the same advantages as user interface events and can share the same event-handling code, making it easy to implement advanced architectures, such as queued state machines using events.

B. Event-Driven Programming

In Lesson 1, *Common Design Techniques*, you learned how event-driven programming extends the LabVIEW dataflow environment to allow user interaction with the front panel. You also learned about programmatically-generated events allow for easy implementation of advanced architectures, such as queued state machines.

Event Structure Components

Use the Event structure, shown as follows, to handle events in a VI.



The Event structure works like a Case structure with a built-in Wait on Notification function. The Event structure can have multiple cases, each of which is a separate event-handling routine. You can configure each case to handle one or more events, but only one of these events can occur at a time. When the Event structure executes, it waits until one of the configured events occur, then executes the corresponding case for that event. The Event structure completes execution after handling exactly one event. It does not implicitly loop to handle multiple events. Like a Wait on Notification function, the Event structure can time out while waiting for notification of an event. When this occurs, a specific Timeout case executes.

The event selector label at the top of the Event structure indicates which events cause the currently displayed case to execute.

View other event cases by clicking the down arrow next to the case name and selecting another case from the shortcut menu.

The Timeout terminal at the top left corner of the Event structure specifies the number of milliseconds to wait for an event before timing out.

The default is –1, which specifies to wait indefinitely for an event to occur. If you wire a value to the Timeout terminal, you must provide a Timeout case.

The Event Data Node behaves similarly to the Unbundle By Name function.

This node is attached to the inside left border of each event case. The node identifies the data LabVIEW provides when an event occurs. You can resize this node vertically to add more data items, and you can set each data item in the node to access any event data element. The node provides different data elements in each case of the Event structure depending on which event(s) you configure that case to handle. If you configure a single case to handle multiple events, the Event Data Node provides only the event data elements that are common to all the events configured for that case.

The Event Filter Node is similar to the Event Data Node.

This node is attached to the inside right border of filter event cases. The node identifies the subset of data available in the Event Data Node that the event case can modify. The node displays different data depending on which event(s) you configure that case to handle. By default, these items are inplace to the corresponding data items in the Event Data Node. If you do not wire a value to a data item of an Event Filter Node, that data item remains unchanged.

Refer to the *Notify and Filter Events* section of this lesson for more information about filter events.

The dynamic event terminals are available by right-clicking the Event structure and selecting **Show Dynamic Event Terminals** from the shortcut menu.

These terminals are used only for dynamic event registration.

Refer to the *Dynamic Event Registration* topic and the *Modifying Registration Dynamically* topic of the *LabVIEW Help* for more information about using these terminals.

Note Like a Case structure, the Event structure supports tunnels. However, by default you do not have to wire Event structure output tunnels in every case. All unwired tunnels use the default value for the tunnel data type. Right-click a tunnel and deselect **Use Default If Unwired** from the shortcut menu to revert to the default Case structure behavior where tunnels must be wired in all cases. You also can configure the tunnels to wire the input and output tunnels automatically in unwired cases.

Refer to the *LabVIEW Help* for information about the default values for data types.

Notify and Filter Events

Notify events are an indication that a user action has already occurred, such as when the user has changed the value of a control. Use notify events to respond to an event after it has occurred and LabVIEW has processed it. You can configure any number of Event structures to respond to the same notify event on a specific object. When the event occurs, LabVIEW sends a copy of the event to each Event structure configured to handle the event in parallel.

Filter events inform you that the user has performed an action before LabVIEW processes it, which allows you to customize how the program responds to interactions with the user interface. Use filter events to participate in the handling of the event, possibly overriding the default behavior for the event. In an Event structure case for a filter event, you can validate or change the event data before LabVIEW finishes processing it, or you can discard the event entirely to prevent the change from affecting the VI. For example, you can configure an Event structure to discard the Panel Close? event, which prevents the user from interactively closing the front panel of the VI.

Filter events have names that end with a question mark, such as Panel Close?, to help you distinguish them from notify events. Most filter events have an associated notify event of the same name, but without the question mark, which LabVIEW generates after the filter event if no event case discarded the event.

For example, you can use the Mouse Down? and Shortcut Menu Activation? filter events to display a context menu when you left-click a control. To perform this action, modify the data returned by the **Button** event data field of the Mouse Down? filter event. The value of the left mouse button is 1, and the value of the right mouse button is 2. In order to display the context menu when you left-click a control, change the **Button** event data field to 2 so that LabVIEW treats a left-click like a right-click. Refer to the Left-click Shortcut Menu VI in the `labview\examples\general` directory for an example of using filter events.

As with notify events, you can configure any number of Event structures to respond to the same filter event on a specific object. However, LabVIEW sends filter events sequentially to each Event structure configured for the event. The order in which LabVIEW sends the event to each Event structure depends on the order in which the events were registered. Each Event structure must complete its event case for the event before LabVIEW can notify the next Event structure. If an Event structure case changes any of the event data, LabVIEW passes the changed data to subsequent Event structures in the chain. If an Event structure in the chain discards the event, LabVIEW does not pass the event to any Event structures remaining in the

chain. LabVIEW completes processing the user action which triggered the event only after all configured Event structures handle the event without discarding it.

Note National Instruments recommends you use filter events only when you want to take part in the handling of the user action, either by discarding the event or by modifying the event data. If you only want to know that the user performed a particular action, use notify events.

Event structure cases that handle filter events have an Event Filter Node. You can change the event data by wiring new values to these terminals. If you do not wire a value to the data item of the Event Filter Node, the default value equals the value that the corresponding item in the Event Data Node returns. You can completely discard an event by wiring a TRUE value to the Discard? terminal.

Note A single case in the Event structure cannot handle both notify and filter events. A case can handle multiple notify events but can handle multiple filter events only if the event data items are identical for all events.

Refer to the *Using Events in LabVIEW* section of this lesson for more information about event registration.

Tip In the Edit Events dialog box, notify events are signified by a green arrow, and filter events are signified by a red arrow.

Using Events in LabVIEW

LabVIEW can generate many different events. To avoid generating unwanted events, use event registration to specify which events you want LabVIEW to notify you about. LabVIEW supports two models for event registration—static and dynamic.

Static registration allows you to specify which events on the front panel of a VI you want to handle in each Event structure case on the block diagram of that VI. LabVIEW registers these events automatically when the VI runs, so the Event structure begins waiting for events as soon as the VI begins running. Each event is associated with a control on the front panel of the VI, the front panel window of the VI as a whole, or the LabVIEW application. You cannot statically configure an Event structure to handle events for the front panel of a different VI. Configuration is static because you cannot change at run time which events the Event structure handles.

Dynamic event registration avoids the limitations of static registration by integrating event registration with the VI Server, which allows you to use Application, VI, and control references to specify at run time the objects for

which you want to generate events. Dynamic registration provides more flexibility in controlling what events LabVIEW generates and when it generates them. However, dynamic registration is more complex than static registration because it requires using VI Server references with block diagram functions to explicitly register and unregister for events rather than handling registration automatically using the information you configured in the Event structure.

Note In general, LabVIEW generates user interface events only as a result of direct user interaction with the active front panel. LabVIEW does not generate events, such as Value Change, when you use shared variables, global variables, local variables, DataSocket, and so on. However, you can use the Value (Signaling) property to generate a Value Change event programmatically. In many cases, you can use programmatically generated events instead of queues and notifiers.

The event data provided by a LabVIEW event always include a time stamp, an enumeration that indicates which event occurred, and a VI Server reference to the object that triggered the event. The time stamp is a millisecond counter you can use to compute the time elapsed between two events or to determine the order of occurrence. The reference to the object that generated the event is strictly typed to the VI Server class of that object. Events are grouped into classes according to what type of object generates the event, such as Application, VI, or Control. If a single case handles multiple events for objects of different VI Server classes, the reference type is the common parent class of all objects. For example, if you configure a single case in the Event structure to handle events for a numeric control and a color ramp control, the type of the control reference of the event source is Numeric because the numeric and color ramp controls are in the Numeric class. If you register for the same event on both the VI and Control class, LabVIEW generates the VI event first.

Note Clusters are the only container objects for which you can generate events. LabVIEW generates Control events for clusters, before it generates events for the objects they contain, except in the case of the Value Change event. The Value Change event generates the event on an element in the cluster, then on the cluster itself. If the Event structure case for a VI event or for a Control event on a container object discards the event, LabVIEW does not generate further events.

Each Event structure and Register For Events function on the block diagram owns a queue that LabVIEW uses to store events. When an event occurs, LabVIEW places a copy of the event into each queue registered for that event. An Event structure handles all events in its queue and the events in the queues of any Register For Events functions that you wired to the dynamic event terminals of the Event structure. LabVIEW uses these queues

to ensure that events are reliably delivered to each registered Event structure in the order the events occur.

By default, when an event enters a queue, LabVIEW locks the front panel that contains the object that generated that event. LabVIEW keeps the front panel locked until all Event structures finish handling the event. While the front panel is locked, LabVIEW does not process front panel activity but places those interactions in a buffer and handles them when the front panel is unlocked.

For example, a user might anticipate that an event case launches an application that requires text entry. Since the user already knows text entry is needed, he might begin typing before the application appears on the front panel. If the **Lock front panel until the event case for this event completes** option is enabled, once the application launches and appears on the front panel, it processes the key presses in the order in which they occurred. If the **Lock front panel until the event case for this event completes** option is disabled, the key presses might be processed elsewhere on the front panel, since LabVIEW does not queue their execution to depend on the completion of the event case.

Front panel locking does not affect certain actions, such as moving the window, interacting with the scroll bars, and clicking the **Abort** button.

LabVIEW can generate events even when no Event structure is waiting to handle them. Because the Event structure handles only one event each time it executes, place the Event structure in a While Loop to ensure that an Event structure can handle all events that occur.

Caution If no Event structure executes to handle an event and front panel locking is enabled, the user interface of the VI becomes unresponsive. If this occurs, click the **Abort** button to stop the VI. You can disable front panel locking by right-clicking the Event structure and removing the checkmark from the **Lock front panel until the event case for this event completes** checkbox in the **Edit Events** dialog box. You cannot turn off front panel locking for filter events.

Static Event Registration

Static event registration is available only for user interface events. Use the Edit Events dialog box to configure an Event structure to handle a statically registered event. Select the event source, which can be the application, the VI, or an individual control. Select a specific event the event source can generate, such as Panel Resize, Value Change, and so on. Edit the case to handle the event data according to the application requirements.

LabVIEW statically registers events automatically and transparently when you run a VI that contains an Event structure. LabVIEW generates events

for a VI only while that VI is running or when another running VI calls the VI as a subVI.

When you run a VI, LabVIEW sets that top-level VI and the hierarchy of subVIs the VI calls on its block diagram to an execution state called reserved. You cannot edit a VI or click the Run button while the VI is in the reserved state because the VI can be called as a subVI at any time while its parent VI runs. When LabVIEW sets a VI to the reserved state, it automatically registers the events you statically configured in all Event structures on the block diagram of that VI. When the top-level VI finishes running, LabVIEW sets it and its subVI hierarchy to the idle execution state and automatically unregisters the events.

Refer to the `labview\examples\general\uievents.llb` for examples of using static event registration.

Configuring Events

Before you configure events for the Event structure to handle, refer to the *Caveats and Recommendations when Using Events in LabVIEW* topic of the *LabVIEW Help*.

Complete the following steps to configure an Event structure case to handle an event.

1. (Optional) If you want to configure the Event structure to handle a user event, a Boolean control within a radio buttons control, or a user interface event that is generated based on a reference to an application, VI, or control, you first must dynamically register that event. Refer to the *Dynamically Registering Events* topic of the *LabVIEW Help* for more information about using dynamic events.
2. Right-click the border of the Event structure and select **Edit Events Handled by This Case** from the shortcut menu to display the **Edit Events** dialog box to edit the current case. You also can select **Add Event Case** from the shortcut menu to create a new case.
3. Specify an event source in the **Event Sources** pane.
4. Select the event you want to configure for the event source, such as **Key Down**, **Timeout**, or **Value Change** from the **Events** list. When you select a dynamic event source from the **Event Sources** list, the **Events** list displays that event. This is the same event you selected when you registered the event. If you have registered for events dynamically and wired **event reg refnum out** to the dynamic event terminal, the sources appear in the **Dynamic** section.
5. If you want to add additional events for the current case to handle, click the + button and repeat steps 3 and 4 to specify each additional event. The **Event Specifiers** section at the top of the dialog box lists all the

events for the case to handle. When you click an item in this list, the **Event Sources** section updates to highlight the event source you selected. You can repeat steps 3 and 4 to redefine each event or click the **X** button to remove the selected event.

6. Click the **OK** button to save the configuration and close the dialog box. The event cases you configured appear as selection options in the event selector label at the top of the Event structure and the Event Data node displays the data common to all events handled in that case.
7. (Optional) You can use a Timeout event to configure an Event structure to wait a specified amount of time for an event to occur. Wire a value to the Timeout terminal at the top left of the Event structure to specify the number of milliseconds the Event structure should wait for an event to occur before generating a Timeout event. The default value for the Timeout terminal is –1, which specifies to wait indefinitely for an event to occur.
8. Repeat steps 1 through 6 for each event case you want to configure.

Refer to the following VIs for examples of using events:

`labview\examples\general\dynamicevents.llb`

`labview\examples\general\uievents.llb`

Event Example

Figure 3-1 shows an Event structure configured with the Menu Selection (User) event. This VI uses the Event structure to capture menu selections made using the user-defined menu named `sample.rtm`. The ItemTag returns the menu item that was selected and the MenuRef returns the refnum to the menubar. This information is passed to the Get Menu Item Info function. Refer to `examples\general\uievents.llb` for more examples of using events.

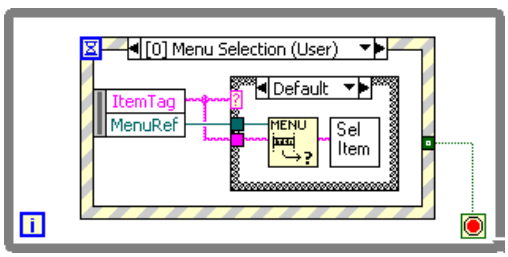


Figure 3-1. Menu Selection (User) Event

Note If you use the Get Menu Selection function with an Event structure configured to handle the same menu item, the Event structure takes precedence, and LabVIEW ignores the Get Menu Selection function. In any given VI, use the Event structure or the Get Menu Selection function to handle menu events, not both.

C. Caveats and Recommendations

The following list describes some of the caveats and recommendations to consider when incorporating events into LabVIEW applications.

- Avoid using an Event structure outside a loop.
  - LabVIEW can generate events even when no Event structure is waiting to handle them. Because the Event structure handles only one event each time it executes, place the Event structure in a While Loop that terminates when the VI is no longer interested in events to ensure that an Event structure handles all events that occur.
- Remember to read the terminal of a latched Boolean control in its Value Change event case.
  - When you trigger an event on a Boolean control configured with a latching mechanical action, the Boolean control does not reset to its default value until the block diagram reads the terminal on the Boolean control. You must read the terminal inside the event case for the mechanical action to work correctly.
- Avoid placing two Event structures in one loop.
  - National Instruments recommends that you place only one Event structure in a loop. When an event occurs in this configuration, the Event structure handles the event, the loop iterates, and the Event structure waits for the next event to occur. If you place two Event structures in a single loop, the loop cannot iterate until both Event structures handle an event. If you have enabled front panel locking for the Event structures, the user interface of the VI can become unresponsive depending on how the user interacts with the front panel.

Refer to the *Caveats and Recommendations when Using Events in LabVIEW* topic of the *LabVIEW Help* for more caveats and recommendations when you use events in LabVIEW.

D. Event-Based Design Patterns

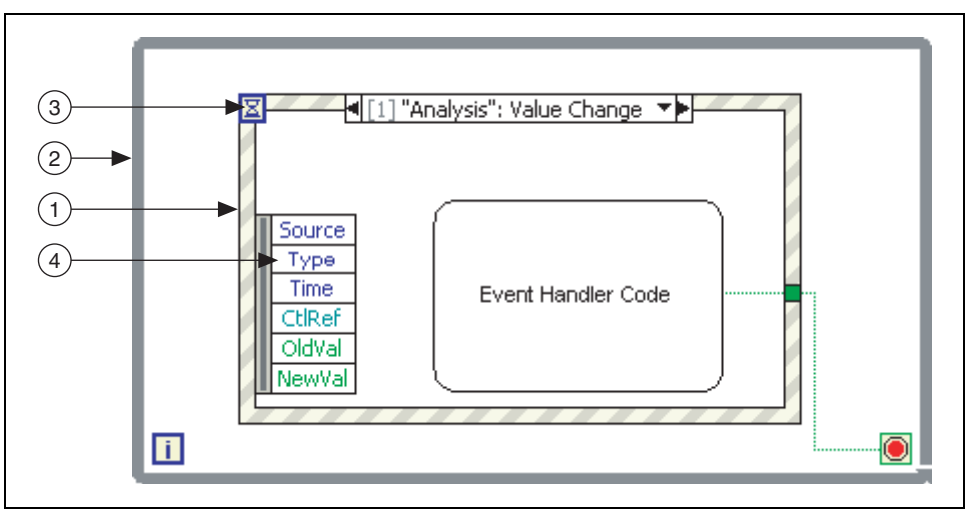
Event-based design patterns provide efficiency gains because they only respond when an event occurs. When LabVIEW executes the Event structure, the VI that contains the Event structure sleeps until a registered events occurs, or generates. When a registered event generates, the Event structure automatically wakes up and executes the appropriate subdiagram to handle the event.

User Interface Event Handler Design Pattern

The user interface event handler design pattern provides a powerful and efficient architecture for handling user interaction with LabVIEW. Use the user interface event handler for detecting when a user changes the value of a control, moves or clicks the mouse, or presses a key.

The standard user interface event handler template consists of an Event structure contained in a While Loop, as shown in Figure 3-2. Configure the Event structure to have one case for each category of event you want to detect. Each event case contains the handling code that executes immediately after an event occurs.

Because the event handler loop wakes up precisely when an event occurs and sleeps in between events, you do not have to poll or read control values repeatedly in order to detect when a user clicks a button. The user interface event handler allows you to minimize processor use without sacrificing interactivity.



1	Event Structure	3	Timeout Terminal
2	While Loop	4	Event Data Node

Figure 3-2. User Interface Event Handler Design Pattern

A common problem when using the user interface event handler is that it computes the While Loop termination before the Event structure executes. This can cause the While Loop to iterate one more time than you expected. To avoid this situation, compute the While Loop termination within all your event handling code.

The event handler code must execute quickly, generally within 200 ms. Anything slower can make it feel as if the user interface is locked up. Also, if the event handler code takes a long time to execute, the Event structure might lock. By default, the front panel locks while an event is handled. You can disable front panel locking for each event case to make the user interface more responsive. However, any new events that are generated while an event is being handled will not be handled immediately. So, the user interface will still seem unresponsive.

Any code that is in an event case cannot be shared with another Event structure. You must use good code design when using the Event structure. Modularize code that will be shared between multiple Event structure cases.

The Event structure includes a Timeout event, which allows you to control when the Timeout event executes. For example, if you set a Timeout of 200 ms, the Timeout event case executes every 200 ms in the absence of other events. You can use the Timeout event to perform critical timing in your code.

Producer/Consumer (Events) Design Pattern

One of the most versatile and flexible design patterns combines the producer/consumer and user interface event handler design patterns. A VI built using the producer/consumer (events) pattern responds to the user interface asynchronously, allowing the user interface to continuously respond to the user. The consumer loop of this pattern responds as events occur, similar to the consumer loop of the producer/consumer (data) design pattern.

The producer/consumer (events) design pattern uses the same implementation as the producer/consumer (data) design pattern except the producer loop uses an Event structure to respond to user interface events, as shown in Figure 3-3. The Event structure enables continuous response to user interaction.

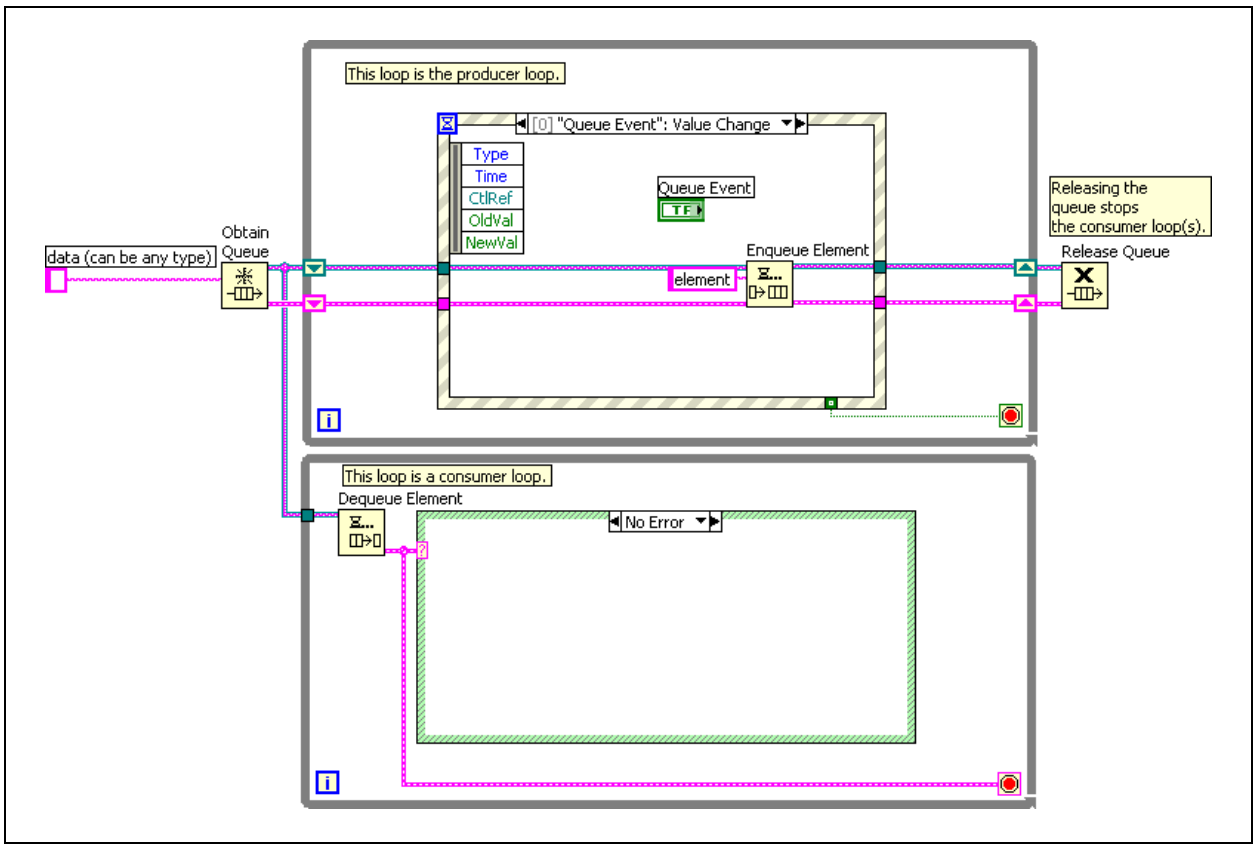


Figure 3-3. Producer/Consumer (Events) Design Pattern

Figure 3-3 shows how you can use Synchronization VIs and functions to add functionality to the design pattern.

Queues have the ability to transfer any data type. The data type transferred in Figure 3-3 is a string. A string is not the most efficient data type for passing data in design patterns. A more efficient data type for passing data in design patterns is a cluster consisting of an enumerated type control and a variant.

