CompactRIO Developers Guide

Recommended LabVIEW Architectures and Development Practices for Machine Control Applications

This document provides an overview of recommended architectures and development practices when building machine control applications using NI CompactRIO controllers and NI touch panel computers.
The examples and architectures are built using NI LabVIEW Version 8.6.

May 2009

CONTENTS

CHAPTER 4

CHAPTER 1
Overview and Background

Developers Guide Overview

This document provides architectures and recommendations you can use to build control applications on NI CompactRIO controllers running the NI LabVIEW Real-Time Module Version 8.6 or later. It explains how you can use new features for CompactRIO – such as the NI Scan Engine, Fault Engine, and Distributed System Manager – that were introduced in LabVIEW 8.6. CompactRIO has built-in components to make control applications easier to design; however, the same basic architecture also works on other platforms such as Compact FieldPoint, PXI, and Windows-based controllers. LabVIEW Real-Time is a full programming language that provides developers numerous ways to construct a controller and helps them create very flexible and complex systems. LabVIEW Real-Time controllers are used in applications ranging from the control of nuclear power plant rods, to hardware-in-the-loop testing for engine electronic control units (ECUs), to adaptive control for oil well drilling, to high-speed vibration monitoring for predictive maintenance. This document is designed to offer a framework for engineers designing industrial control applications, especially engineers who are familiar with programmable logic controllers (PLCs), and is intended as a complementary guide to standard LabVIEW Real-Time training. Use this guide to learn how to construct LabVIEW Real-Time applications that incorporate not only the features common on PLCs but also the flexibility to handle nontraditional applications such as high-speed buffered I/O, data logging, or machine vision.

Terminology

You can use LabVIEW Real-Time to build a control application in a variety of ways as long as you understand the following fundamental concepts of real-time programming and control applications.

- *Responsiveness* – A control application needs to react to an event such as an I/O change, human machine interface (HMI) input, or internal state change. The time required to take action after an event is known as responsiveness, and different control applications have different tolerances for responsiveness, varying from microseconds to minutes. Most industrial applications have responsiveness requirements in the milliseconds to seconds range. An important design criterion for a control application is the required responsiveness because this determines the control loop rates and affects your I/O, processor, and software decisions.

- *Determinism and jitter* – Determinism is the repeatability of the timing of a control loop. Jitter, the error in timing, is how you measure determinism. For example, if a loop is set to run and update outputs once every 50 ms, but it sometimes runs at 50.5 ms, then the jitter is 0.5 ms Increased determinism and reliability are the primary advantages of a real-time control system, and good determinism is critical for stable control applications. Low determinism leads to poor analog control and can make a system unresponsive.

- *Priority* – Most controllers use a single processor to handle all control, monitoring, and communication tasks. Because there is a single resource (processor) with multiple parallel demands, you need a way to manage the demands that are most important. By setting critical control loops to a high priority, you can have a full-featured controller that still exhibits good determinism and responsiveness. For instance, in an application with a temperature control loop and embedded logging functionality, you can set the control loop to a high priority to preempt the logging operation and provide deterministic temperature control. This ensures that lower-priority tasks, such as logging, a Web server, HMI, and so on, do not negatively affect analog controls or digital logic.

Machine Control Architecture Overview

Machine control systems typically incorporate an HMI and a real-time control system. Real-time controllers offer reliable, predictable machine behavior, while HMIs provide the machine operator a graphical user interface (GUI) for monitoring the machine's state and setting its operating parameters. In a typical machine control system, you implement the control system using a controller based on a programmable logic controller (PLC) or a programmable automation controller (PAC). Baseline controller functionality includes the following:

- Analog and digital I/O
- A memory table for sharing I/O and variable (tag) values
- A sequencing engine that defines the machine behavior

In addition to these PLC-class capabilities, National Instruments PACs can support more sophisticated functionality such as the following:

- High-speed data acquisition and analysis
- Motion control
- Vision/inspection
- Custom hardware-based signal processing
- Data logging

You can program HMIs on a PC running Windows or a touch panel computer running an embedded OS such as Windows XP Embedded. HMI features typically include the following:

- Touch screen operation
- A paged display system with navigation controls
- Data entry objects (buttons, keypads, and so on)
- Alarm/event displays and logs

Control System Configurations

The simplest machine control system consists of a single controller running in a "headless" configuration (see Figure 1.1). This configuration is used in applications that do not need an HMI except for maintenance or diagnostic purposes.