Self Review: Quiz

1. Using user interface events allows you to synchronize user actions on the front panel with block diagram execution.
   a. True
   b. False

2. The Event structure handles only one event each time it executes.
   a. True
   b. False

3. Which of the following are examples of user interface events?
   a. Mouse click
   b. Keystroke
   c. Event Filter Node
   d. Value change of a control

4. Which of the following operations will generate a Value Change event for a numeric control?
   a. Click inside the digital display window and enter a number from the keyboard
   b. Click the increment or decrement arrow buttons
   c. Place the cursor to the right of the digit you want to change and press the up or down arrow keys
   d. Update the numeric control using a local variable

Self Review: Quiz Answers

1. Using user interface events allows you to synchronize user actions on the front panel with block diagram execution.
   a. True
   b. False

2. The Event structure handles only one event each time it executes.
   a. True
   b. False

3. Which of the following are examples of user interface events?
   a. Mouse click
   b. Keystroke
   c. Event Filter Node
   d. Value change of a control

4. Which of the following operations will generate a Value Change event for a numeric control?
   a. Click inside the digital display window and enter a number from the keyboard
   b. Click the increment or decrement arrow buttons
   c. Place the cursor to the right of the digit you want to change and press the up or down arrow keys
   d. Update the numeric control using a local variable

Notes

4

Error Handling

By default, LabVIEW automatically handles any error when a VI runs by suspending execution, highlighting the subVI or function where the error occurred, and displaying an error dialog box. Automatic error handling is convenient for quick prototypes and proof-of-concept development, but not recommended for professional application development. If you rely on automatic error handling your application might stop in a critical section of your code because of an error dialog box. The user might be unable to continue running the application or fix the problem.

By manually implementing error handling, you control when popup dialogs occur. If you plan to create a stand-alone application, you must incorporate manual error handling because LabVIEW does not display automatic error handling dialog boxes in the LabVIEW Run-Time Engine.

Topics

A. Importance of Error Handling

Error handling is the mechanism for anticipation, detection, and resolution of warnings and errors. Error handling is an essential component in your LabVIEW application development. With error handling you quickly pinpoint the source of programming errors. Without it, you might observe unexpected behavior but struggle to find the source of the problem.

Error handling is also extremely valuable when you test your application to ensure that your error reporting is meaningful and that the error handling code safely stops your application when an error occurs. For example, during stress testing you are setting values or conditions that are beyond the normal operational capacity of your application which often result in errors. When such errors occur, you want to ensure proper shutdown of your application.

Error handling continues to be important after an application is deployed. Error handling can help detect system and environment differences—such as differences in file systems, memory, and disk resources.

B. Detect and Report Errors

To implement good error handling, you must determine the actions to take when an error occurs at every point in your application. To begin with, you must utilize the error terminals on functions and VIs. Since the error cluster is implemented as a flow-through parameter, you should propagate errors by wiring the error out cluster of the first node you want to execute to the error in cluster of the next node you want to execute. You must continue to do this for sequences of nodes.

As the VI runs, LabVIEW tests for errors at each node. If LabVIEW does not find any errors, the node executes normally. If LabVIEW detects an error, the node passes the error to the next node without executing that part of the code. Any subVIs that you create should also implement this flow-through behavior.

Use the Merge Error VI to merge the error out cluster values from parallel sequences. Refer to Figure 4-1 for an example of merging error information from parallel node sequences.

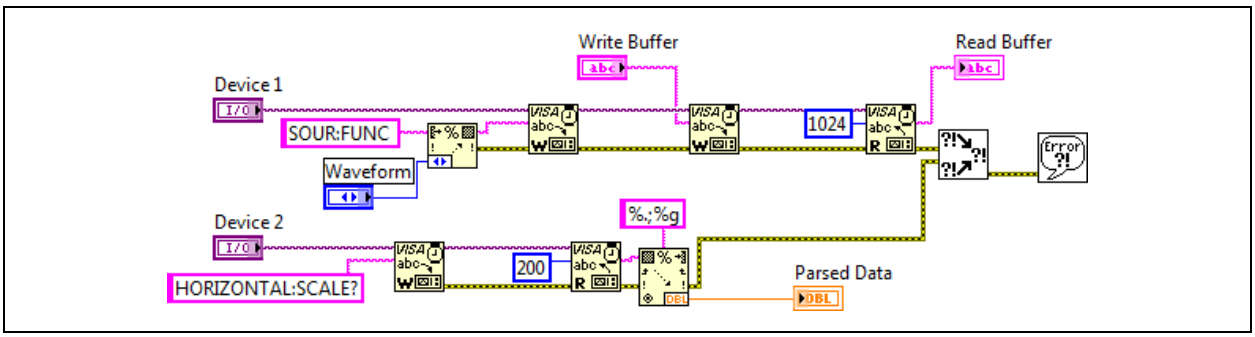


Figure 4-1. Merge Errors From Multiple Sources

At the end of your application after all error sources are merged into one error cluster, you must report errors to the user using the Simple Error Handler VI or another error reporting mechanism.

C. Errors and Warnings

Recall that the error in and error out clusters include the following components of information:

- **status** is a Boolean value that reports TRUE if an error occurred.
- **code** is a 32-bit signed integer that identifies the error numerically. A nonzero error code coupled with a status of FALSE signals a warning rather than a error.
- **source** is a string that identifies where the error occurred.

Notice that an error is defined as status value of TRUE, regardless of the code value. A nonzero code with a status of FALSE is considered a warning. Although most errors have negative code values and warnings have positive code values, this is not universally true. Therefore you should rely on the status value and the code value to detect errors and warnings.

Warnings are typically considered less severe than errors. Some APIs and functions, such as the Regular Expression function, only report errors. However, other APIs such as the VISA API for controlling stand-alone instruments often reports warnings. A common VISA warning occurs when calling the VISA Read function and specifying the number of bytes to read. In this case, VISA returns a warning with the following description: **`The number of bytes transferred is equal to the requested input count. More data might be available.`**

Unlike when an error occurs, nodes execute normally even when LabVIEW detects a warning. Even though code executes normally, it is important that during development you monitor warnings to ensure proper behavior of your application. To ensure that warning information is propagated correctly, it is important to use shift-registers for error cluster wires in loops

so that the warning information is propagated through all iterations. Refer to Figure 4-2 for proper use of the shift-register to propagate errors and warnings to successive loop iterations.

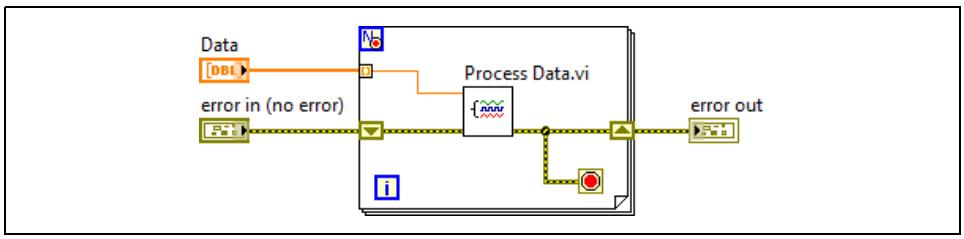


Figure 4-2. Use Shift Registers to Propagate Errors and Warnings

By default, the Simple Error Handler VI displays a dialog with a description of any errors that occurred and does not report warnings. However, the Simple Error Handler VI can be configured for other error handling behavior. You can select the type of dialog from the following options:

- **No dialog**—Displays no dialog box. This is useful if you want to have programmatic control over handling errors.
- **OK message (default)**—Displays a dialog box with a single OK button. After the user acknowledges the dialog box, the VI returns control to the main VI.
- **Continue or stop message**—Displays a dialog box with buttons, which the user can use to either continue or stop. If the user selects Stop, the VI calls the Stop function to halt execution.
- **OK message with warnings**—Displays a dialog box with any warnings and a single OK button. After the user acknowledges the dialog box, the VI returns control to the main VI.
- **Continue or stop message with warnings**—Displays a dialog box with any warnings and buttons, which the user can use to either continue or stop. If the user selects Stop, the VI calls the Stop function to halt execution.

D. Ranges of 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.

Some numeric error codes are used by more than one group of VIs and functions. For example, error 65 is both a serial error code, indicating a serial port timeout, and a networking error code, indicating that a network connection is already established.

E. Error Handlers

An error handler is a VI or code that changes the normal flow of the program when an error occurs. The Simple Error Handler VI is an example of a built-in error handler that is used in LabVIEW. You can implement other error handlers that are customized for your application. For example, you might choose to log error information to a file. Another common error handler is a VI that redirects code to a cleanup or shutdown routine when an error occurs so that your application exits gracefully. Figure 4-3 shows a state machine error handler that sets the next state to be the Shutdown state when an error in status is TRUE.

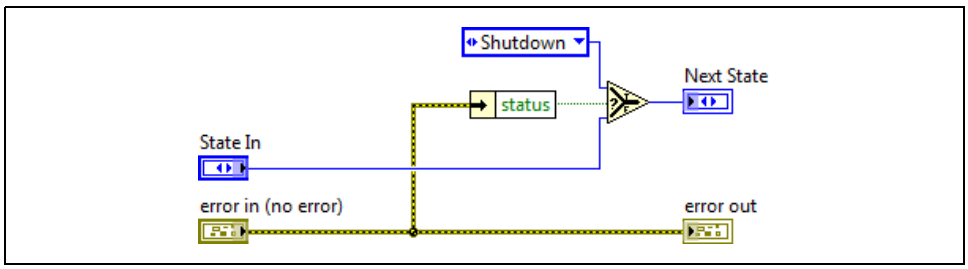


Figure 4-3. State Machine Error Handler

Self Review: Quiz

1. Merge Errors VI concatenates error information from multiple sources.
   a. True
   b. False

2. All errors have negative error codes and all warnings have positive error codes.
   a. True
   b. False

Self Review: Quiz Answers

1. Merge Errors VI concatenates error information from multiple sources.
   a. True
   b. False

2. All errors have negative error codes and all warnings have positive error codes.
   a. True
   b. False

Notes

5

Controlling the User Interface

When writing programs, often you must change the attributes of front panel objects programmatically. For example, you may want to make an object invisible until a certain point in the execution of the program. In LabVIEW, you can use VI Server to access the properties and methods of front panel objects. This lesson explains the Property Nodes, Invoke Nodes, VI Server, and control references.

Topics

A. Property Nodes

B. Invoke Nodes

C. VI Server Architecture

D. Control References

A. Property Nodes

Property Nodes access the properties of an object. In some applications, you might want to programmatically modify the appearance of front panel objects in response to certain inputs. For example, if a user enters an invalid password, you might want a red LED to start blinking. Another example is changing the color of a trace on a chart. When data points are above a certain value, you might want to show a red trace instead of a green one. Property Nodes allow you to make these modifications programmatically. You also can use Property Nodes to resize front panel objects, hide parts of the front panel, add cursors to graphs, and so on.

Property Nodes in LabVIEW are very powerful and have many uses. Refer to the *LabVIEW Help* for more information about Property Nodes.

Creating Property Nodes

When you create a property from a front panel object by right-clicking the object, selecting **Create»Property Node**, and selecting a property from the shortcut menu, LabVIEW creates a Property Node on the block diagram that is implicitly linked to the front panel object. If the object has a label, the Property Node has the same label. You can change the label after you create the node. You can create multiple Property Nodes for the same front panel object.

Using Property Nodes

When you create a Property Node, it initially has one terminal representing a property you can modify for the corresponding front panel object. Using this terminal on the Property Node, you can either set (write) the property or get (read) the current state of that property.

For example, if you create a Property Node for a digital numeric control using the Visible property, a small arrow appears on the right side of the Property Node terminal, indicating that you are reading that property value. You can change the action to write by right-clicking the terminal and selecting **Change To Write** from the shortcut menu. Wiring a False Boolean value to the Visible property terminal causes the numeric control to vanish from the front panel when the Property Node receives the data. Wiring a True Boolean value causes the control to reappear.

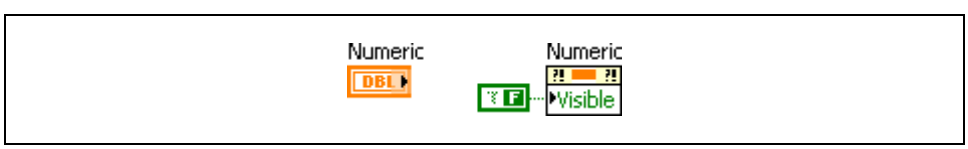


Figure 5-1. Using Property Nodes

To get property information, right-click the node and select **Change All to Read** from the shortcut menu. To set property information, right-click the node and select **Change All to Write** from the shortcut menu. If a property is read only, **Change to Write** is dimmed in the shortcut menu. If the small direction arrow on the Property Node is on the right, you are getting the property value. If the small direction arrow on a Property Node is on the left, you are setting the property value. If the Property Node in Figure 5-1 is set to Read, when it executes it outputs a True value if the control is visible or a False value if it is invisible.

Tip Some properties are read-only, such as the Label property, or write only, such as the Value (Signaling) property.

To add terminals to the node, right-click the white area of the node and select **Add Element** from the shortcut menu or use the Positioning tool to resize the node. Then, you can associate each Property Node terminal with a different property from its shortcut menu.

Tip Property Nodes execute each terminal in order from top to bottom.

Some properties use clusters. These clusters contain several properties that you can access using the cluster functions. Writing to these properties as a group requires the Bundle function and reading from these properties requires the Unbundle function. To access bundled properties, select **All Elements** from the shortcut menu. For example, you can access all the elements in the Position property by selecting **Properties»Position» All Elements** from the shortcut menu.

However, you also can access the elements of the cluster as individual properties, as shown in Figure 5-2.

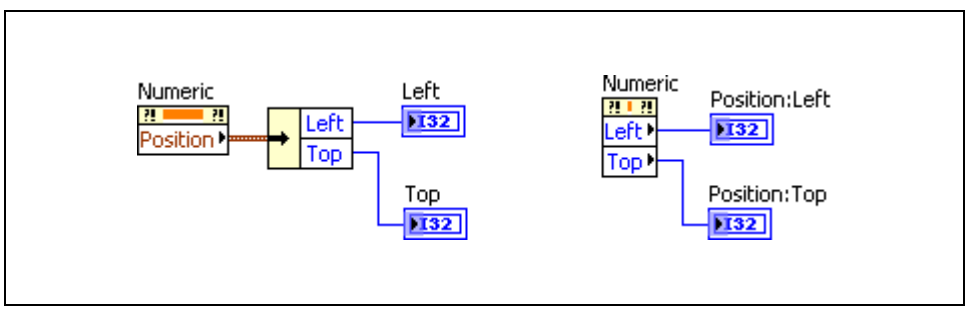


Figure 5-2. Properties Using Clusters

B. Invoke Nodes

Invoke Nodes access the methods of an object.

Use the Invoke Node to perform actions, or methods, on an application or VI. Unlike the Property Node, a single Invoke Node executes only a single method on an application or VI. Select a method by using the Operating tool to click the method terminal or by right-clicking the white area of the node and selecting **Methods** from the shortcut menu. You also can create an implicitly linked Invoke Node by right-clicking a front panel object, selecting **Create»Invoke Node**, and selecting a method from the shortcut menu.

The name of the method is always the first terminal in the list of parameters in the Invoke Node. If the method returns a value, the method terminal displays the return value. Otherwise, the method terminal has no value.

The Invoke Node lists the parameters from top to bottom with the name of the method at the top and the optional parameters, which are dimmed, at the bottom.

Example Methods

An example of a method common to all controls is the Reinitialize to Default method. Use this method to reinitialize a control to its default value at some point in your VI. The VI class has a similar method called Reinitialize All to Default.

Figure 5-3 is an example of a method associated with the Waveform Graph class. This method exports the waveform graph image to the clipboard or to a file.

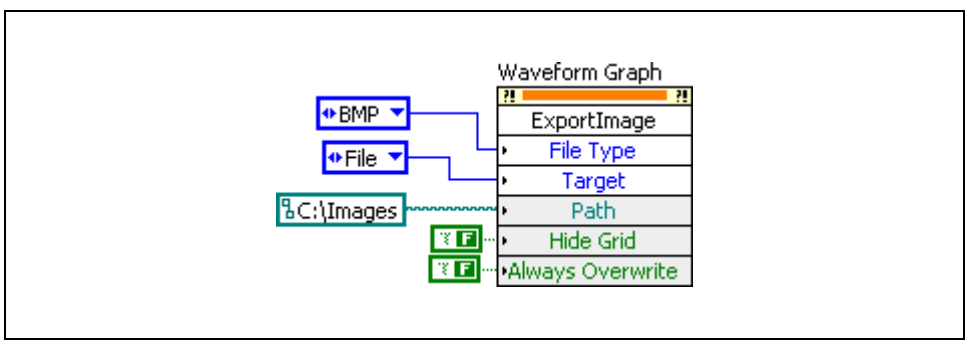


Figure 5-3. Invoke Node for the Export Image Method

C. VI Server Architecture

The VI Server is an object-oriented, platform-independent technology that provides programmatic access to LabVIEW and LabVIEW applications. VI Server performs many functions; however, this lesson concentrates on using the VI Server to control front panel objects and edit the properties of a VI and LabVIEW.To understand how to use VI Server, it is useful to understand the terminology associated with it.

Object-Oriented Terminology

Object-oriented programming is based on objects. An *object* is a member of a class. A *class* defines what an object is able to do, what operations it can perform (methods), and what properties it has, such as color, size, and so on.

Objects can have methods and properties. *Methods* perform an operation, such as reinitializing the object to its default value. *Properties* are the attributes of an object. The properties of an object could be its size, color, visibility, and so on.

Control Classes

LabVIEW front panel objects inherit properties and methods from a class. When you create a Stop control, it is an object of the Boolean class and has properties and methods associated with that class, as shown in Figure 5-4.

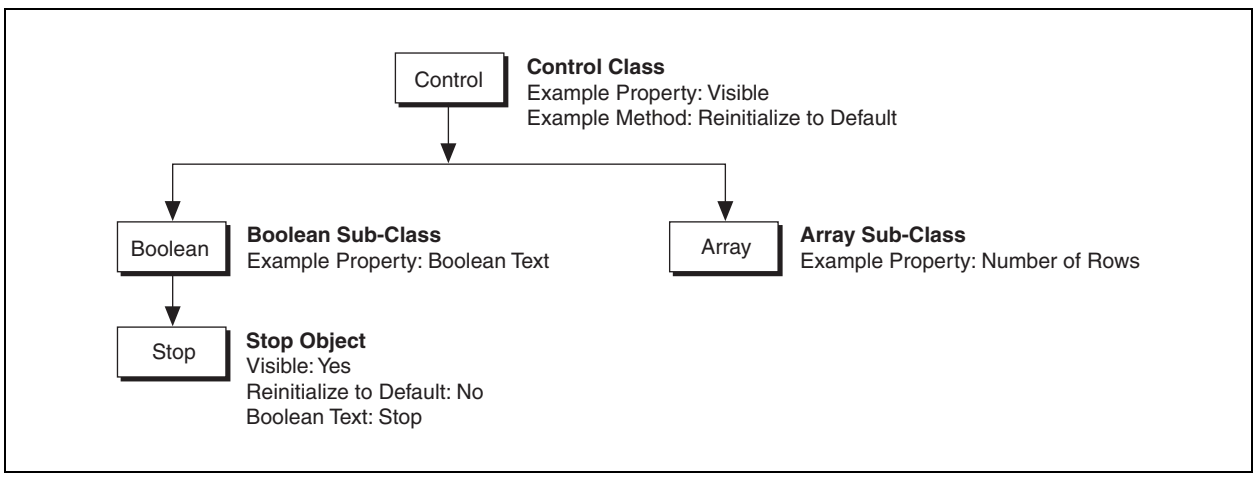


Figure 5-4. Boolean Class Example

VI Class

Controls are not the only objects in LabVIEW to belong to a class. A VI belongs to the VI Class and has its own properties and methods associated with it. For instance, you can use VI class methods to abort a VI, to adjust the position of the front panel window, and to get an image of the block diagram. You can use VI class properties to change the title of a front panel

window, to retrieve the size of the block diagram, and to hide the **Abort** button.

D. Control References

A Property Node created from the front panel object or block diagram terminal is an implicitly linked Property Node. This means that the Property Node is linked to the front panel object. What if you must place your Property Nodes in a subVI? Then the objects are no longer located on the front panel of the VI that contains the Property Nodes. In this case, you need an explicitly linked Property Node. You create an explicitly linked Property Node by wiring a reference to a generic Property Node.

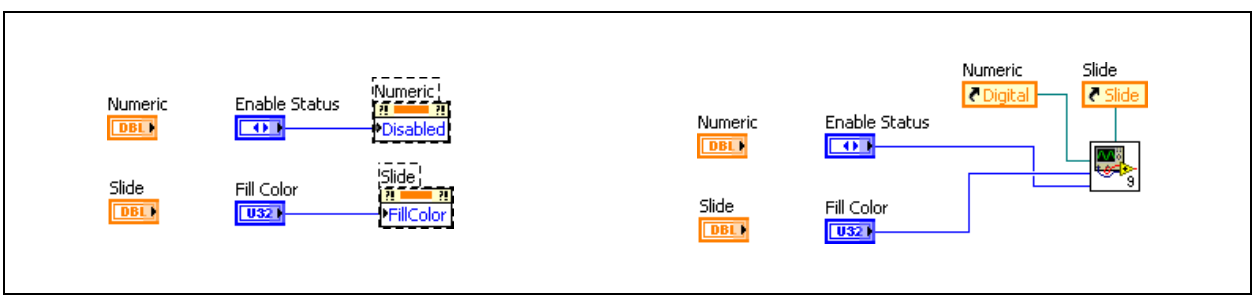
If you are building a VI that contains several Property Nodes or if you are accessing the same property for several different controls and indicators, you can place the Property Node in a subVI and use control references to access that node. A control reference is a reference to a specific front panel object.

This section shows one way to use control references. Refer to the *Controlling Front Panel Objects* topic of the *LabVIEW Help* for more information about control references.

Creating a SubVI with Property Nodes

As shown in Figure 5-5, the simplest way to create explicitly linked Property Nodes is to complete the following steps:

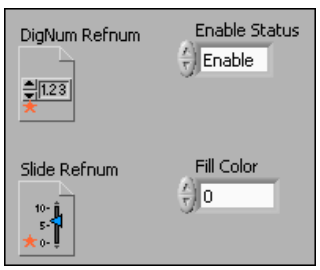
1. Create your VI.
2. Select the portion of the block diagram that is in the subVI, as shown in the first part of Figure 5-5.
3. Select **Edit»Create SubVI**. LabVIEW automatically creates the control references needed for the subVI.
4. Customize and save the subVI. As you can see in the second part of Figure 5-5, the subVI uses the default icon.

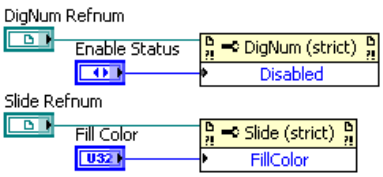


Objects Selected to Create SubVI | Edit»Create SubVI used

Figure 5-5. Using Edit»Create SubVI to Create Control References

Figure 5-6 shows the subVI created. Notice that the front panel Control Refnum controls have been created and connected to a Property Node on the block diagram.





Created SubVI Front Panel Created SubVI Block Diagram

Figure 5-6. Sub VI Created Using Edit»Create SubVI

Note A red star on the Control Reference control indicates that the refnum is strictly typed. Refer to the *Strictly Typed and Weakly Typed Control Refnums* section of the *Controlling Front Panel Objects* topic of the *LabVIEW Help* for more information about weakly and strictly typed control references.

Creating Control References

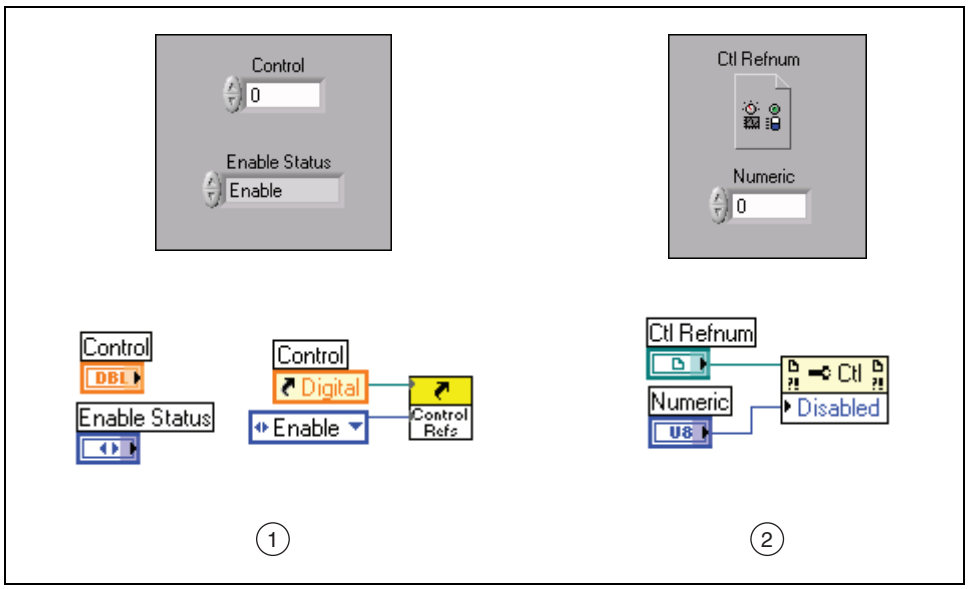
To create a control reference for a front panel object, right-click the object or its block diagram terminal and select **Create»Reference** from the shortcut menu.

You can wire this control reference to a generic Property Node. You can pass the control reference to a subVI using a control refnum terminal.

Using Control References

Setting properties with a control reference is useful for setting the same property for multiple controls. Some properties apply to all classes of controls, such as the Disabled property. Some properties are only applicable to certain control classes, such as the Lock Boolean Text in Center property.

The following example shows how to construct a VI that uses a control reference on the subVI to set the Enable/Disable state of a control on the main VI front panel.



1	Main VI	2	SubVI

Figure 5-7. Control References

The main VI sends a reference for the digital numeric control to the subVI along with a value of zero, one, or two from the enumerated control. The subVI receives the reference by means of the **Ctl Refnum** on its front panel window. Then, the reference is passed to the Property Node. Because the Property Node now links to the numeric control in the main VI, the Property Node can change the properties of that control. In this case, the Property Node manipulates the enabled/disabled state.

Notice the appearance of the Property Node in the block diagram. You cannot select a property in a generic Property Node until the class is chosen. The class is chosen by wiring a reference to the Property Node. This is an example of an explicitly linked Property Node. It is not linked to a control until the VI is running and a reference is passed to the Property Node. The advantage of this type of Property Node is its generic nature. Because it has no explicit link to any one control, it may be reused for many different controls. This generic Property Node is available on the **Functions** palette.

Selecting the Control Type

When you add a Control Refnum to the front panel of a subVI, you next need to specify the VI Server Class of the control. This specifies the type of control references that the subVI will accept. In the previous example, Control was selected as the VI Server Class type, as shown in Figure 5-7. This allows the VI to accept a reference to any type of front panel control.

However, you can specify a more specific class for the refnum to make the subVI more restrictive. For example, you can select Digital as the class, and the subVI only can accept references to numeric controls of the class

Digital. Selecting a more generic class for a control refnum allows it to accept a wider range of objects, but limits the available properties to those that apply to all objects which the Property Node can accept.

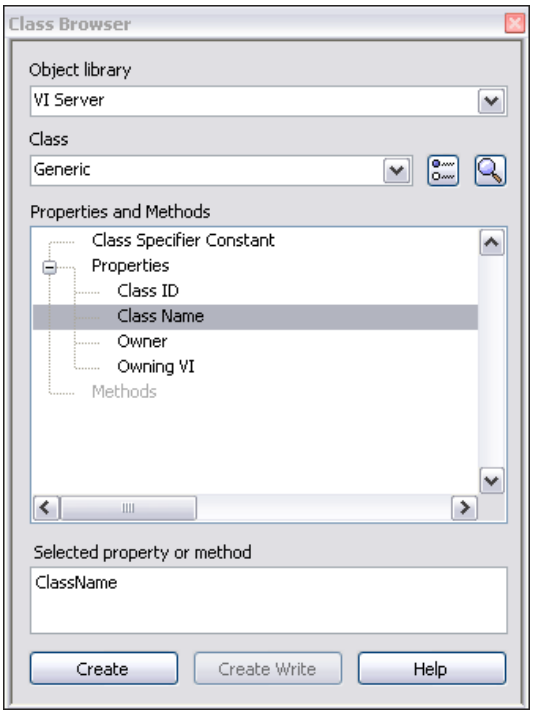
To select a specific control class, right-click the control and select **Select VI Server Class»Generic»GObject»Control** from the shortcut menu. Then, select the specific control class.

Creating Properties and Methods with the Class Browser Window

You can use the Class Browser window to select an object library and create a new property or method.

Complete the following steps to create a new property or method using the Class Browser window.

1. Select **View»Class Browser** to display the Class Browser window.



2. From the Object library pull-down menu, select a library.
3. Select a class from the Class pull-down menu. Use the following buttons to navigate the classes.

- Click the **Select View** button to toggle between an alphabetical view and a hierarchical view of the items in the Class pull-down menu and the Properties and Methods list.

- Click the **Search** button to launch the Class Browser Search dialog box.

4. From the Properties and Methods list in the Class Browser window, select a property or method. The property or method you select appears in the Selected property or method box.
5. Click the **Create** button or the **Create Write** button to attach a node with the selected property or method to your mouse cursor and add the node to the block diagram. The Create button creates a property for reading or a method. This button dims when you select a write-only property. To create a property for writing, click the **Create Write** button. The Create Write button dims when you select a method or read-only property. You also can drag a property or method from the Properties and Methods list directly to the block diagram.
6. Repeat steps 2 through 5 for any other properties and methods you want to create and add to the block diagram.

Self-Review: Quiz

1. For each of the following items, determine whether they operate on a VI class or a Control class.
   - Format and Precision
   - Blinking
   - Reinitialize to Default Value
   - Show Tool Bar

2. You have a ChartGraph control refnum, shown at left, in a subVI. Which of the following control references could you wire to the control refnum terminal of the subVI? (multiple answers)
   a. Control reference of an XY graph
   b. Control reference of a numeric array
   c. Control reference of a waveform chart
   d. Control reference of a Boolean control

Self-Review: Quiz Answers

1. For each of the following items, determine whether they operate on a VI class or a Control class.
   - Format and Precision: **Control**
   - Blinking: **Control**
   - Reinitialize to Default Value: **Control**
   - Show Tool Bar: **VI**

2. You have a GraphChart control refnum, shown at left, in a subVI. Which control references could you wire to the control refnum terminal of the subVI?
   a. Control reference of an XY graph
   b. Control reference of a numeric array
   c. Control reference of a waveform chart
   d. Control reference of a Boolean control

Notes

6

File I/O Techniques

Frequently, the decision to separate the production of data and the consumption of data into separate processes occurs because you must write the data to a file as it is acquired. In such cases, you must choose a file format. This lesson explains ASCII, Binary, and Technical Data Management Streaming (TDMS) file formats and when each is a good choice for your application.

Topics

A. File Formats

B. Binary Files

C. TDMS Files

A. File Formats

At their lowest level, all files written to your computer's hard drive are a series of binary bits. However, many formats for organizing and representing data in a file are available. In LabVIEW, three of the most common techniques for storing data are the ASCII file format, direct binary storage, and the TDMS file format. Each of these formats has advantages and some formats work better for storing certain data types than others.

When to Use Text (ASCII) Files

Use text format files for your data to make it available to other users or applications if disk space and file I/O speed are not crucial, if you do not need to perform random access reads or writes, and if numeric precision is not important.

Text files are the easiest format to use and to share. Almost any computer can read from or write to a text file. A variety of text-based programs can read text-based files.

Store data in text files when you want to access it from another application, such as a word processing or spreadsheet application. To store data in text format, use the String functions to convert all data to text strings. Text files can contain information of different data types.

Text files typically take up more memory than binary and datalog files if the data is not originally in text form, such as graph or chart data, because the ASCII representation of data usually is larger than the data itself. For example, you can store the number –123.4567 in 4 bytes as a single-precision, floating-point number. However, its ASCII representation takes 9 bytes, one for each character.

In addition, it is difficult to randomly access numeric data in text files. Although each character in a string takes up exactly 1 byte of space, the space required to express a number as text typically is not fixed. To find the ninth number in a text file, LabVIEW must first read and convert the preceding eight numbers.

You might lose precision if you store numeric data in text files. Computers store numeric data as binary data, and typically you write numeric data to a text file in decimal notation. Loss of precision is not an issue with binary files.

When to Use Binary Files

Storing binary data, such as an integer, uses a fixed number of bytes on disk. For example, storing any number from 0 to 4 billion in binary format, such as 1, 1,000, or 1,000,000, takes up 4 bytes for each number.

Use binary files to save numeric data and to access specific numbers from a file or randomly access numbers from a file. Binary files are machine readable only, unlike text files, which are human readable. Binary files are the most compact and fastest format for storing data. You can use multiple data types in binary files, but it is uncommon.

Binary files are more efficient because they use less disk space and because you do not need to convert data to and from a text representation when you store and retrieve data. A binary file can represent 256 values in 1 byte of disk space. Often, binary files contain a byte-for-byte image of the data as it was stored in memory, except for cases like extended and complex numeric values. When the file contains a byte-for-byte image of the data as it was stored in memory, reading the file is faster because conversion is not necessary.

Datalog Files

A specific type of binary file, known as a datalog file, is the easiest method for logging cluster data to file. Datalog files store arrays of clusters in a binary representation. Datalog files provide efficient storage and random access, however, the storage format for datalog files is complex, and therefore they are difficult to access in any environment except LabVIEW. Furthermore, in order to access the contents of a datalog file, you must know the contents of the cluster type stored in the file. If you lose the definition of the cluster, the file becomes very difficult to decode. For this reason, datalog files are not recommended for sharing data with others or for storing data in large organizations where you could lose or misplace the cluster definition.

When to Use TDMS Files

To reduce the need to design and maintain your own data file format, National Instruments has created a flexible data model called Technical Data Management Streaming, which is natively accessible through LabVIEW, LabWindows™/CVI™, and DIAdem, and is portable to other applications such as Microsoft Excel. The TDMS data model offers several unique benefits such as the ability to scale your project requirements and easily attach descriptive information to your measurements while streaming your data to disk.

The TDMS file format consists of two files—a `.tdms` file and a `.tdms_index` file. The `.tdms` file is a binary file that contains data and stores properties about that data. The `.tdms_index` file is a binary index

file that speeds up access while reading and provides consolidated information on all the attributes and pointers in the TDMS file. All the information in the `.tdms_index` file is also contained in the `.tdms` file. For this reason, the `.tdms_index` file can be automatically regenerated from the `.tdms` file. Therefore, when you distribute TDMS files, you only need to distribute the `.tdms` file. The internal structure of the TDMS file format is publicly documented, so it is possible to create third-party programs to write and read TDMS files. In addition, there is a TDM Excel Add-in Tool available on `ni.com` that you can install to load `.tdms` files into Microsoft Excel.

Use TDMS files to store test or measurement data, especially when the data consists of one or more arrays. TDMS files are most useful when storing arrays of simple data types such as numbers, strings, or Boolean data. TDMS files cannot store arrays of clusters directly. If your data is stored in arrays of clusters, use another file format, such as binary, or break the cluster up into channels and use the structure of the TDMS file to organize them logically.

Use TDMS files to create a structure for your data. Data within a file is organized into channels. You can also organize channels into channel groups. A file can contain multiple channel groups. Well-grouped data simplifies viewing and analysis and can reduce the time required to search for a particular piece of data.

Use TDMS files when you want to store additional information about your data. For example, you might want to record the following information:

- Type of tests or measurements
- Operator or tester name
- Serial numbers
- Unit Under Test (UUT) numbers for the device tested
- Time of the test
- Conditions under which the test or measurement was conducted

B. Binary Files

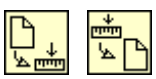
Although all file I/O methods eventually create binary files, you can directly interact with a binary file by using the Binary File functions. The following list describes the common functions that interact with binary files.

Open/Create/Replace File—Opens a reference to a new or existing file for binary files as it does for ASCII Files.

Write to Binary File—Writes binary data to a file. The function works much like the Write to Text File function, but can accept most data types.

Read from Binary File—Reads binary data starting at its current file position. You must specify to the function the data type to read. Use this function to access a single data element or wire a value to the count input. This causes the function to return an array of the specified data type.

Get File Size—Returns the size of the file in bytes. Use this function in combination with the Read from Binary File function when you want to read all of a binary file. Remember that if you are reading data elements that are larger than a byte you must adjust the count to read.



Get/Set File Position—These functions get and set the location in the file where reads and writes occur. Use these functions for random file access.

Close File—Closes an open reference to a file.

Figure 6-1 shows an example that writes an array of doubles to a binary file. Refer to the *Arrays* section of this lesson for more information about the **Prepend array or string size?** option.

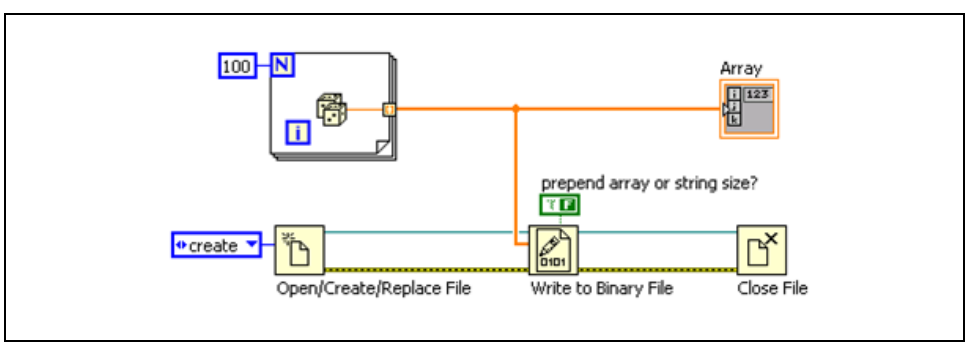


Figure 6-1. Writing a Binary File

Binary Representation

Each LabVIEW data type is represented in a specified way when written to a binary file. This section discusses the representation of each type and important issues when dealing with the binary representation of each type.

Tip A bit is a single binary value. Represented by a 1 or a 0, each bit is either on or off. A byte is a series of 8 bits.

Boolean Values

LabVIEW represents Boolean values as 8-bit values in a binary file. A value of all zeroes represents False. Any other value represents True. This divides files into byte-sized chunks and simplifies reading and processing files. To

efficiently store Boolean values, convert a series of Boolean values into an integer using the Boolean Array To Number function. Figure 6-2 shows two methods for writing six Boolean values to a binary file.

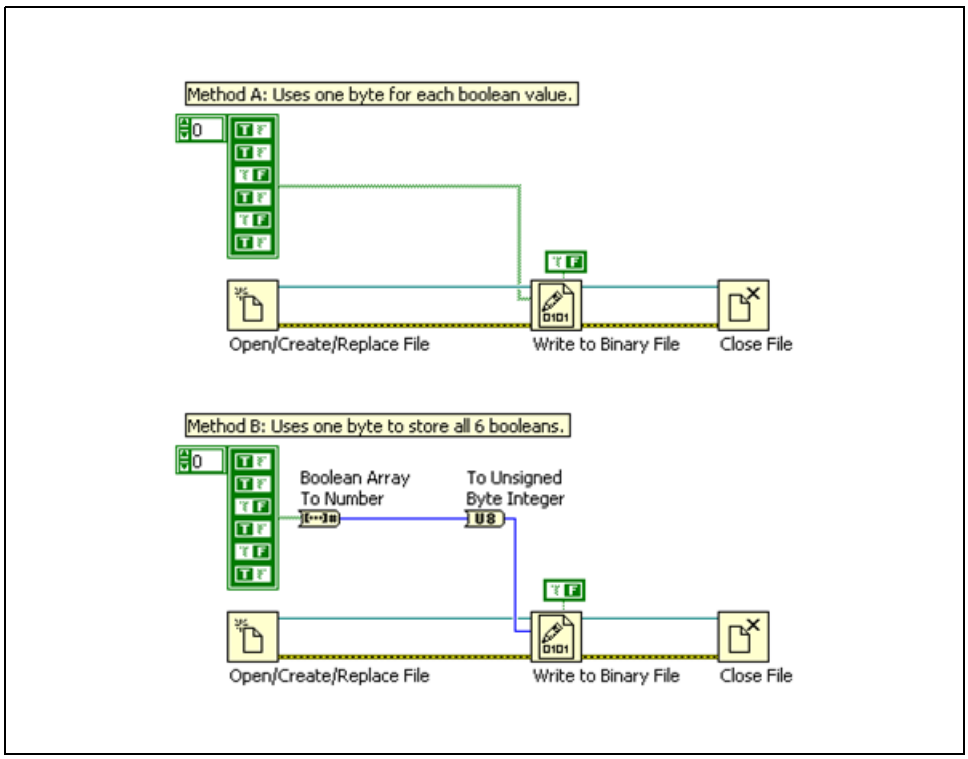


Figure 6-2. Writing Boolean Values to a Binary File

Table 6-1 displays a binary representation of the file contents resulting from running the programs in Figure 6-2. Notice that Method B is a more efficient storage method.

Table 6-1. Results of Figure 6-2

Method A	00000001 00000001 00000000 00000001 00000000 00000001
Method B	00101011

8-bit Integers

Unsigned 8-bit integers (U8) directly correspond to bytes written to the file. When you must write values of various types to a binary file, convert each type into an array of U8s using the Boolean Array To Number, String to Byte Array, Split Number, and Type Cast functions. Then, you can concatenate the various arrays of U8s and write the resulting array to a file. This process is unnecessary when you write a binary file that contains only one type of data.

Table 6-2. U8 Representation

Binary Value	U8 Value
00000000	0
00000001	1
00000010	2
11111111	255

Other Integers

Multi-byte integers are broken into separate bytes and are stored in files in either little-endian or big-endian byte order. Using the Write to Binary File function, you can choose whether you store your data in little-endian or big-endian format.

Little-endian byte order stores the least significant byte first, and the most significant byte last. Big-endian order stores the most significant byte first, and the least significant byte last.

From a hardware point of view, Intel x86 processors use the little-endian byte order while Motorola, PowerPC and most RISC processors use the big-endian byte order. From a software point of view, LabVIEW uses the big-Endian byte order when handling and storing data to disk, regardless of the platform. However, the operating system usually reflects the byte order format of the platform it's running on. For example, Windows running on an Intel platform usually stores data to file using the little-endian byte order. Be aware of this when storing binary data to disk. The binary file functions of LabVIEW have a byte order input that sets the endian form of the data.

Table 6-3. Integer Representations

U32 Value	Little-endian Value	Big-endian Value
0	00000000 00000000 00000000 00000000	00000000 00000000 00000000 00000000
1	00000001 00000000 00000000 00000000	00000000 00000000 00000000 00000001
255	11111111 00000000 00000000 00000000	00000000 00000000 00000000 11111111

Table 6-3. Integer Representations (Continued)

U32 Value	Little-endian Value	Big-endian Value
65535	11111111 11111111 00000000 00000000	00000000 00000000 11111111 11111111
4,294,967,295	11111111 11111111 11111111 11111111	11111111 11111111 11111111 11111111

Floating-Point Numbers

Floating point numbers are stored as described by the IEEE 754 Standard for Binary Floating-Point Arithmetic. Single-precision numerics use 32-bits each and double-precision numerics use 64-bits each. The length of extended-precision numerics depends on the operating system.

Strings

Strings are stored as a series of unsigned 8-bit integers, each of which is a value in the ASCII Character Code Equivalents Table. This means that there is no difference between writing strings with the Binary File functions and writing them with the Text File functions.

Arrays

Arrays are represented as a sequential list of each of their elements. The actual representation of each element depends on the element type. When you store an array to a file you have the option of preceding the array with a header. A header contains a 4-byte integer representing the size of each dimension. Therefore, a 2D array with a header contains two integers, followed by the data for the array. Figure 6-3 shows an example of writing a 2D array of 8-bit integers to a file with a header. The **prepend array or string size?** input of the Write to Binary File function enables the header. Notice that the default value of this terminal is True. Therefore, headers are added to all binary files by default.

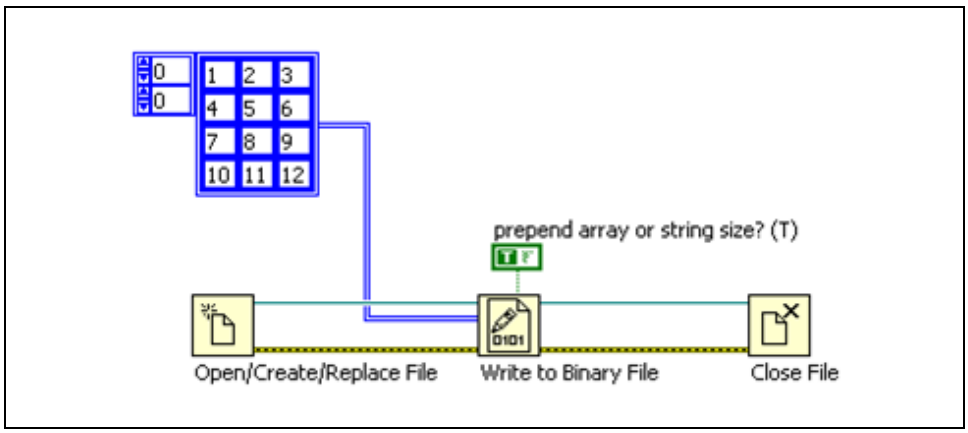


Figure 6-3. Writing a 2D Array of Unsigned Integers to a File with a Header

Table 6-4 shows the layout of the file that the code in Figure 6-3 generates. Notice that the headers are represented as 32-bit integers even though the data is 8-bit integers.

Table 6-4. Example Array Representation In Binary File

4	3	1	2	3	4	5	6	7	8	9	10	11	12

Clusters

Datalog files best represent clusters in binary files. Refer to the *Datalog Files* section for more information.

Sequential vs. Random Access

When reading a binary file, there are two methods of accessing data. The first is to read each item in order, starting at the beginning of a file. This is called sequential access and works similar to reading an ASCII file. The second is to access data at an arbitrary point within the file for random access. For example, if you know that a binary file contains a 1D array of 32-bit integers that was written with a header and you want to access the tenth item in the array, you could calculate the offset in bytes of that element in the file and then read only that element. In this example, the element has an offset of 4 (the header) + 10 (the array index) × 4 (the number of bytes in an I32) = 44.

Sequential Access

To sequentially access all the data in a file, you can call the Get File Size function and use the result to calculate the number of items in the file, based on the size of each item and the layout of the file. You can then wire the number of items to the count terminal of the Read Binary function.

Figure 6-4 shows an example of this method.

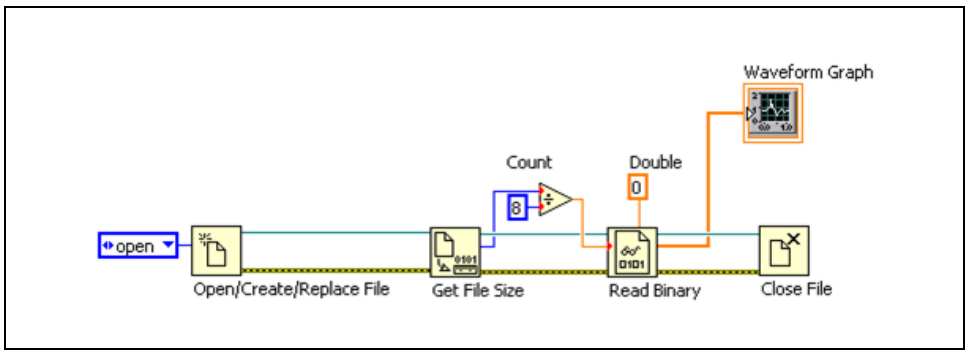


Figure 6-4. Sequentially Reading an Entire File

Alternately, you can sequentially access the file one item at a time by repeatedly calling the Read Binary function with the default count of 1. Each read operation updates the position within the file so that you read a new item each time read is called. When using this technique to access data you can check for the **End of File** error after calling the Read Binary function or calculate the number of reads necessary to reach the end of the file by using the Get File Size function.