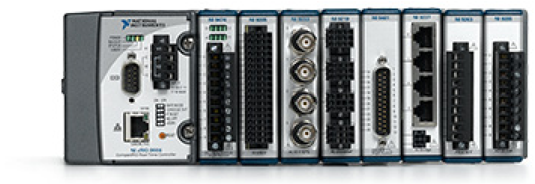


Figure 1.1. A Headless Controller

The next level of system capability and complexity adds an HMI and/or more controller nodes (see Figure 1.2). This configuration is typical for machines controlled by a local operator.

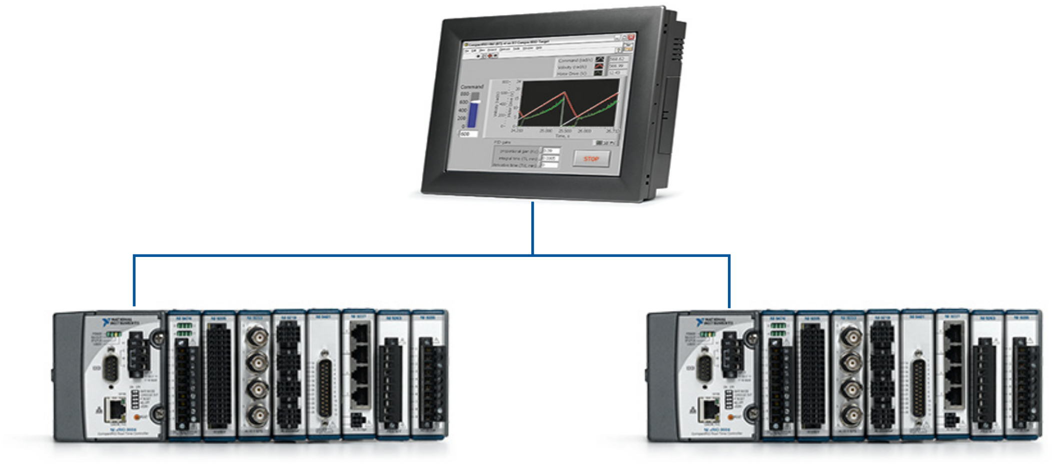


Figure 1.2. A Local Machine Control System

Complex machine control applications may involve many controllers and HMIs (Figure 1.3). They often involve a high-end server that acts as a data-logging and forwarding engine. This system configuration supports physically large or complex machines. With it, you can interact with the machine from various locations or distribute specific monitoring and control responsibilities among a group of operators.