Random Access

To randomly access a binary file, use the Set File Position function to set the read offset to the point in the file you want to begin reading. Notice that the offset is in bytes. Therefore, you must calculate the offset based on the layout of the file. In Figure 6-5, the VI returns the array item with the index specified, assuming that the file was written as a binary array of double-precision numerics with no header, like the one written by the example in Figure 6-1.

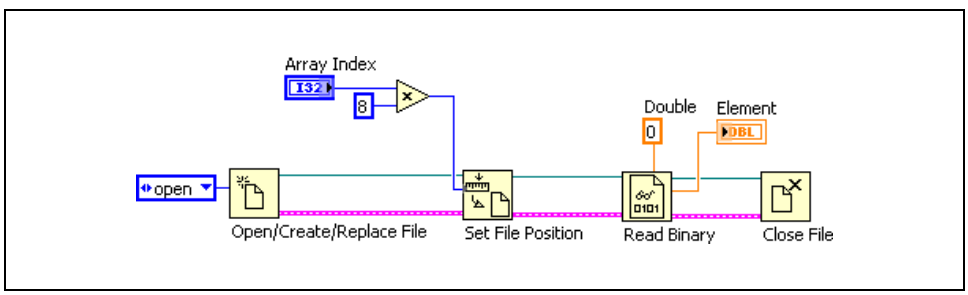


Figure 6-5. Randomly Accessing a Binary File

Datalog Files

Datalog files are designed for storing a list of records to a file. Each record is represented by a cluster, and can contain multiple pieces of data with any data type. Datalog files are binary files, however, they use a different API

than other binary files. The Datalog functions allow you to read and write arrays of clusters to and from datalog files.

When you open a datalog file for either reading or writing, you must specify the record type used by the file. To do this, wire a cluster of the appropriate type to the Open/Create/Replace Datalog function. After the file is open, you program datalog files like any other binary file. Random access is available, although offsets are specified in records instead of bytes.

Figure 6-6 shows an example of writing a datalog file. Notice that the cluster bundles the data and opens the datalog file.

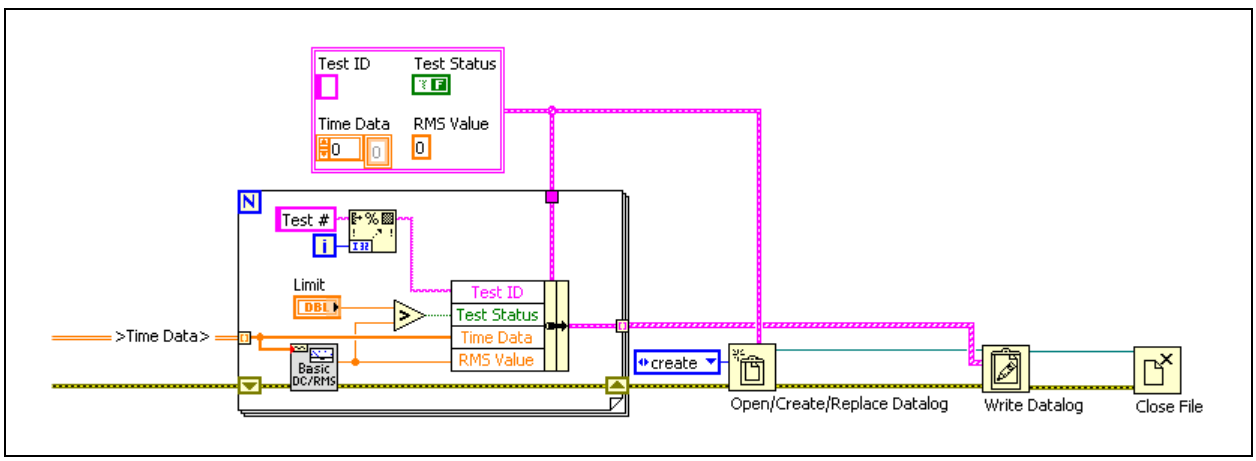


Figure 6-6. Writing a Datalog File

Figure 6-7 shows an example of randomly accessing a datalog file. Notice that the Record Definition cluster matches the cluster used to write the file. If the **record type** wired to the Open/Create/Replace Datalog function does not match the records in the file being opened, an error occurs.

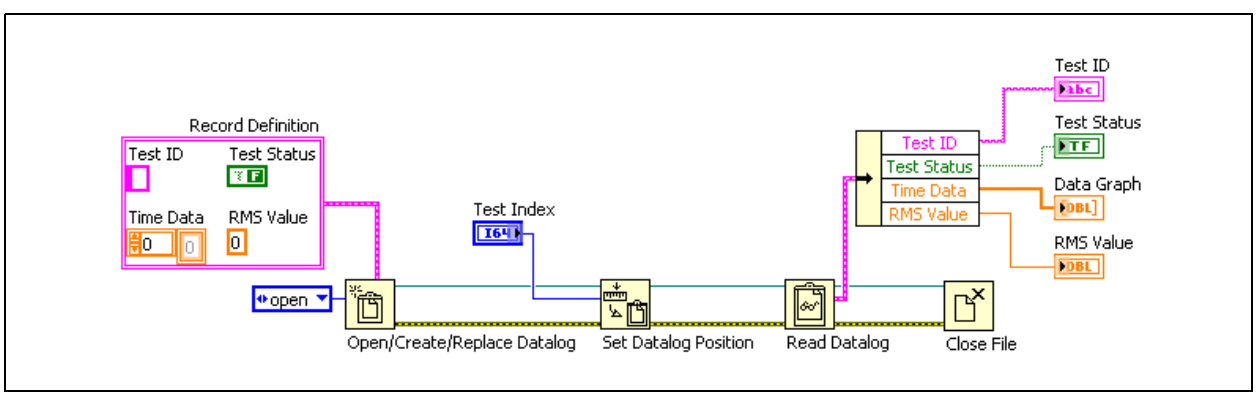


Figure 6-7. Reading a Datalog File

Instead of using random access, you can read an entire datalog file by wiring the output of the Get Number of Records function to the **count** input of the Read Datalog function.

C. TDMS Files

Creating TDMS Files

In LabVIEW, you can create TDMS Files in two ways. Use the Write to Measurement File Express VI and Read from Measurement File Express VI or the TDM Streaming API.

With the Express VIs you can quickly save and retrieve data from the TDMS format. Figure 6-8 shows the configuration dialog box for the Write to Measurement File Express VI. Notice that you can choose to create a LabVIEW measurement data file (LVM) or TDMS file type. However, these Express VIs give you little control over your data grouping and properties and do not allow you to use some of the features that make TDMS files useful, such as defining channel names and channel group names.

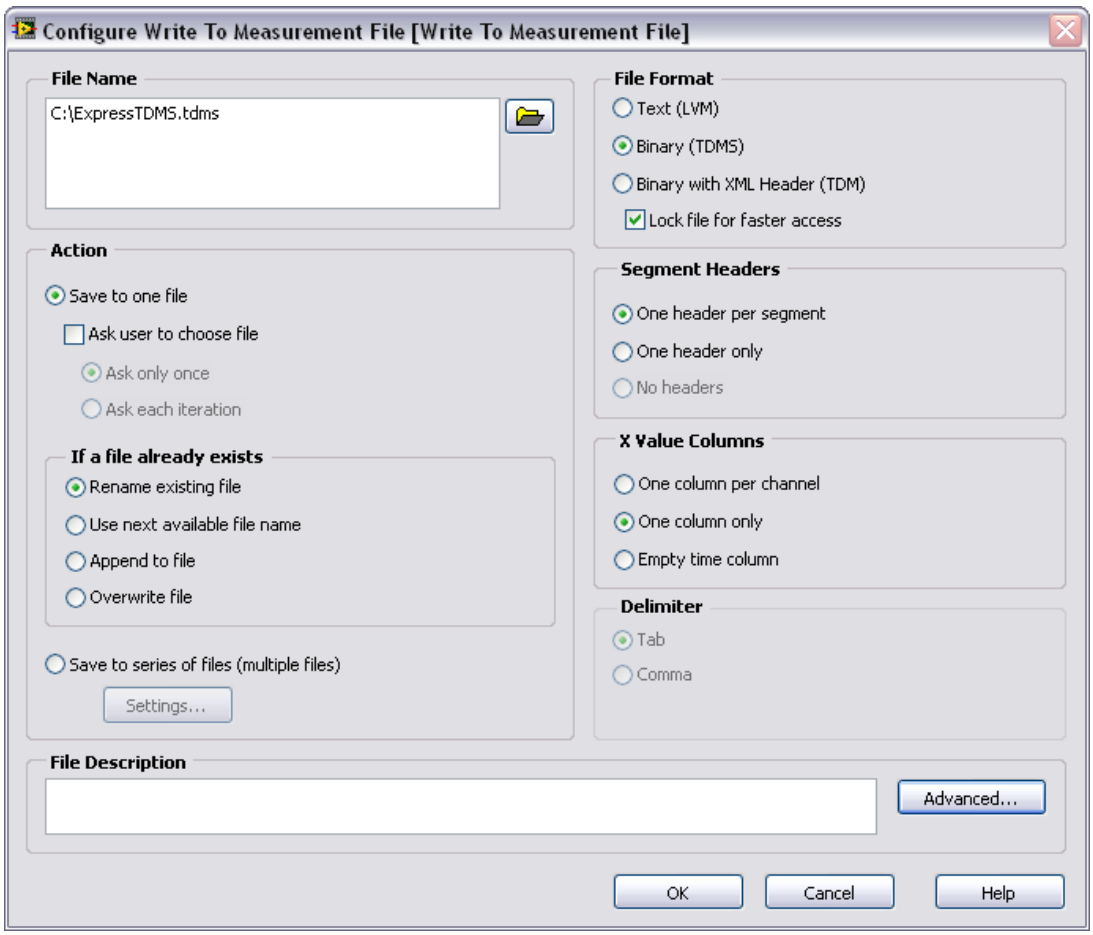


Figure 6-8. Creating a TDMS with Write to Measurement File Express VI

To gain access to the full capabilities of TDMS files, use the TDM Streaming functions. Use the TDM Streaming functions to attach descriptive information to your data and quickly save and retrieve data.

Some of the commonly used TDM Streaming functions are described in the *TDMS API* section of this lesson.

Data Hierarchy

Use TDMS files to organize your data in channels and in channel groups.

A channel stores measurement signals or raw data in a TDMS file. The signal is an array of measurement data. Each channel also can have properties that describe the data. The data stored in the signal is stored as binary data on disk to conserve disk space and efficiency.

A channel group is a segment of a TDMS file that contains properties to store information as well as one or more channels. You can use channel groups to organize your data and to store information that applies to multiple channels.

TDMS files each contain as many channel group and channel objects as you want. Each of the objects in a file has properties associated with it, which creates three levels of properties you can use to store data. For example, test conditions are stored at the file level. UUT information is stored at the channel or channel group level. Storing plenty of information about your tests or measurements can make analysis easier.

TDMS API

The following describes some of the most commonly used TDM Streaming VIs and functions.

- **TDMS Open**—Opens a reference to a TDMS file for reading or writing.

- **TDMS Write**—Streams data to the specified TDMS file. It also allows you to create channels and channel groups within your file.

- **TDMS Read**—Reads the specified TDMS file and returns data from the specified channel and/or channel group.

- **TDMS Set Properties**—Sets the properties of the specified TDMS file, channel group, or channel.

- **TDMS Get Properties**—Returns the properties of the specified TDMS file, channel group, or channel.

- **TDMS Close**—Closes a reference to a TDMS File. Notice that you only must close the file reference, any references that you acquire to channels and channel groups close automatically when you close the file reference.

- **TDMS File Viewer**—Opens the specified TDMS file and presents the file data in the TDMS File Viewer dialog box.

- **TDMS List Contents**—Provides a list of group and channel names contained within the specified TDMS file.

- **TDMS Defragment**—Defragments the file data in the specified TDMS data. Use this function to clean up your TDMS data when it becomes cluttered and to increase performance.

- **TDMS Flush**—Flushes the system memory of all TDMS data to maintain data security.

TDMS Programming

Writing a TDMS File

Figure 6-9 shows the simplest form of writing measurement data with the TDMS API. This example writes data to the channel Main Channel in the channel group Main Group.

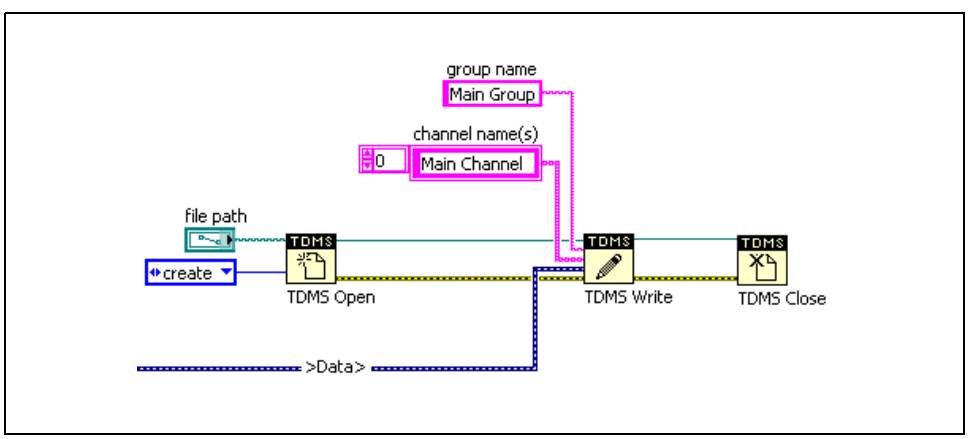


Figure 6-9. Write Data to a TDMS File at the Channel Level

Reading a TDMS File

Figure 6-10 shows the simplest form of reading data using the TDMS API. This example reads all the data in the channel Main Channel from channel group Main Group and displays it in the Channel Data waveform graph. Next, the example reads data from all the channels in the channel group Main Group and displays it in the Group Data waveform graph.

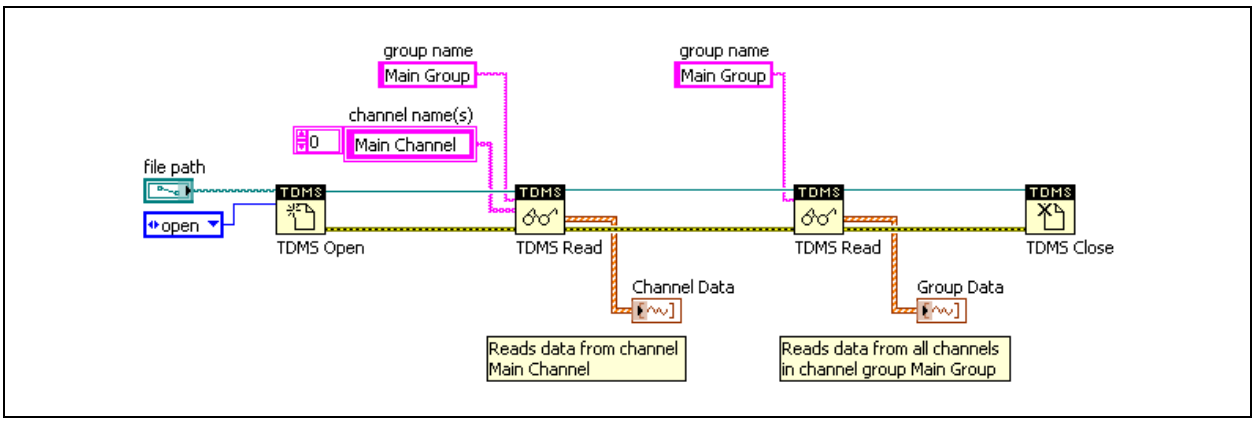


Figure 6-10. Read Data Back from the TDMS File

Writing TDMS Custom Properties

The TDMS data model automatically creates certain properties when some data types are written. However, in many cases you may want to create a property specific to your particular needs, such as UUT, serial number, and test temperature. This task can be accomplished using the TDMS Set Properties function with which you can write properties at the file, group, or channel level.

The file level of a property determines which input terminals need to be wired. For example, to write a group property, only the group name input must be wired. To write a channel property, both the group name and channel name inputs must be wired. But, to write a file property, the group name and channel names inputs should be left unwired. Figure 6-11 illustrates these examples.

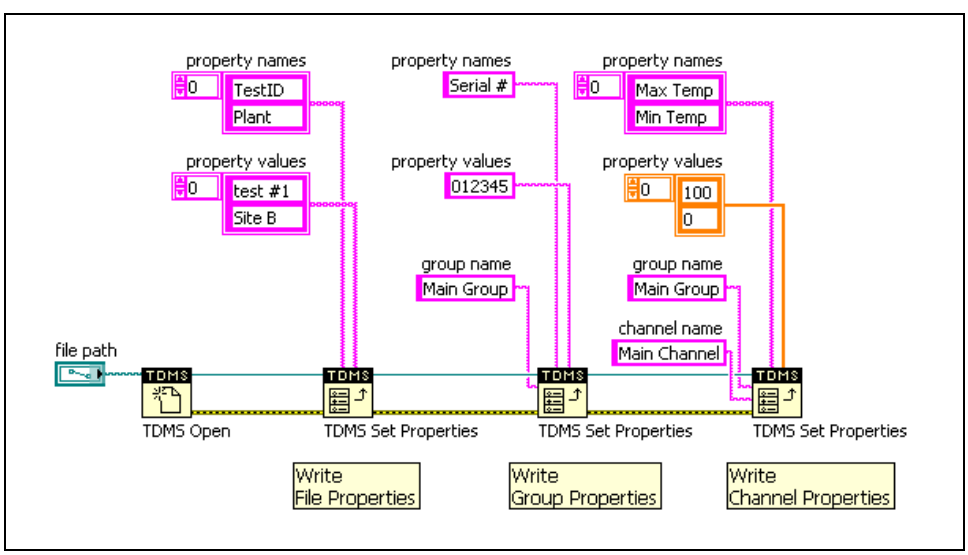


Figure 6-11. Write Custom Properties at Three Different Levels before Writing Data to the File

With the TDMS Set Properties function, you can specify an individual property by itself or specify many properties by using arrays. Figure 6-11 shows two properties specified at the file level (TestID and Plant). You could expand this to specify many more properties by increasing the size of the array. Arrays are not necessary if only a single property, such as a serial number, is written.

Property values can also be different data types. In Figure 6-11, string properties are written at the file and group level. But at the channel level, two numeric properties are written to specify the minimum and maximum temperature.

Reading TDMS Custom Properties

When a TDMS file has been written, the properties can be read back into LabVIEW using TDMS Get Properties function. Properties are returned only for the level specified by the wiring of the group name and channel name inputs. This process is similar to writing the properties, as it is shown in Figure 6-12.

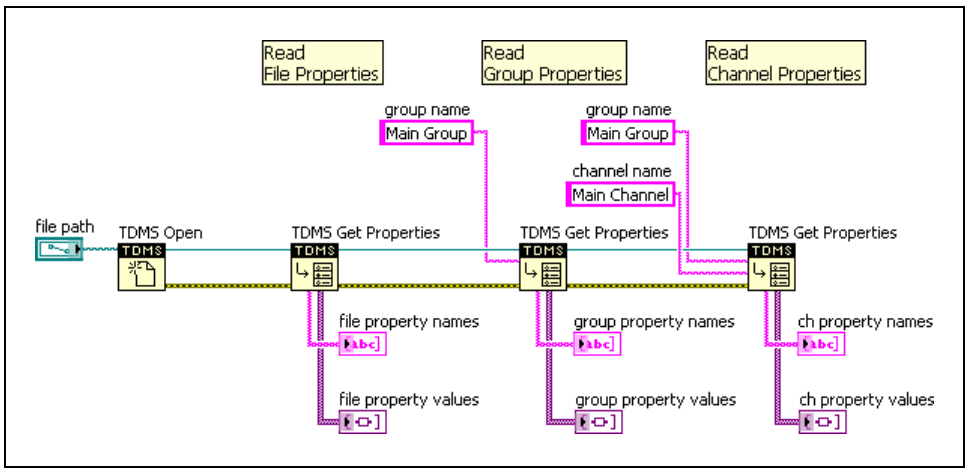


Figure 6-12. Read TDMS Properties from Three Different Levels

In this configuration, the property values are returned as an array of Variant data because the data could be a string, double, Boolean, or another data type. The data can be displayed on the front panel as a Variant or it can be converted in LabVIEW to the appropriate data type. If a property name and its data type are known, they can be wired as inputs to TDMS Get Properties function and read directly with the correct data type.

TDMS File Viewer

Use the TDMS File Viewer VI when developing a TDMS application to automatically see everything that has been written to a TDMS file by making a single VI call. The TDMS File Viewer VI is flexible and can read complex TDMS files. The TDMS File Viewer VI is included with the TDMS API, so it can be easily placed in a program. Place the TDMS File Viewer VI after the file is closed to use it. The TDMS File Viewer launches another window in which you can view the data and properties inside the TDMS file.

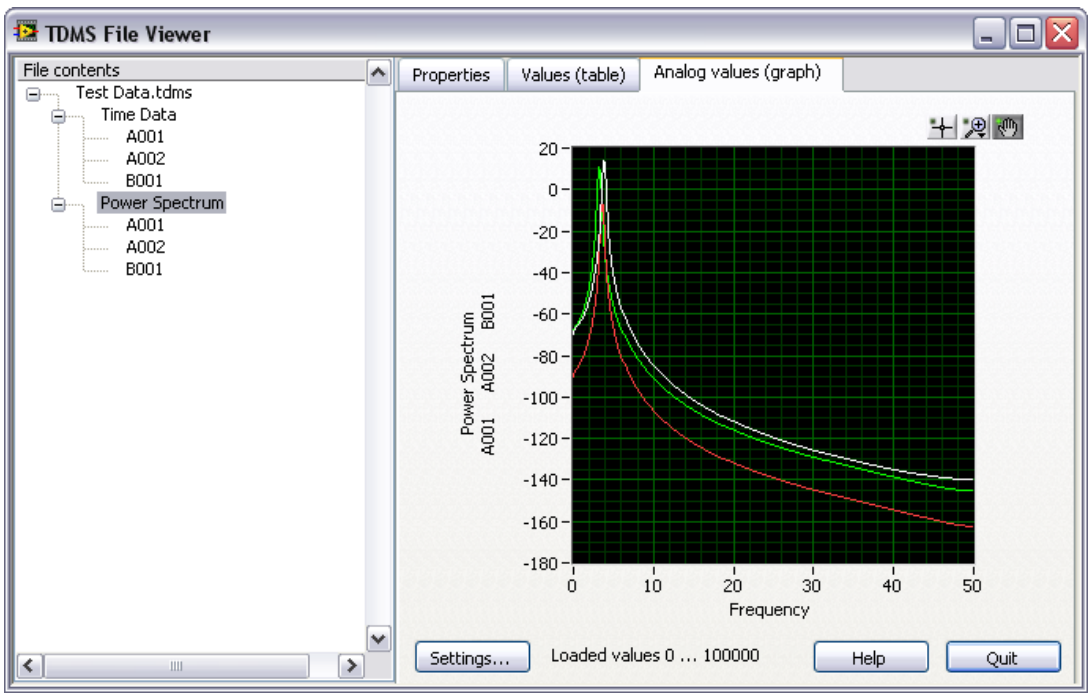


Figure 6-13. TDMS File Viewer

Grouping TDMS Data

Carefully consider the best way to group your data because the data grouping can have a significant impact on the execution speed and implementation complexity of writes and reads. Consider the original format of your data and how you want to process or view the data when choosing a grouping scheme.

One technique is to group data by the type of data. For example, you might put numeric data in one channel group and string data in another, or you might put time domain data in one group and frequency domain data in another. This makes it easy to compare the channels in a group, but can make it difficult to find two channels that are related to each other. Figure 6-14 shows an example of grouping by the type of data. In this example, the temperature data is placed in one group and the wind data is placed in another. Each group contains multiple channels of data. Notice that when grouping by data type you typically have a fixed number of groups, two in this case, and a dynamically determined number of channels.

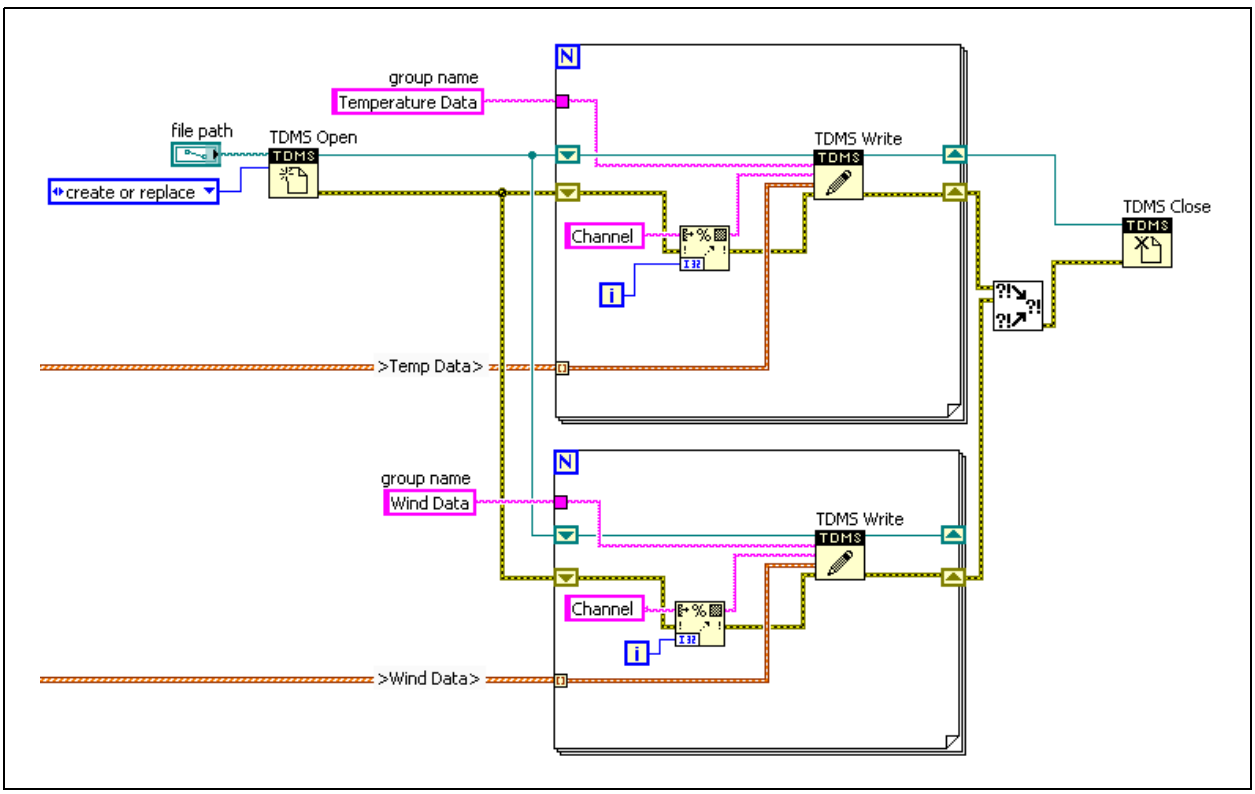


Figure 6-14. Grouping Data by Type

Another technique is to group related data. For example, you might put all the data that applies to a single UUT in one group. Grouping related data allows you to easily locate all the related data about a particular subject, but makes it harder to compare individual pieces of data among subjects.

Relational grouping helps convert cluster-based storage to a TDMS format. You can store all the information from a given cluster in a channel group, with arrays in the cluster representing channels within the group, and scalar items in the cluster representing properties of the channel group. Figure 6-15 shows an example of relational grouping.

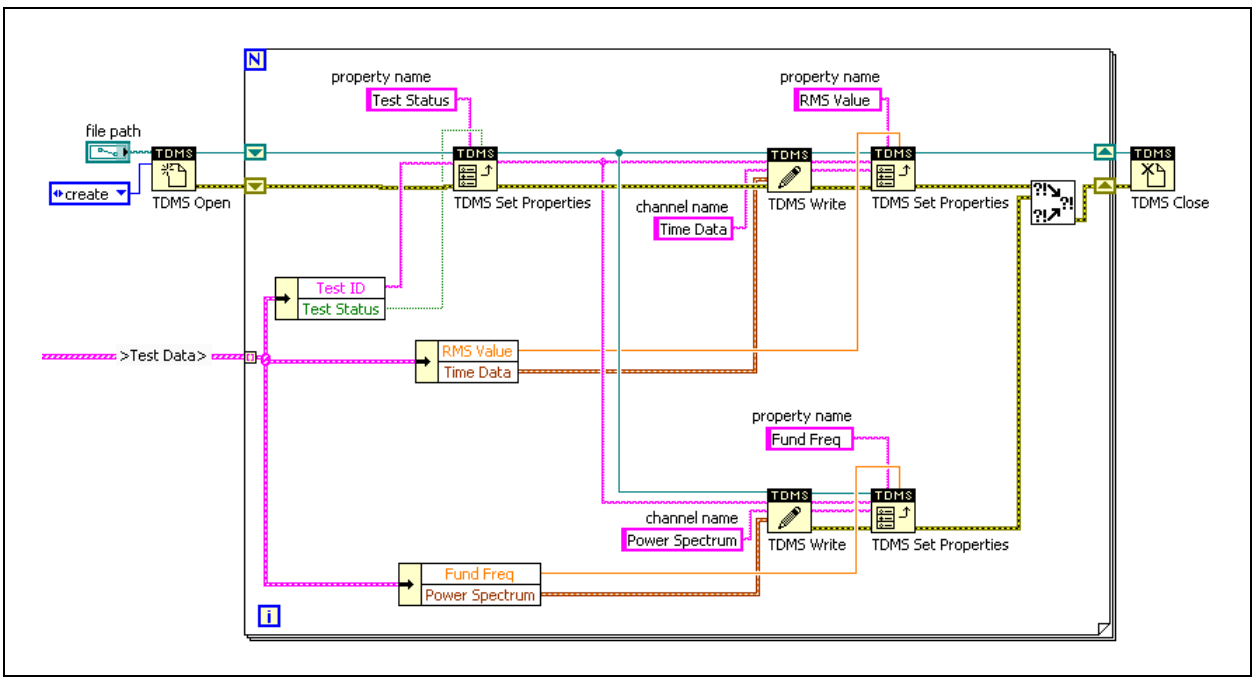


Figure 6-15. Grouping Related Data

Notice that the input data is an array of clusters, each of which contains multiple pieces of information about a test. Each test is stored as a separate channel group. Information that applies to the entire test, such as Test Status, is stored as properties of the channel group. Arrays of data, such as the time data and power spectrum, are stored in channels, and information which relates to the arrays of data, such as the RMS Value and Fundamental Frequency, are stored as properties of the channels. Relational data typically uses a fixed number of channels in a group, but the number of groups is dynamic.

Self-Review: Quiz

1. You need to store test results and organize the data into descriptive groups. In the future, you need to efficiently view the test results by group. Which file storage format should you use?
   a. Tab-delimited ASCII
   b. Custom binary format
   c. TDMS
   d. Datalog

2. You must write a program which saves Portable Network Graphics (PNG) image files. Which file storage method should you use?
   a. Storage file VIs
   b. Binary file functions
   c. ASCII file VIs
   d. Datalog file VIs

3. You must store data that other engineers will later analyze with Microsoft Excel. Which file storage format should you use?
   a. Tab-delimited ASCII
   b. Custom binary format
   c. TDMS
   d. Datalog

4. Which of the following is a little-endian representation of an unsigned 32-bit integer (U32) with a value of 10?
   a. 00001010 00000000 00000000 00000000
   b. 00000000 00000000 00000000 00001010
   c. 00001010
   d. 01010000 00000000 00000000 00000000

5. You can use the Binary File functions to read ASCII files.
   a. True
   b. False

6. TDMS files store properties only at the channel or channel group level.
   a. True
   b. False

Self-Review: Quiz Answers

1. You must store the results of tests to a file. In the future, you need to efficiently search for the tests which meet specific criteria. Which file storage format makes it easiest to query the data?
   a. Tab-delimited ASCII
   b. Custom binary format
   c. TDMS
   d. Datalog

2. You must write a program which saves Portable Network Graphics (PNG) image files. Which file storage method should you use?
   a. Storage file VIs
   b. Binary file functions
   c. ASCII file VIs
   d. Datalog file VIs

3. You need to store data which other engineers will later analyze with Microsoft Excel. Which file storage format should you use?
   a. Tab-delimited ASCII
   b. Custom binary format
   c. TDMS
   d. Datalog

4. Which of the following is a little endian representation of an unsigned 32-bit integer (U32) with a value of 10?
   a. 00001010 00000000 00000000 00000000
   b. 00000000 00000000 00000000 00001010
   c. 00001010
   d. 01010000 00000000 00000000 00000000

5. You can use the Binary File functions to read ASCII files.
   a. True
   b. False

6. TDMS files store properties only at the channel or channel group level.
   a. True
   b. False

Notes

Notes

7

Improving an Existing VI

A common problem when you inherit VIs from other developers is that features may have been added without attention to design, thus making it progressively more difficult to add features later in the life of the VI. This is known as software decay. One solution to software decay is to refactor the software. Refactoring is the process of redesigning software to make it more readable and maintainable so that the cost of change does not increase over time. Refactoring changes the internal structure of a VI to make it more readable and maintainable, without changing its observable behavior.

In this lesson, you will learn methods to refactor inherited code and experiment with typical issues that appear in inherited code.

Topics

A. Refactoring Inherited Code

B. Typical Refactoring Issues

C. Comparing VIs

A. Refactoring Inherited Code

Write large and/or long-term software applications with readability in mind because the cost of reading and modifying the software is likely to outweigh the cost of executing the software. It costs more for a developer to read and understand poorly designed code than it does to read code that was created to be readable. In general, more resources are allocated to reading and modifying software than to the initial implementation. Therefore VIs that are easy to read and modify are more valuable than those that are not.

Creating well-designed software facilitates rapid development and decreases possible decay. If a system starts to decay, you can spend large amounts of time tracking down regression failures, which is not productive. Changes also can take longer to implement because it is harder to understand the system if it is poorly designed.

Consider the inherited VI shown in Figure 7-1.

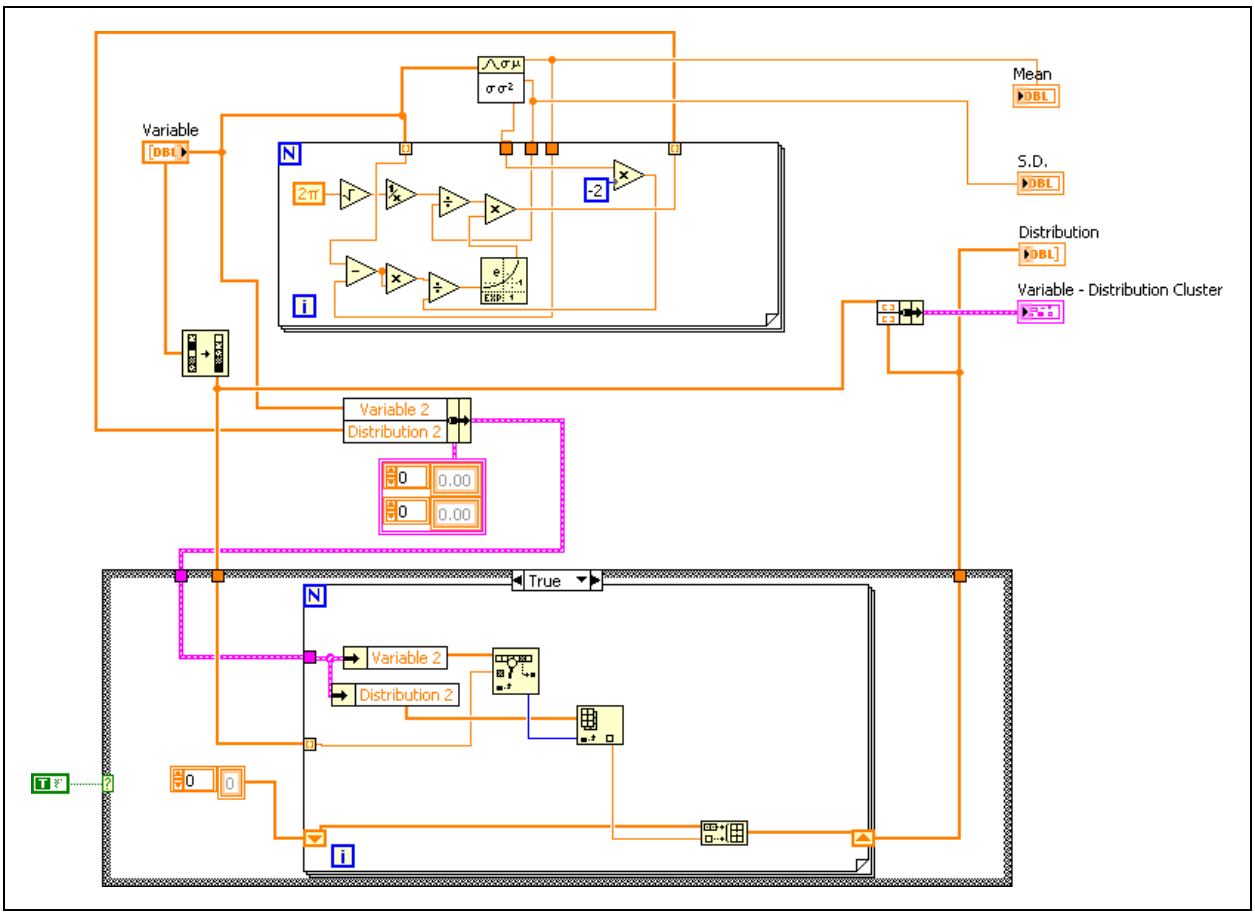


Figure 7-1. Inherited VI

You can refactor the code as shown in Figure 7-2.

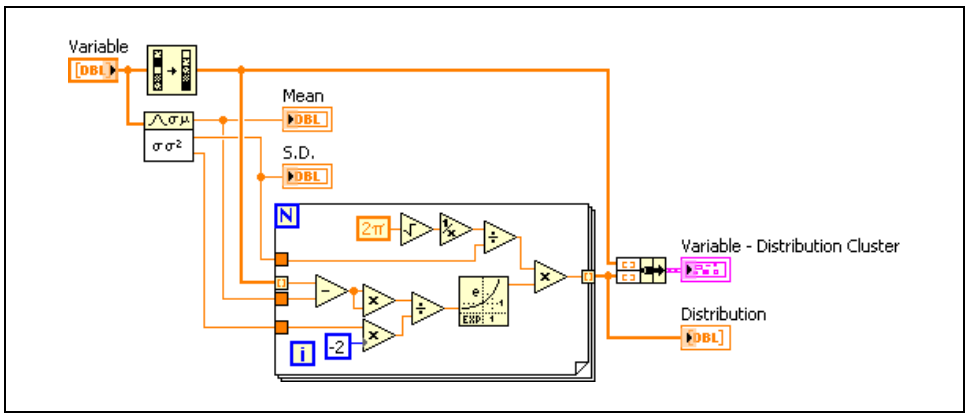


Figure 7-2. Refactored Inherited Code

The refactored code performs the same function as the inherited code, but the refactored code is more readable. The inherited code violates many of the block diagram guidelines you have learned.

When you make a VI easier to understand and maintain, you make it more valuable because it is easier to add features to or debug the VI. The refactoring process does not change observable behavior. Changing the way a VI interacts with clients (users or other VIs) introduces risks that are not present when you limit changes to those visible only to developers. The benefit of keeping the two kinds of changes separate is that you can better manage risks.

Refactoring versus Performance Optimization

Although you can make changes that optimize the performance of a VI, this is not the same as refactoring. Refactoring specifically changes the internal structure of a VI to make it easier to read, understand, and maintain.
A performance optimization is not refactoring because the goal of optimization is not to make the VI easier to understand and modify. In fact, performance optimization can make VIs more difficult to read and understand, which might be an acceptable trade-off. Sometimes you must sacrifice readability for improved performance, however, readability usually takes priority over speed of performance.

When to Refactor

The right time to refactor is when you are adding a feature to a VI or debugging it. Although you might be tempted to rewrite the VI from scratch, there is value in a VI that works, even if the block diagram is not readable. Good candidates for complete rewrites are VIs that do not work or VIs that satisfy only a small portion of your needs. You also can rewrite simple VIs

that you understand well. Consider what works well in an existing VI before you decide to refactor.

B. Typical Refactoring Issues

When you refactor a VI, manage the risk of introducing bugs by making small, incremental changes to the VI and testing the VI after each change. The flowchart shown in Figure 7-3 indicates the process for refactoring a VI.

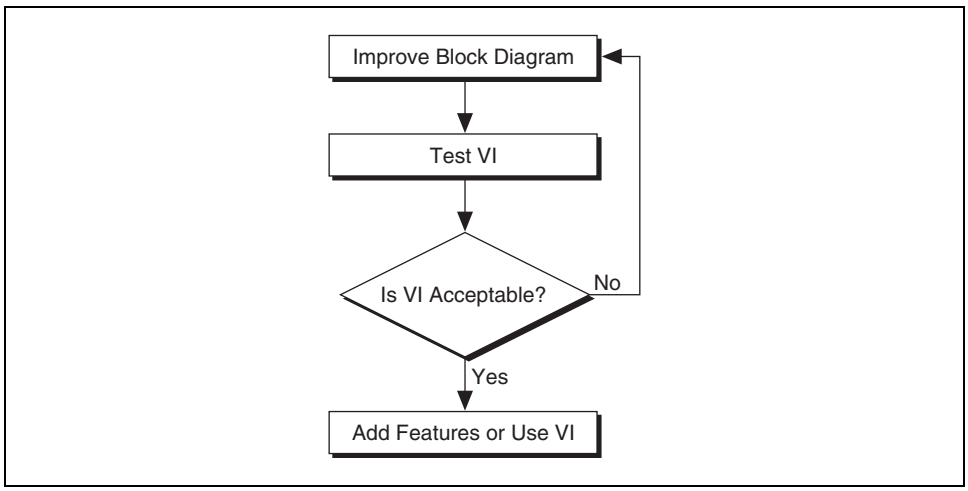


Figure 7-3. Refactoring Flowchart

When you refactor to improve the block diagram, make small cosmetic changes before tackling larger issues. For example, it is easier to find duplicated code if the block diagram is well organized and the terminals are well labeled.

There are several issues that can complicate working with an inherited VI. The following sections describe typical problems and the refactoring solutions you can use to make inherited VIs more readable.

Disorganized or Poorly Designed Block Diagram

Improve the readability of a disorganized VI by relocating objects within the block diagram. You also can create subVIs for sections of the VI that are disorganized. Place comments on areas of a VI that are disorganized to improve the readability of the VI.

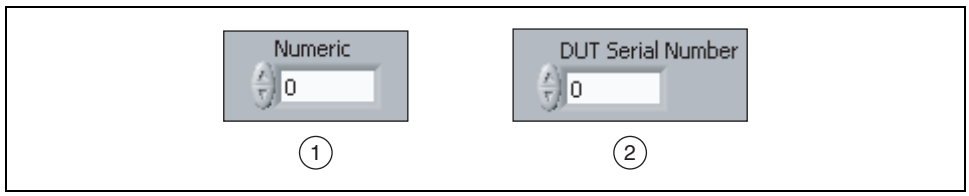
Overly Large Block Diagram

A VI that has a block diagram that is larger than the screen size is difficult to read. You should refactor the VI to make it smaller. The act of scrolling complicates reading a block diagram and understanding the code. Improve a large block diagram by moving objects around. Another technique to

reduce the screen space a block diagram occupies is to create subVIs for sections of code within the block diagram. If you cannot reduce the block diagram to fit on the screen, limit the scrolling to one direction.

Poorly Named Objects and Poorly Designed Icons

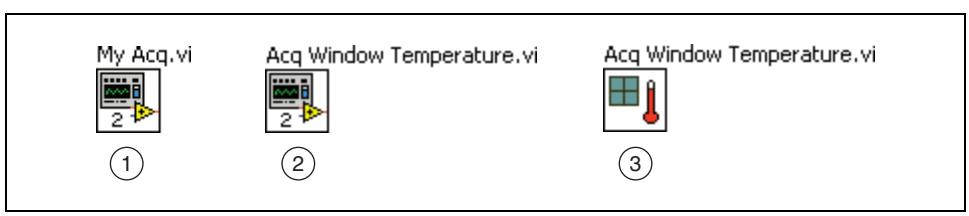
Inherited VIs often contain controls and indicators that do not have meaningful names. For example, the name of Control 1, shown in Figure 7-4, does not indicate its purpose. Control 2 is the same control, renamed to make the block diagram more readable and understandable.



1 Poorly Named Control
2 Meaningfully Named Control

Figure 7-4. Naming Controls

VI names and icons also are important for improving the readability of a VI. For example, the name `My Acq.vi`, shown on the left in Figure 7-5, does not provide any information about the purpose of the VI. You can give the VI a more meaningful name by saving a copy of the VI with a new name and replacing all instances of the VI with the renamed VI. A simpler method is to open all callers of the VI you want to rename, then save the VI with a new name. When you use this method, LabVIEW automatically relinks all open callers of the VI to the new name. `Acq Window Temperature.vi` reflects a more meaningful name for the VI.



1 Poorly Named VI
2 Meaningfully Named VI
3 Meaningful VI Name and VI Icon

Figure 7-5. Poorly Named SubVI

The VI icon also should clarify the purpose of the VI. The default icons used for VI 1 and VI 2 in Figure 7-5 do not represent the purpose of the VI. You can improve the readability of the VI by providing a meaningful icon, as shown for VI 3.

By renaming controls and VIs and creating meaningful VI icons, you can improve the readability of an inherited VI.

Unnecessary Logic

When you read the block diagram in Figure 7-6, notice that it contains unnecessary logic. If a portion of the block diagram does not execute, delete it. Understanding code that executes is difficult, but trying to understand code that never executes is inefficient and complicates the block diagram.

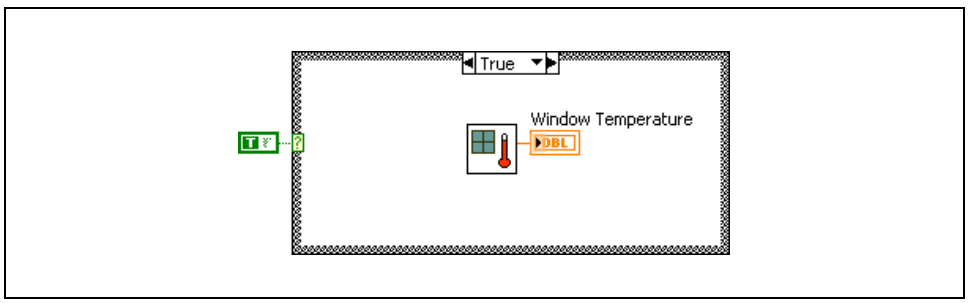


Figure 7-6. Unnecessary Logic

Duplicated Logic

If a VI contains duplicated logic, you always should refactor the VI by creating a subVI for the duplicated logic. This can improve the readability and testability of the VI.

Lack of Dataflow Programming

If there are Sequence structures and local variables on the block diagram, the VI probably does not use data flow to determine the programming flow.

You should replace most Sequence structures with the state machine design pattern. Delete local variables and wire the controls and indicators directly.

Complicated Algorithms

Complicated algorithms can make a VI difficult to read. Complicated algorithms can be more difficult to refactor because there is a higher probability that the changes introduce errors. When you refactor a complicated algorithm, make minor changes and test the code frequently. In some cases you can refactor a complicated algorithm by using built-in LabVIEW functions. For example, the VI in Figure 7-7 checks a user name and password against a database.

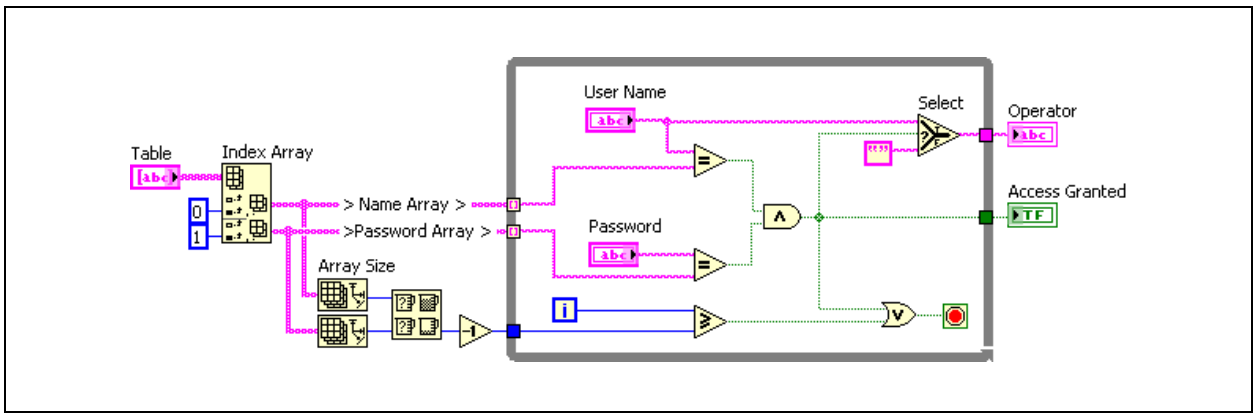


Figure 7-7. Complicated Algorithm VI

You could refactor this VI using the built-in functions for searching strings, as shown in Figure 7-8.

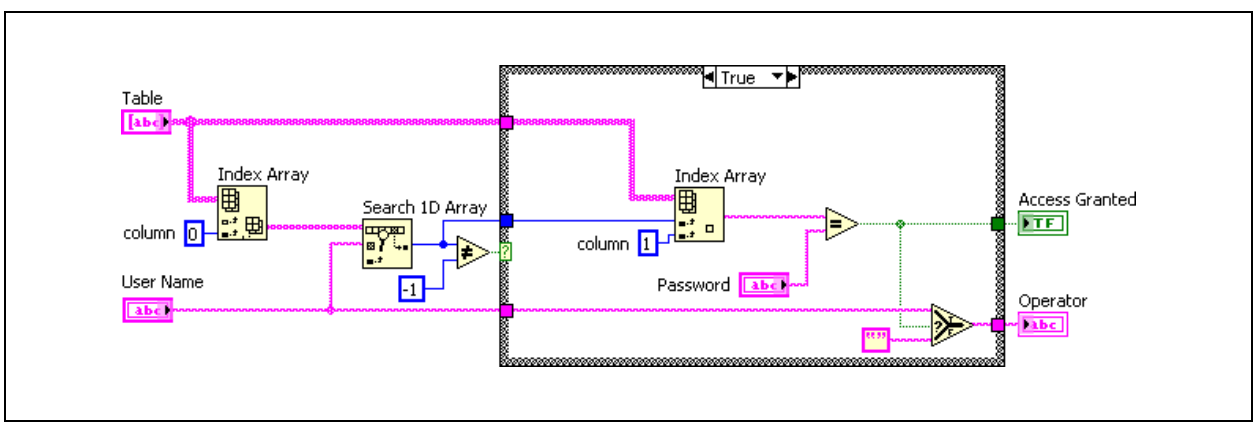


Figure 7-8. Refactored VI

C. Comparing VIs

The LabVIEW Professional Development System includes a utility to determine the differences between two VIs loaded into the memory. Select **Tools»Compare»Compare VIs** to display the Compare VIs dialog box.

From this dialog box, you can select the VIs you want to compare, as well as the characteristics of the VIs to check. When you compare the VIs, both VIs display a Differences window that lists all differences between the two VIs. In this window, you can select various differences and details that you can circle for clarity.

Refactoring Checklist

Use the following refactoring checklist to help determine if you should refactor a VI. If you answer yes to any of the items in the checklist, refer to the guidelines in the *When to Refactor* section of this lesson to refactor the VI.

- ❑ Disorganized block diagram
- ❑ Overly large block diagram
- ❑ Poorly named objects and poorly designed icons
- ❑ Unnecessary logic
- ❑ Duplicated logic
- ❑ Lack of dataflow programming
- ❑ Complicated algorithms

Notes

Notes

8

Creating and Distributing Applications

This lesson describes the process of creating a stand-alone application and installer for your LabVIEW projects.

Topics

A. Preparing the Files

B. Build Specifications

C. Building the Application and Installer

A. Preparing the Files

A stand-alone application allows the user to run your VIs without installing the LabVIEW development system. Installers distribute the stand-alone application. Installers can include the LabVIEW Run-Time Engine, which is necessary for running stand-alone applications. However, you can also download the LabVIEW Run-Time Engine at `ni.com/downloads`.

Before you can create a stand-alone application with your VIs, you must first prepare your files for distribution. The following topics describe a few of the issues you need to consider as part of your preparation. Refer to the *Preparing Files* section of the *Building Applications Checklist* topic in the *LabVIEW Help* for more information.

VI Properties

Use the VI Properties dialog box to customize the window appearance and size. you might want to configure a VI to hide scroll bars, or you might want to hide the buttons on the toolbar.

Path Names

Consider the path names you use in the VI. Assume you read data from a file during the application, and the path to the file is hard-coded on the block diagram. Once an application is built, the file is embedded in the executable, changing the path of the file. Being aware of these issues will help you to build more robust applications in the future.

Quit LabVIEW

In a stand-alone application, the top-level VI must quit LabVIEW or close the front panel when it is finished executing. To completely quit and close the top-level VI, you must call the Quit LabVIEW function on the block diagram of the top-level VI.

External Code

Know what external code your applications uses. For example, do you call any system or custom DLLs or shared libraries? Are you going to process command line arguments? These are advanced examples that are beyond the scope of this course, but you must consider them for the application. Refer to the *Using External Code in LabVIEW* topic in the *LabVIEW Help*.

VI Server Properties and Methods

Not all VI Server properties and methods are supported in the LabVIEW Run-Time Engine. Unsupported VI server properties and methods return errors when called from the Run-Time Engine. Use proper error handling to detect any of these unsupported properties and methods. You can also

review the *VI Server Properties and Methods Not Supported in the LabVIEW Run-Time Engine* topic in the *LabVIEW Help*.

Providing Online Help in Your LabVIEW Applications

As you put the finishing touches on your application, you should provide online help to the user. To create effective documentation for VIs, create VI and object descriptions that describe the purpose of the VI or object and give users instructions for using the VI or object.

Use the following functions, located on the **Help** palette, to programmatically show or hide the **Context Help** window and link from VIs to HTML files or compiled help files:

- Use the Get Help Window Status function to return the status and position of the **Context Help** window.
- Use the Control Help Window function to show, hide, or reposition the **Context Help** window.
- Use the Control Online Help function to display the table of contents, jump to a specific topic in the file, or close the online help.
- Use the Open URL in Default Browser VI to display a URL or HTML file in the default Web browser.

B. Build Specifications

After you have prepared your files for distribution, you need to create a build specification for your application. The Build Specifications node in the Project Explorer window allows you to create and configure build specifications for LabVIEW builds. A build specification contains all the settings for the build, such as files to include, directories to create, and settings for VIs.

Note If you previously hid **Build Specifications** in the **Project Explorer** window, you must display the item again to access it in the **Project Explorer** window.

You can create and configure the following types of build specifications:

- Stand-alone applications—Use stand-alone applications to provide other users with executable versions of VIs. Applications are useful when you want users to run VIs without installing the LabVIEW development system. Stand-alone applications require the LabVIEW Run-Time Engine. **(Windows)** Applications have a `.exe` extension. **(Mac OS)** Applications have a `.app` extension.
- Installers—**(Windows)** Use installers to distribute stand-alone applications, shared libraries, and source distributions that you create with the Application Builder. Installers that include the LabVIEW

Run-Time Engine are useful if you want users to be able to run applications or use shared libraries without installing LabVIEW.

- .NET Interop Assemblies—**(Windows)** Use .NET interop assemblies to package VIs for the Microsoft .NET Framework. You must install the Microsoft .NET Framework 2.0 or higher to build a .NET interop assembly using the Application Builder.
- Shared libraries—Use shared libraries if you want to call VIs using text-based programming languages, such as LabWindows/CVI, Microsoft Visual C++, and Microsoft Visual Basic. Using shared libraries provides a way for programming languages other than LabVIEW to access code developed with LabVIEW. Shared libraries are useful when you want to share the functionality of the VIs you build with other developers. Other developers can use the shared libraries but cannot edit or view the block diagrams unless you enable debugging. **(Windows)** Shared libraries have a `.dll` extension. **(Mac OS)** Shared libraries have a `.framework` extension. **(Linux)** Shared libraries have a `.so` extension. You can use `.so` or you can begin with `lib` and end with `.so`, optionally followed by the version number. This allows other applications to use the library.
- Source distributions—Use source distributions to package a collection of source files. Source distributions are useful if you want to send code to other developers to use in LabVIEW. You can configure settings for specified VIs to add passwords, remove block diagrams, or apply other settings. You also can select different destination directories for VIs in a source distribution without breaking the links between VIs and subVIs.
- Web services (RESTful)—**(Windows)** Publish VIs within LabVIEW Web services to provide a standardized method for the LabVIEW Web Server to deploy applications that any HTTP client can access. Web services support clients across most major platforms and programming languages and allow you to easily implement and deploy Web applications over a network using LabVIEW.
- Zip files—Use zip files when you want to distribute files or an entire LabVIEW project as a single, portable file. A zip file contains compressed files, which you can send to users. Zip files are useful if you want to distribute selected source files to other LabVIEW users. You also can use the Zip VIs to create zip files programmatically.

Refer to the *Configuring Build Specifications* section of the *Building Applications Checklist* topic in the *LabVIEW Help* for more information.

C. Building the Application and Installer

System Requirements

Applications that you create with Build Specifications generally have the same system requirements as the LabVIEW development system. Memory requirements vary depending on the size of the application created.

You can distribute these files without the LabVIEW development system; however, to run stand-alone applications and shared libraries, users must have the LabVIEW Run-Time Engine installed.

Configuring Build Specifications

You must create build specifications in the **Project Explorer** window. Expand **My Computer**, right-click **Build Specifications**, select **New** and the type of build you want to configure from the shortcut menu. Use the pages in the **Source Distribution Properties**, **Application Properties**, **Shared Library Properties**, **Installer Properties**, or **Zip File Properties** dialog boxes to configure settings for the build specification. After you define these settings, click the **OK** button to close the dialog box and update the build specification in the project. The build specification appears under **Build Specifications**. Right-click a specification and select **Build** from the shortcut menu to complete the build. You also can select **Build All** from the shortcut menu to build all specifications under **Build Specifications**. If you rebuild a given specification, LabVIEW overwrites the existing files from the previous build that are part of the current build.

Refer to the *Caveats and Recommendations for Building Installers* topic in the *LabVIEW Help* for more information.

Summary

- LabVIEW features the Application Builder, which enables you to create stand-alone executables and installers. The Application Builder is available in the Professional Development Systems or as an add-on package.
- Creating a professional, stand-alone application with your VIs involves understanding the following:
  - The architecture of your VI
  - The programming issues particular to the VI
  - The application building process
  - The installer building process

Notes

Notes

A

Using Variables

In this appendix, you learn to use variables to transfer data among multiple loops and VIs. You also learn about the programming issues involved when using variables and how to overcome these challenges.

Topics

A. Parallelism

B. Variables

C. Functional Global Variables

D. Race Conditions

A. Parallelism

In this course, parallelism refers to executing multiple tasks at the same time. Consider the example of creating and displaying two sine waves at a different frequencies. Using parallelism, you place one sine wave in a loop, and the second sine wave in a different loop.

A challenge in programming parallel tasks is passing data among multiple loops without creating a data dependency. For example, if you pass the data using a wire, the loops are no longer parallel. In the multiple sine wave example, you may want to share a single stop mechanism between the loops, as shown in Figure A-1.

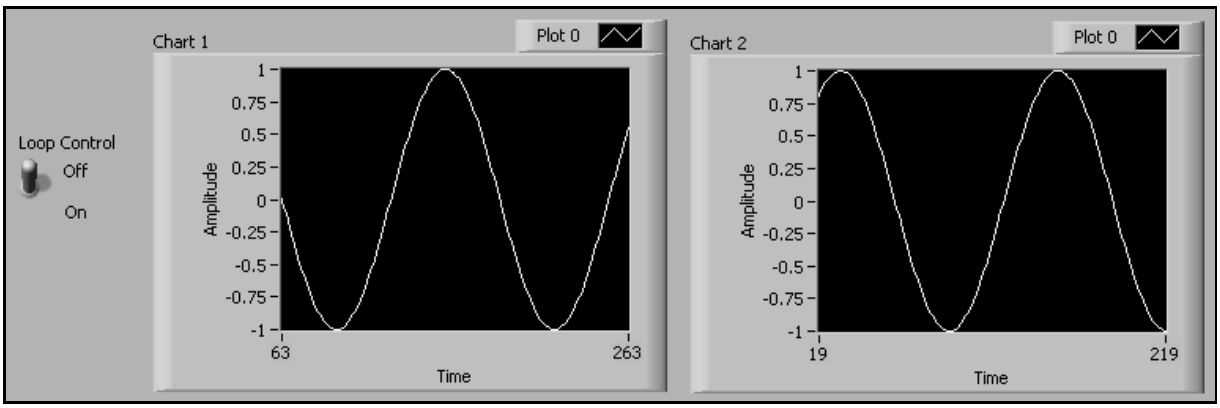


Figure A-1. Parallel Loops Front Panel

Examine what happens when you try to share data among parallel loops with a wire using those different methods.

Method 1 (Incorrect)

Place the **Loop Control** terminal outside of both loops and wire it to each conditional terminal, as shown in Figure A-2. The Loop control is a data input to both loops, therefore the **Loop Control** terminal is read only once, before either While Loop begins executing. If False is passed to the loops, the While Loops run indefinitely. Turning off the switch does not stop the VI because the switch is not read during the iteration of either loop.

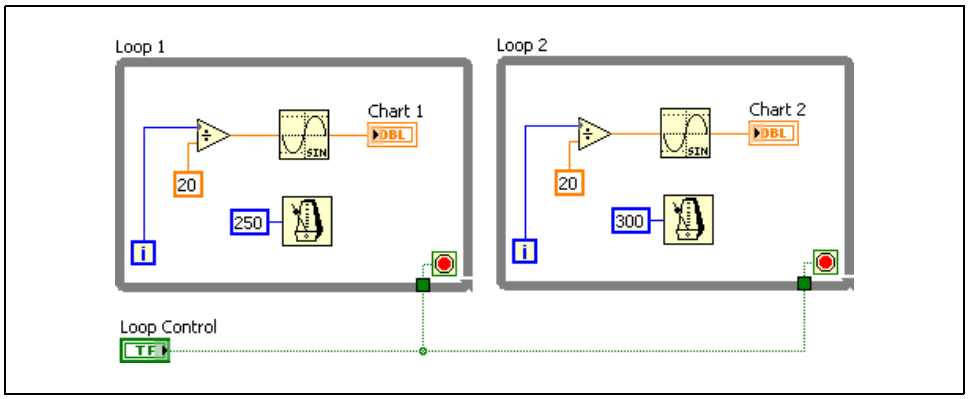


Figure A-2. Parallel Loops Method 1 Example

Method 2 (Incorrect)

Move the **Loop Control** terminal inside Loop 1 so that it is read in each iteration of Loop 1, as shown in the following block diagram. Although Loop 1 terminates properly, Loop 2 does not execute until it receives all its data inputs. Loop 1 does not pass data out of the loop until the loop stops, so Loop 2 must wait for the final value of the **Loop Control**, available only after Loop 1 finishes. Therefore, the loops do not execute in parallel. Also, Loop 2 executes for only one iteration because its conditional terminal receives a True value from the **Loop Control** switch in Loop 1.

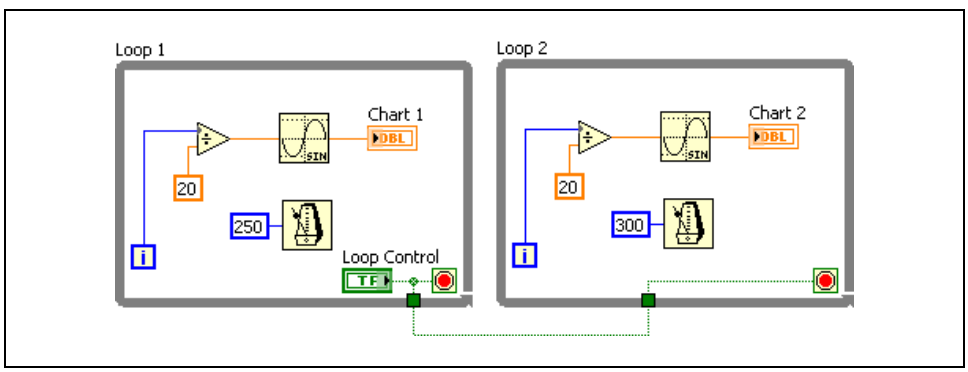


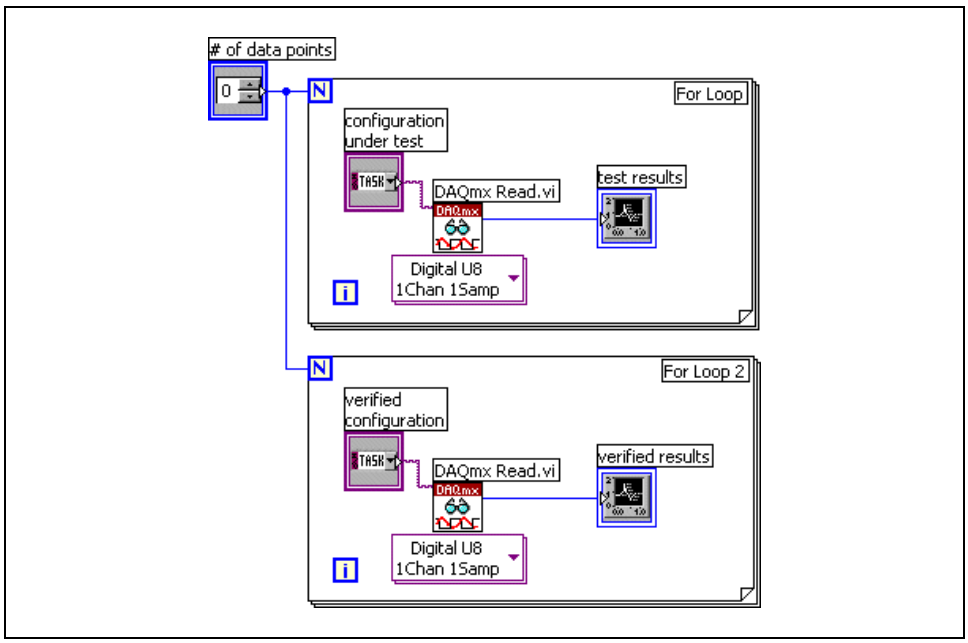
Figure A-3. Parallel Loops Method 2 Example

Method 3 (Solution)

If you could read the value of the loop control from a file, you would no longer have a dataflow dependency between the loops, as each loop can independently access the file. However, reading and writing to files can be time consuming, at least in processor time. Another way to accomplish this task is to find the location where the loop control data is stored in memory and read that memory location directly. The rest of this lesson will provide information on methods for solving this problem.

B. Variables

In LabVIEW, the flow of data rather than the sequential order of commands determines the execution order of block diagram elements. Therefore, you can create block diagrams that have simultaneous operations. For example, you can run two For Loops simultaneously and display the results on the front panel, as shown in the following block diagram.



However, if you use wires to pass data between parallel block diagrams, they no longer operate in parallel. Parallel block diagrams can be two parallel loops on the same block diagram without any data flow dependency, or two separate VIs that are called at the same time.

The block diagram in Figure A-4 does not run the two loops in parallel because of the wire between the two subVIs.

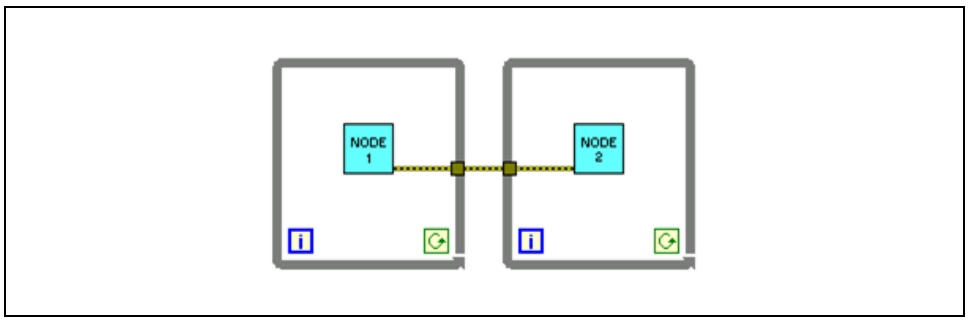


Figure A-4. Data Dependency Imposed by Wire

The wire creates a data dependency, because the second loop does not start until the first loop finishes and passes the data through its tunnel. To make the two loops run concurrently, remove the wire. To pass data between the subVIs, use another technique, such as a variable.

In LabVIEW, *variables* are block diagram elements that allow you to access or store data in another location. The actual location of the data varies depending on the type of the variable. Local variables store data in front panel controls and indicators. Global variables and single-process shared variables store data in special repositories that you can access from multiple VIs. Functional global variables store data in While Loop shift registers. Regardless of where the variable stores data, all variables allow you to circumvent normal data flow by passing data from one place to another without connecting the two places with a wire. For this reason, variables are useful in parallel architectures, but also have certain drawbacks, such as race conditions.

Using Variables in a Single VI

Local variables transfer data within a single VI.

In LabVIEW, you read data from or write data to a front panel object using its block diagram terminal. However, a front panel object has only one block diagram terminal, and your application might need to access the data in that terminal from more than one location.

Local and global variables pass information between locations in the application that you cannot connect with a wire. Use local variables to access front panel objects from more than one location in a single VI. Use global variables to access and pass data among several VIs.

Use a Feedback Node to store data from a previous VI or loop execution.

Creating Local Variables

Right-click an existing front panel object or block diagram terminal and select **Create»Local Variable** from the shortcut menu to create a local variable. A local variable icon for the object appears on the block diagram.

You also can select a local variable from the **Functions** palette and place it on the block diagram. The local variable node is not yet associated with a control or indicator.

To associate a local variable with a control or indicator, right-click the local variable node and select **Select Item** from the shortcut menu. The expanded shortcut menu lists all the front panel objects that have owned labels.

LabVIEW uses owned labels to associate local variables with front panel objects, so label the front panel controls and indicators with descriptive owned labels.

Reading and Writing to Variables

After you create a variable, you can read data from a variable or write data to it. By default, a new variable receives data. This kind of variable works as an indicator and is a write local or global. When you write new data to the local or global variable, the associated front panel control or indicator updates to the new data.

You also can configure a variable to behave as a data source, or a variable. Right-click the variable and select **Change To Read** from the shortcut menu to configure the variable to behave as a control. When this node executes, the VI reads the data in the associated front panel control or indicator.

To change the variable to receive data from the block diagram rather than provide data, right-click the variable and select **Change To Write** from the shortcut menu.

On the block diagram, you can distinguish read variables from write variables the same way you distinguish controls from indicators. A read variable has a thick border similar to a control. A write variable has a thin border similar to an indicator.

Local Variable Example

In the *Parallelism* section of this lesson, you saw an example of a VI that used parallel loops. The front panel contained a single switch that stopped the data generation displayed on two graphs. On the block diagram, the data for each chart is generated within an individual While Loop to allow for separate timing of each loop. The Loop Control terminal stopped both While Loops. In this example, the two loops must share the switch to stop both loops at the same time.

For both charts to update as expected, the While Loops must operate in parallel. Connecting a wire between While Loops to pass the switch data makes the While Loops execute serially, rather than in parallel. Figure A-5 shows a block diagram of this VI using a local variable to pass the switch data.

Loop 2 reads a local variable associated with the switch. When you set the switch to False on the front panel, the switch terminal in Loop 1 writes a False value to the conditional terminal in Loop 1. Loop 2 reads the **Loop Control** local variable and writes a False to the Loop 2 conditional terminal. Thus, the loops run in parallel and terminate simultaneously when you turn off the single front panel switch.

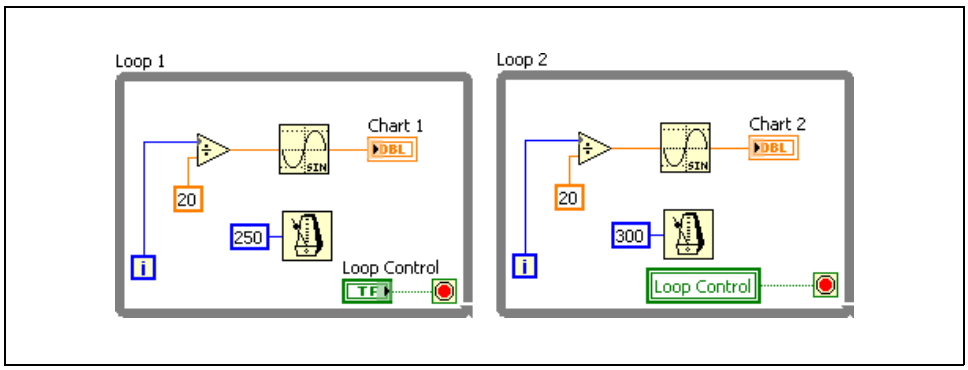


Figure A-5. Local Variable Used to Stop Parallel Loops

With a local variable, you can write to or read from a control or indicator on the front panel. Writing to a local variable is similar to passing data to any other terminal. However, with a local variable you can write to it even if it is a control or read from it even if it is an indicator. In effect, with a local variable, you can access a front panel object as both an input and an output.

For example, if the user interface requires users to log in, you can clear the **`Login`** and **`Password`** prompts each time a new user logs in. Use a local variable to read from the **Login** and **Password** string controls when a user logs in and to write empty strings to these controls when the user logs out.

Using Variables Among VIs

You also can use variables to access and pass data among several VIs that run simultaneously. A local variable shares data within a VI. A global variable also shares data, but it shares data among multiple VIs. For example, suppose you have two VIs running simultaneously. Each VI contains a While Loop and writes data points to a waveform chart. The first VI contains a Boolean control to terminate both VIs. You can use a global variable to terminate both loops with a single Boolean control. If both loops were on a single block diagram within the same VI, you could use a local variable to terminate the loops.

You also can use a single-process shared variable in the same way you use a global variable. A shared variable is similar to a local variable or a global variable, but allows you to share data across a network. A shared variable can be single-process or network-published. Although network-published shared variables are beyond the scope of this course, by using the single-process shared variable, you can later change to a network-published shared variable.

Use a global variable to share data among VIs on the same computer, especially if you do not use a project file. Use a single-process shared variable if you may need to share the variable information among VIs on multiple computers in the future.

Creating Global Variables

Use global variables to access and pass data among several VIs that run simultaneously. Global variables are built-in LabVIEW objects. When you create a global variable, LabVIEW automatically creates a special global VI, which has a front panel but no block diagram. Add controls and indicators to the front panel of the global VI to define the data types of the global variables it contains. In effect, this front panel is a container from which several VIs can access data.

For example, suppose you have two VIs running simultaneously. Each VI contains a While Loop and writes data points to a waveform chart. The first VI contains a Boolean control to terminate both VIs. You must use a global variable to terminate both loops with a single Boolean control. If both loops were on a single block diagram within the same VI, you could use a local variable to terminate the loops.

Select a global variable from the **Functions** palette and place it on the block diagram.

Double-click the global variable node to display the front panel of the global VI. Place controls and indicators on this front panel the same way you do on a standard front panel.

LabVIEW uses owned labels to identify global variables, so label the front panel controls and indicators with descriptive owned labels.

You can create several single global VIs, each with one front panel object, or if you want to group similar variables together, you can create one global VI with multiple front panel objects.

You can create several single global variables, each with one front panel object, or you can create one global variable with multiple front panel objects.

A global variable with multiple objects is more efficient because you can group related variables together. The block diagram of a VI can include several global variable nodes that are associated with controls and indicators on the front panel of a global variable. These global variable nodes are either copies of the first global variable node that you placed on the block diagram of the global VI, or they are the global variable nodes of global VIs that you placed on the current VI. You place global VIs on other VIs the same way you place subVIs on other VIs. Each time you place a new global variable node on a block diagram, LabVIEW creates a new VI associated only with that global variable node and copies of it.

Figure A-6 shows a global variable front panel window with a numeric, a string, and a cluster containing a numeric and a Boolean control. The toolbar does not show the **Run**, **Stop**, or related buttons as a normal front panel window.

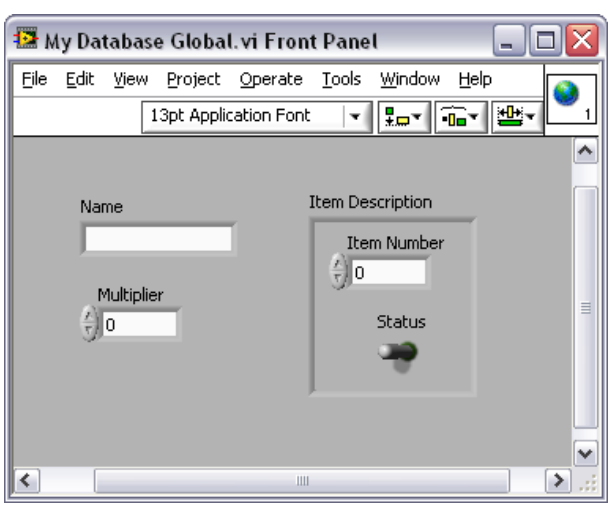


Figure A-6. Global Variable Front Panel Window

After you finish placing objects on the global VI front panel, save it and return to the block diagram of the original VI. You must then select the object in the global VI that you want to access. Click the global variable node and select a front panel object from the shortcut menu. The shortcut

menu lists all the front panel objects in the global VI that have owned labels. You also can right-click the global variable node and select a front panel object from the **Select Item** shortcut menu.

You also can use the Operating tool or Labeling tool to click the global variable node and select the front panel object from the shortcut menu.

If you want to use this global variable in other VIs, select the **Select a VI** option on the **Functions** palette. By default, the global variable is associated with the first front panel object with an owned label that you placed in the global VI. Right-click the global variable node you placed on the block diagram and select a front panel object from the **Select Item** shortcut menu to associate the global variable with the data from another front panel object.

Creating Single-Process Shared Variables

You must use a project file to use a shared variable. To create a single-process shared variable, right-click **My Computer** in the **Project Explorer** window and select **New»Variable**. The **Shared Variable Properties** dialog box appears, as shown in Figure A-7.

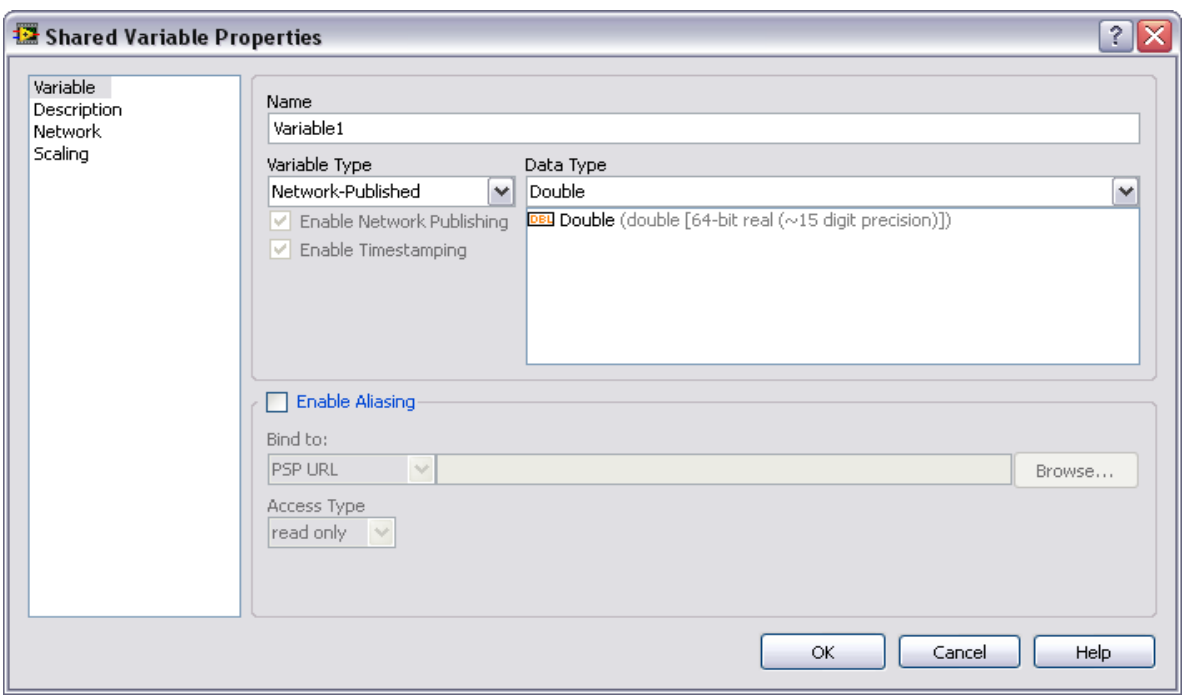


Figure A-7. Shared Variable Properties Dialog Box

Under **Variable Type**, select **Single Process**. Give the variable a name and a data type. After you create the shared variable, it automatically appears in a new library in your project file. Save the library. You can add additional shared variables to this library as needed. You can drag and drop the variable from the listing in the **Project Explorer** window directly to the block

diagram. Use the short-cut menu to switch between writing or reading. Use the error clusters on the variable to impose data flow.

Using Variables Carefully

Local and global variables are advanced LabVIEW concepts. They are inherently not part of the LabVIEW dataflow execution model. Block diagrams can become difficult to read when you use local and global variables, so you should use them carefully. Misusing local and global variables, such as using them instead of a connector pane or using them to access values in each frame of a sequence structure, can lead to unexpected behavior in VIs. Overusing local and global variables, such as using them to avoid long wires across the block diagram or using them instead of data flow, slows performance.

Variables often are used unnecessarily. The example in Figure A-8 shows a traffic light application implemented as a state machine. Each state updates the lights for the next stage of the light sequence. In the state shown, the east and west traffic has a green light, while the north and south traffic has a red light. This stage waits for 4 seconds, as shown by the Wait (ms) function.

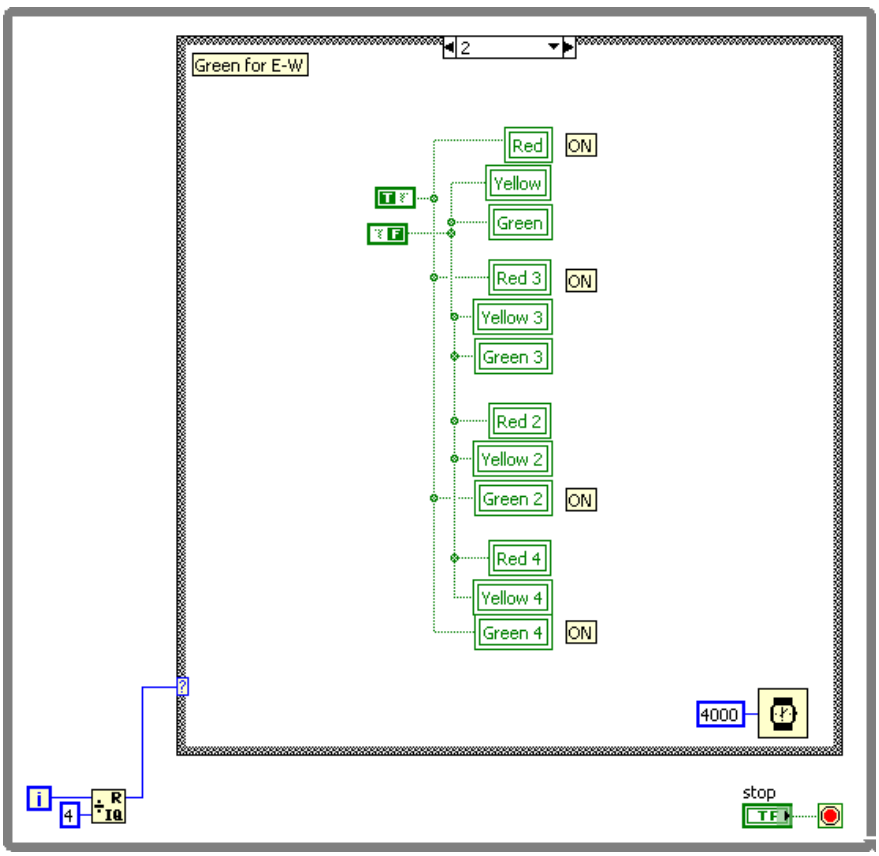


Figure A-8. Too Many Variables Used

The example shown in Figure A-9 accomplishes the same task, but more efficiently and using a better design. Notice that this example is much easier to read and understand than the previous example, mostly by reducing variable use. By placing the indicators in the While Loop outside the Case structure, the indicators can update after every state without using a variable. This example is less difficult to modify for further functionality, such as adding left turn signals, than the previous example.

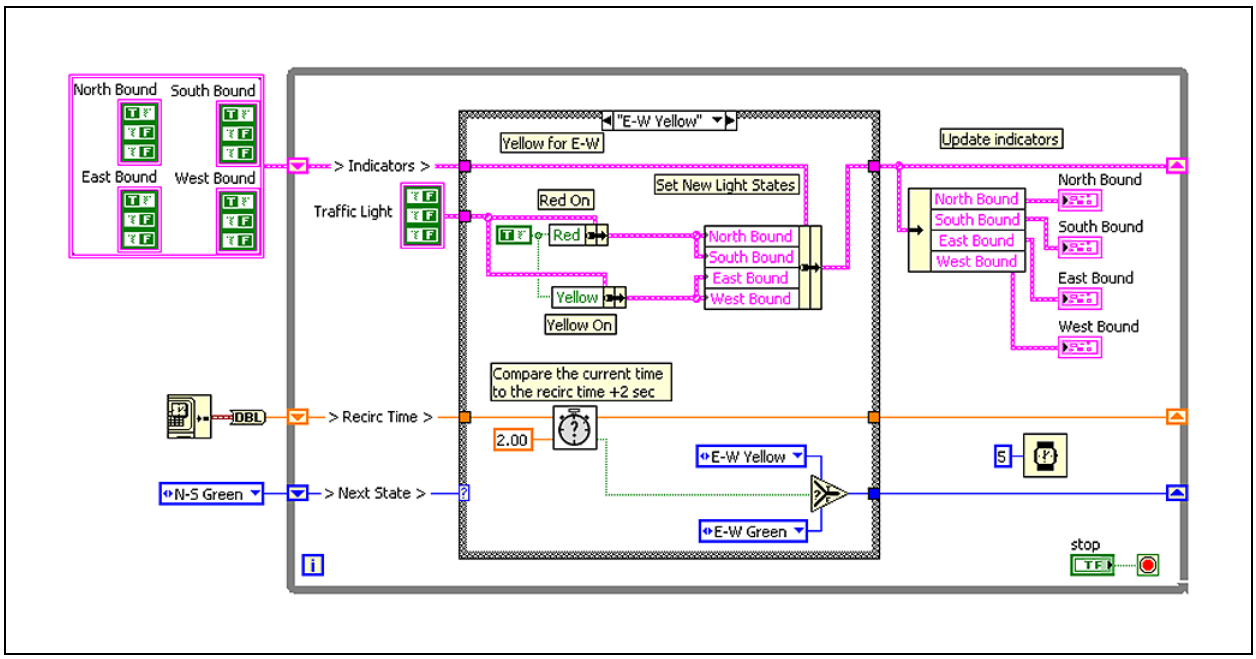


Figure A-9. Reduced Variables

Initializing Variables

To initialize a local or global variable, verify that the variable contains known data values before the VI runs. Otherwise, the variables might contain data that causes the VI to behave incorrectly. If the variable relies on a computation result for the initial value, make sure LabVIEW writes the value to the variable before it attempts to access the variable for any other action. Wiring the write action in parallel with the rest of the VI can cause a race condition.

To make sure it executes first, you can isolate the code that writes the initial value for the variable to the first frame of a sequence structure or to a subVI and wire the subVI to execute first in the data flow of the block diagram.

If you do not initialize the variable before the VI reads the variable for the first time, the variable contains the default value of the associated front panel object.

Figure A-10 shows a common mistake when using variables. A shared variable synchronizes the stop conditions for two loops. This example operates the first time it runs, because the default value of a Boolean is False. However, each time this VI runs, the **Stop** control writes a True value to the variable. Therefore, the second and subsequent times that this VI runs, the lower loop stops after only a single iteration unless the first loop updates the variable quickly enough.

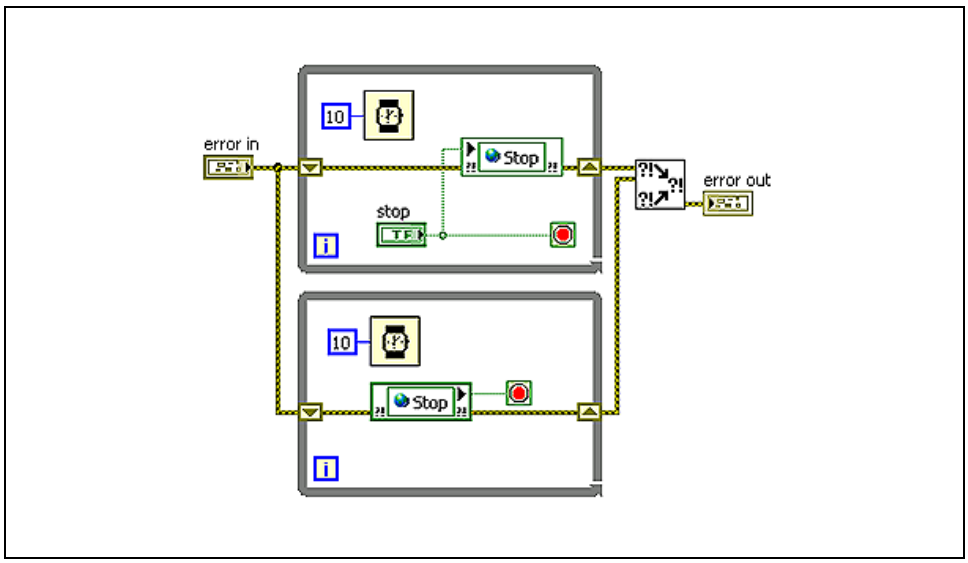


Figure A-10. Failing to Initialize a Shared Variable

Figure A-11 shows the VI with code added to initialize the shared variable. Initialize the variable before the loops begin to insure that the second loop does not immediately stop.

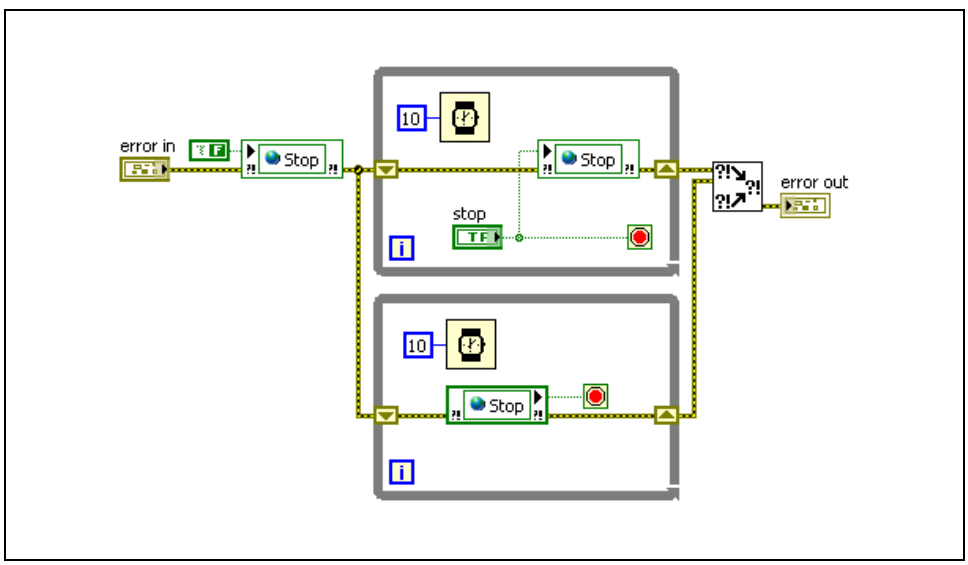
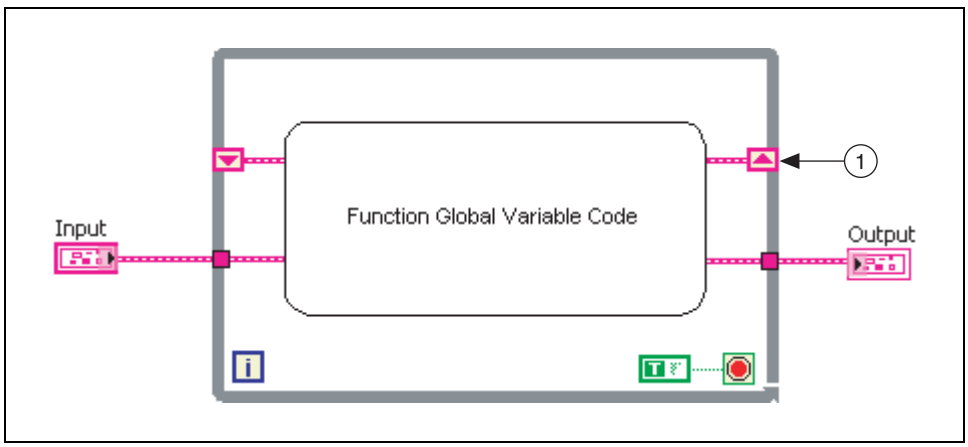


Figure A-11. Initializing a Shared Variable Properly

C. Functional Global Variables

You can use uninitialized shift registers in For Loops or While Loops to store data as long as the VI is in memory. The shift register holds the last state of the shift register. Place a While Loop within a subVI and use the shift registers to store data that can be read from or written to. Using this technique is similar to using a global variable. This method is often called a functional global variable. The advantage to this method over a global variable is that you can control access to the data in the shift register. The general form of a functional global variable includes an uninitialized shift register with a single iteration For Loop or While Loop, as shown in Figure A-12.



1 Uninitialized Shift Register

Figure A-12. Functional Global Variable Format

A functional global variable usually has an **action** input parameter that specifies which task the VI performs. The VI uses an uninitialized shift register in a While Loop to hold the result of the operation.

Figure A-13 shows a simple functional global variable with set and get functionality.

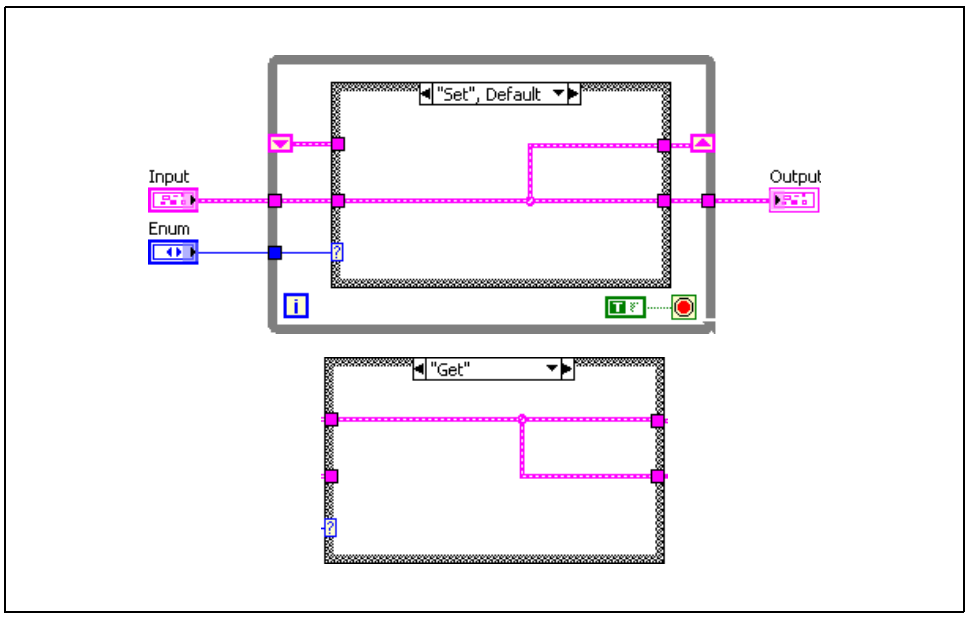


Figure A-13. Functional Global Variable with Set and Get Functionality

In this example, data passes into the VI and the shift register stores the data if you configure the enumerated data type to Set. Data is retrieved from the shift register if the enumerated data type is configured to Get.

Although you can use functional global variables to implement simple global variables, as shown in the previous example, they are especially useful when implementing more complex data structures, such as a stack or a queue buffer. You also can use functional global variables to protect access to global resources, such as files, instruments, and data acquisition devices, that you cannot represent with a global variable.

Note A functional global variable is a subVI that is not reentrant. This means that when the subVI is called from multiple locations, the same copy of the subVI is used. Therefore, only one call to the subVI can occur at a time.

Using Functional Global Variables for Timing

One powerful application of functional global variables is to perform timing in your VI. Many VIs that perform measurement and automation require some form of timing. Often an instrument or hardware device needs time to initialize, and you must build explicit timing into your VI to take into account the physical time required to initialize a system. You can create a functional global variable that measures the elapsed time between each time the VI is called, as shown in Figure A-14.

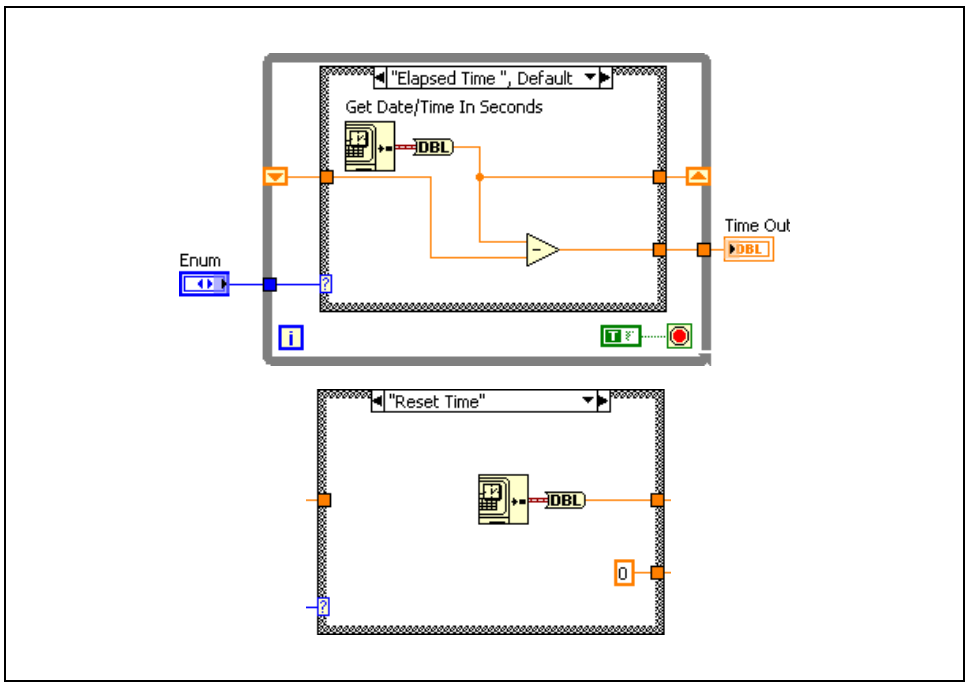


Figure A-14. Elapsed Time Functional Global Variable

The Elapsed Time case gets the current date and time in seconds and subtracts it from the time that is stored in the shift register. The Reset Time case initializes the functional global variable with a known time value.

The Elapsed Time Express VI implements the same functionality as this functional global variable. The benefit of using the functional global variable is that you can customize the implementation easily, such as adding a pause option.

D. Race Conditions

A race condition occurs when the timing of events or the scheduling of tasks unintentionally affects an output or data value. Race conditions are a common problem for programs that execute multiple tasks in parallel and share data between them. Consider the following example in Figure A-15 and Figure A-16.

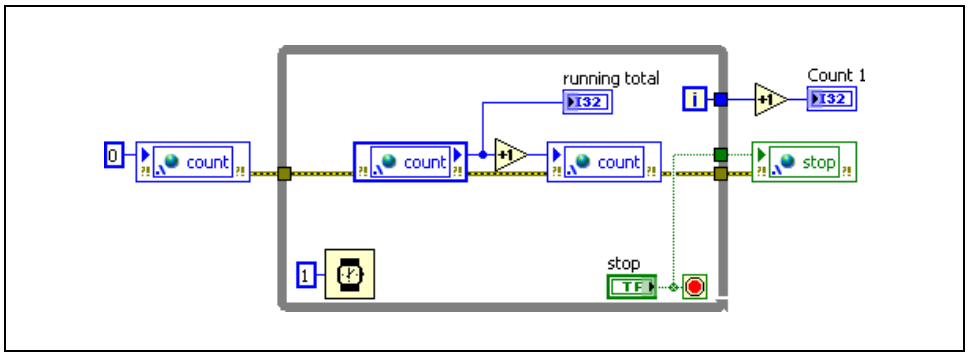


Figure A-15. Race Condition Example: Loop 1

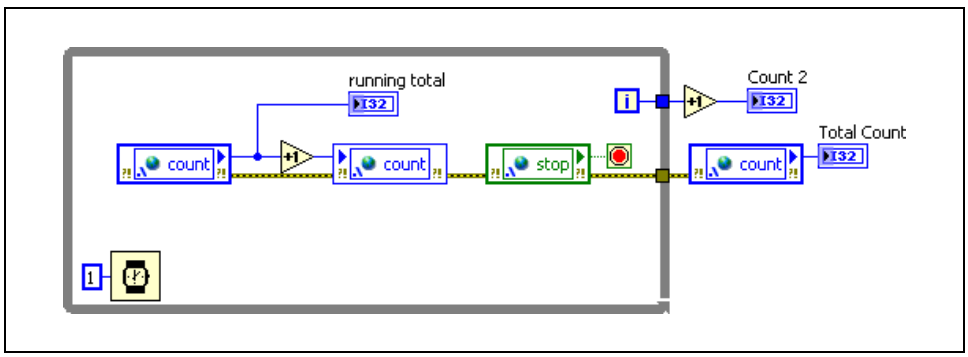


Figure A-16. Race Condition Example: Loop 2

The two loops both increment a shared variable during each iteration. If you run this VI, the expected result after clicking the **Stop** button is that the **Total Count** is equal to the sum of **Count 1** and **Count 2**. If you run the VI for a short period of time, you generally see the expected result. However, if you run the VI for a longer period of time, the **Total Count** is less than the sum of **Count 1** and **Count 2**, because this VI contains a race condition.

On a single processor computer, actions in a multi-tasking program like this example actually happen sequentially, but LabVIEW and the operating system rapidly switch tasks so that the tasks effectively execute at the same time. The race condition in this example occurs when the switch from one task to the other occurs at a certain time. Notice that both of the loops perform the following operations:

- Read the shared variable.
- Increment the value read.
- Write the incremented value to the shared variable.

Now consider what happens if the loop operations happen to be scheduled in the following order:

1. Loop 1 reads the shared variable.
2. Loop 2 reads the shared variable.
3. Loop 1 increments the value it read.
4. Loop 2 increments the value it read.
5. Loop 1 writes the incremented value to the shared variable.
6. Loop 2 writes the incremented value to the shared variable.

In this example, both loops write the same value to the variable, and the increment of the first loop is effectively overwritten by Loop 2. This generates a race condition, which can cause serious problems if you intend the program to calculate an exact count.

In this particular example, there are few instructions between when the shared variable is read and when it is written. Therefore, the VI is less likely to switch between the loops at the wrong time. This explains why this VI runs accurately for short periods and only loses a few counts for longer periods.

Race conditions are difficult to identify and debug, because the outcome depends upon the order in which the operating system executes scheduled tasks and the timing of external events. The way tasks interact with each other and the operating system, as well as the arbitrary timing of external events, make this order essentially random. Often, code with a race condition can return the same result thousands of times in testing, but still can return a different result, which can appear when the code is in use.

The best way to avoid race conditions is by using the following techniques:

- Controlling and limiting shared resources.
- Identifying and protecting critical sections within your code.
- Specifying execution order.

Controlling and Limiting Shared Resources

Race conditions are most common when two tasks have both read and write access to a resource, as is the case in the previous example. A resource is any entity that is shared between the processes. When dealing with race conditions, the most common shared resources are data storage, such as variables. Other examples of resources include files and references to hardware resources.

Allowing a resource to be altered from multiple locations often introduces the possibility for a race condition. Therefore, an ideal way to avoid race conditions is to minimize shared resources and the number of writers to the remaining shared resources. In general, it is not harmful to have multiple readers or monitors for a shared resource. However, try to use only one writer or controller for a shared resource. Most race conditions only occur when a resource has multiple writers.

In the previous example, you can reduce the dependency upon shared resources by having each loop maintain its count locally. Then, share the final counts after clicking the **Stop** button. This involves only a single read and a single write to a shared resource and eliminates the possibility of a race condition. If all shared resources have only a single writer or controller, and the VI has a well sequenced instruction order, then race conditions do not occur.

Protecting Critical Sections

A critical section of code is code that must behave consistently in all circumstances. When you use multi-tasking programs, one task may interrupt another task as it is running. In nearly all modern operating systems, this happens constantly. Normally, this does not have any effect upon running code, however, when the interrupting task alters a shared resource that the interrupted task assumes is constant, then a race condition occurs.

Figure A-15 and Figure A-16 contain critical code sections. If one of the loops interrupts the other loop while it is executing the code in its critical section, then a race condition can occur. One way to eliminate race conditions is to identify and protect the critical sections in your code. There are many techniques for protecting critical sections. Two of the most effective are functional global variables and semaphores.

Functional Global Variables

One way to protect critical sections is to place them in subVIs. You can only call a non-reentrant subVI from one location at a time. Therefore, placing critical code in a non-reentrant subVI keeps the code from being interrupted by other processes calling the subVI. Using the functional global variable architecture to protect critical sections is particularly effective, because shift registers can replace less protected storage methods like global or single-process shared variables. Functional global variables also encourage the creation of multi-functional subVIs that handle all tasks associated with a particular resource.

After you identify each section of critical code in your VI, group the sections by the resources they access, and create one functional global variable for each resource. Critical sections performing different operations each can become a command for the functional global variable, and you can group critical sections that perform the same operation into one command, thereby re-using code.

You can use functional global variables to protect critical sections of code in Figure A-15 and Figure A-16. To remove the race condition, replace the shared variables with a functional global variable and place the code to increment the counter within the functional global variable, as shown in Figure A-17, Figure A-18, and Figure A-19.

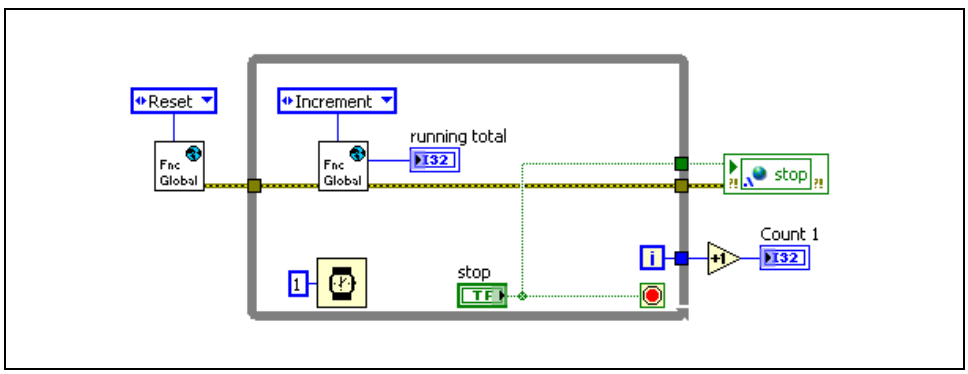


Figure A-17. Using Functional Global Variables to Protect the Critical Section in Loop 1

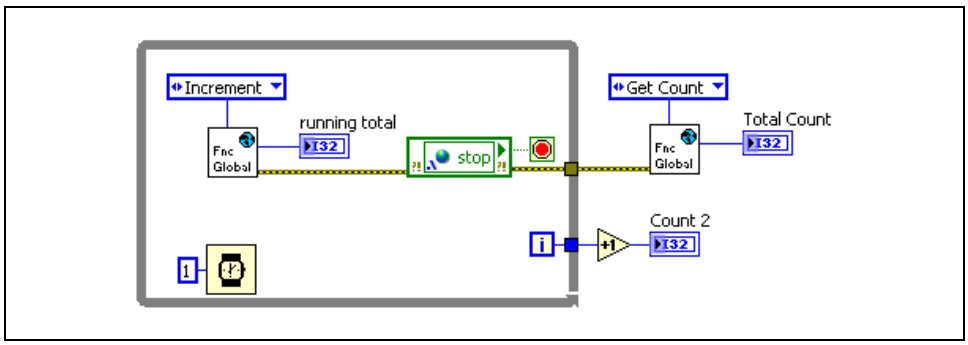


Figure A-18. Using Functional Global Variables to Protect the Critical Section in Loop 2

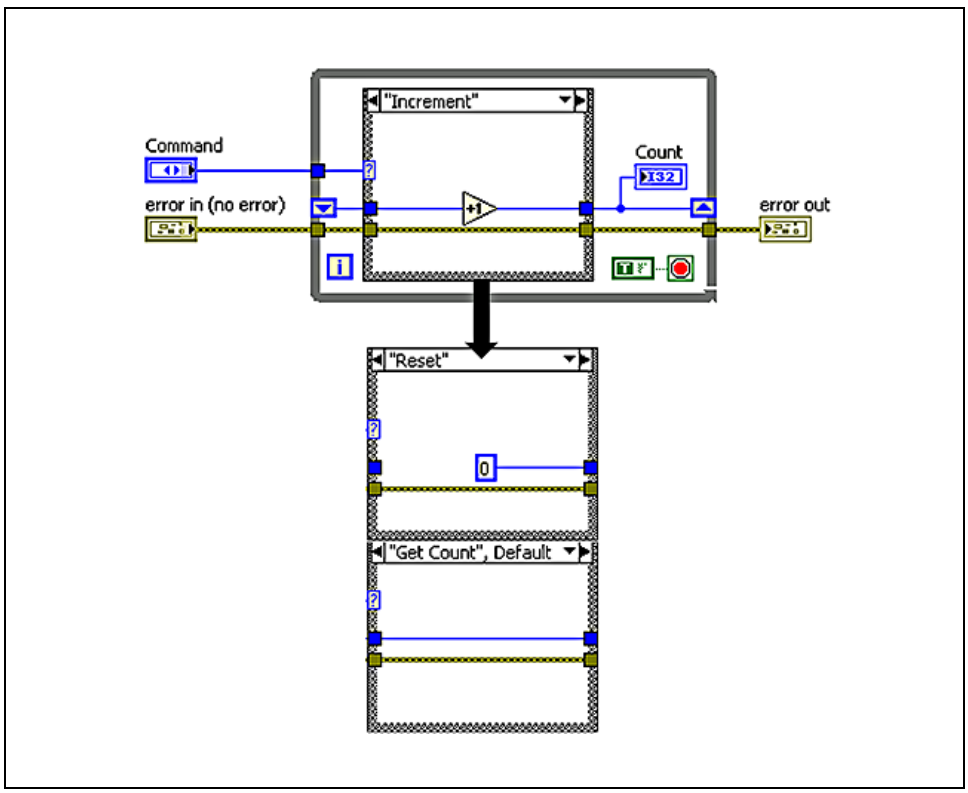


Figure A-19. Functional Global Variable Eliminates the Race Condition

Semaphores

Semaphores are synchronization mechanisms specifically designed to protect resources and critical sections of code. You can prevent critical sections of code from interrupting each other by enclosing each between an Acquire Semaphore and Release Semaphore VI. By default, a semaphore only allows one task to acquire it at a time. Therefore, after one of the tasks enters a critical section, the other tasks cannot enter their critical sections until the first task completes. When done properly, this eliminates the possibility of a race condition.

You can use semaphores to protect the critical sections of the VIs, as shown in Figure A-15 and Figure A-16. A named semaphore allows you to share the semaphore between VIs. You must open the semaphore in each VI, then acquire it just before the critical section and release it after the critical section. Figure A-20 and Figure A-21 show a solution to the race condition using semaphores.

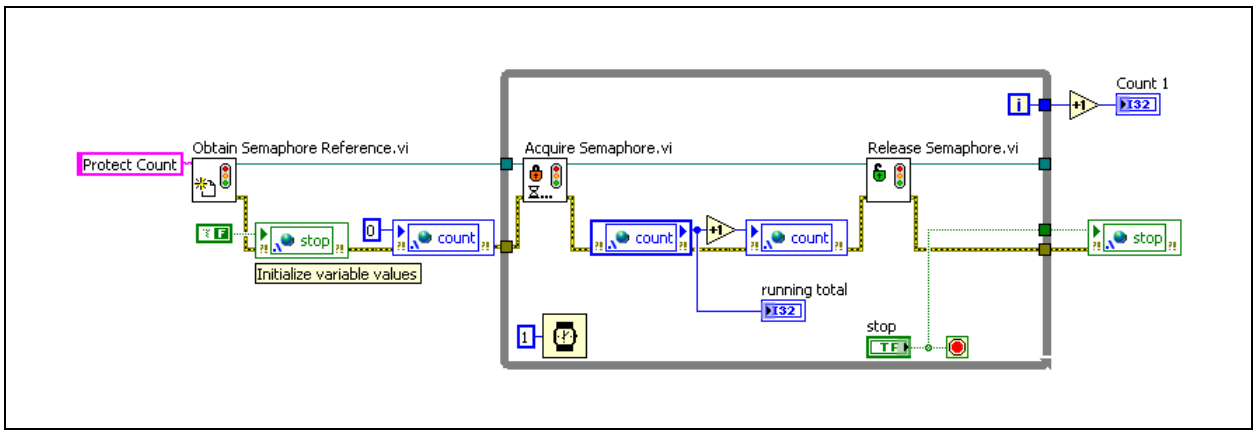


Figure A-20. Protecting the Critical Section with a Semaphore in Loop 1

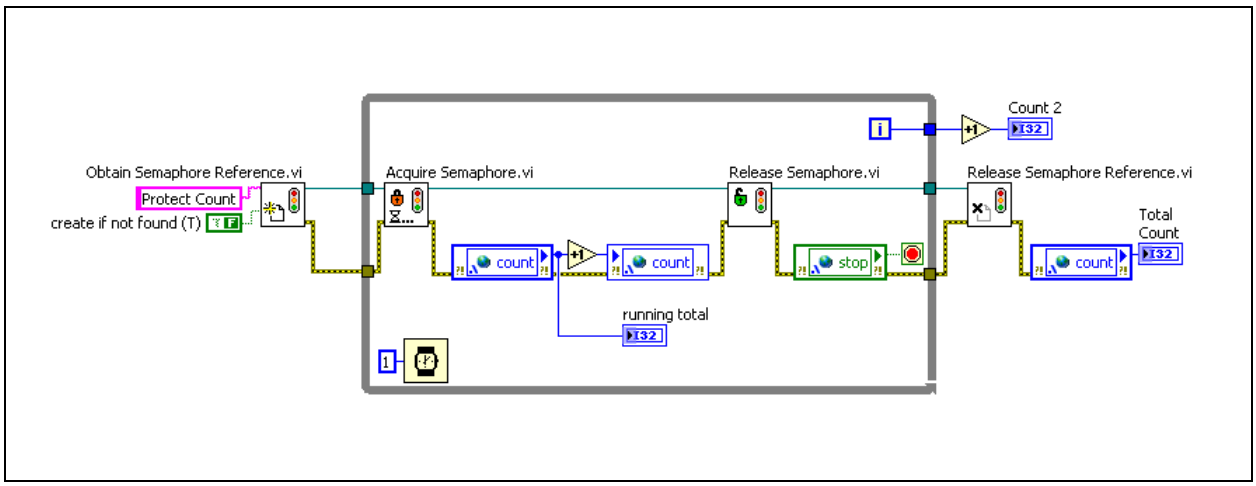


Figure A-21. Protecting the Critical Section with a Semaphore in Loop 2

Specifying Execution Order

Code in which data flow is not properly used to control the execution order can cause some race conditions. When a data dependency is not established, LabVIEW can schedule tasks in any order, which creates the possibility for race conditions if the tasks depend upon each other. Consider the example in Figure A-22.

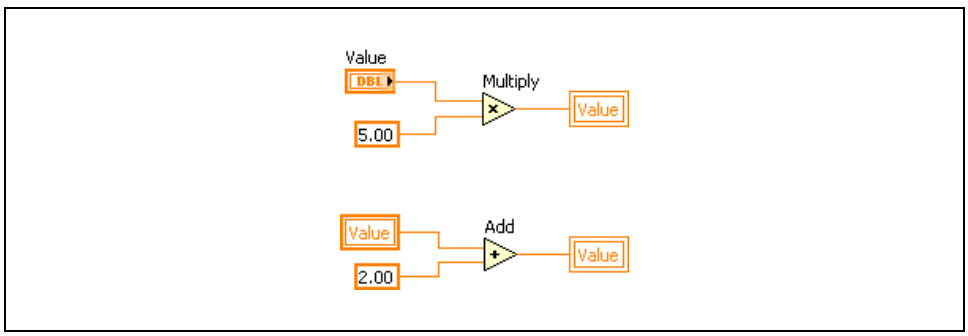


Figure A-22. Simple Race Condition

The code in this example has four possible outcomes, depending on the order in which the operations execute.

Outcome 1: Value = (Value × 5) + 2

1. Terminal reads Value.
2. Value × 5 is stored in Value.
3. Local variable reads Value × 5.
4. (Value × 5) + 2 is stored in Value.

Outcome 2: Value = (Value + 2) × 5

1. Local variable reads Value.
2. Value + 2 is stored in Value.
3. Terminal reads Value+2.
4. (Value + 2) × 5 is stored in Value.

Outcome 3: Value = Value × 5

1. Terminal reads Value.
2. Local variable reads Value.
3. Value + 2 is stored in Value.
4. Value × 5 is stored in Value.

Outcome 4: Value = Value + 2

1. Terminal reads Value.
2. Local variable reads Value.
3. Value × 5 is stored in Value.
4. Value + 2 is stored in Value.

Although this code is considered a race condition, the code generally behaves less randomly than the first race condition example because LabVIEW usually assigns a consistent order to the operations. However, you should avoid situations such as this one because the order and the behavior of the VI can vary. For example, the order could change when running the VI under different conditions or when upgrading the VI to a newer version of LabVIEW. Fortunately, race conditions of this nature are easily remedied by controlling the data flow.

Self-Review: Quiz

1. You should use variables frequently in your VIs.
   a. True
   b. False

2. Which of the following cannot transfer data?
   a. Semaphores
   b. Functional global variables
   c. Local variables
   d. Single process shared variables

3. Which of the following must be used within a project?
   a. Local variable
   b. Global variable
   c. Functional global variable
   d. Single-process shared variable

4. Which of the following cannot be used to pass data between multiple VIs?
   a. Local variable
   b. Global variable
   c. Functional global variable
   d. Single-process shared variable

Self-Review: Quiz Answers

1. You should use variables frequently in your VI.
   a. True
   b. False

   You should use variables only when necessary. Use wires to transfer data whenever possible.

2. Which of the following cannot transfer data?
   a. Semaphores
   b. Functional global variables
   c. Local variables
   d. Single process shared variables

3. Which of the following must be used within a project?
   a. Local variable
   b. Global variable
   c. Functional global variable
   d. Single-process shared variable

4. Which of the following cannot be used to pass data between multiple VIs?
   a. Local variable
   b. Global variable
   c. Functional global variable
   d. Single-process shared variable

Notes