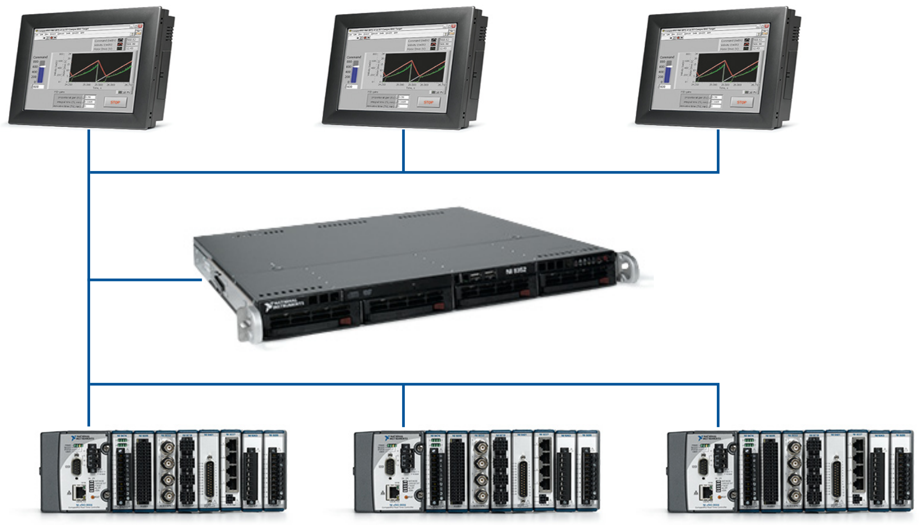


Figure 1.3. A Distributed Machine Control System

Control System Architectural Block Diagrams

A control system with a PAC and an HMI contains all the software components you need to build most machine control applications. Understanding how to build a basic PAC and HMI helps you scale to any machine control system.

1. A controller has components tCommunicate and interface to outside devices such as sensors and actuators, HMIs, and network devices
2. Store current data in a memory table (sometimes called a tag engine)
3. Run logic to control the machine or process
4. Perform housekeeping tasks such as start-up
5. Monitor and report system faults

An HMI has similar components except instead of performing control, it provides a user interface (UI). You can implement additional tasks such as alarm and event detection and logging on both the controller and the HMI.

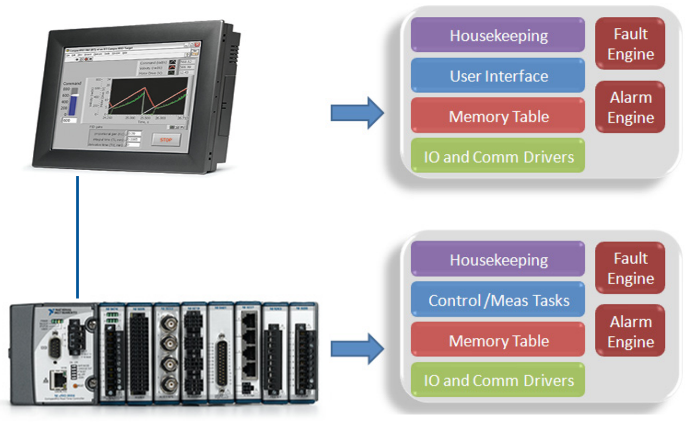


Figure 1.4. High-Level View of the Local Machine Control Architecture

By analyzing controller operations, you can break down the system into smaller components, each responsible for a specific task in the overall application. Figure 1.5 shows the controller architecture and individual components of the machine control application. Some of these components are ready-to-run as part of the machine control reference architecture, while others must be developed as part of the design and implementation of a specific machine control application.

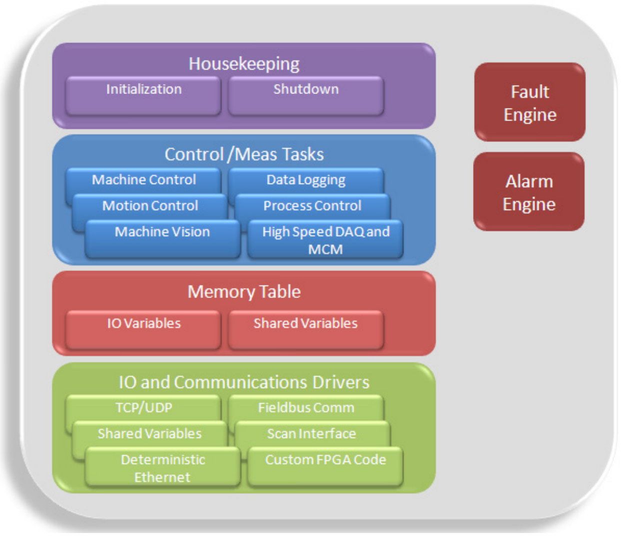


Figure 1.5. Controller Architecture and Components

This document walks through recommended implementations for various controller and HMI architecture components. It also offers example code and, in some cases, alternative implementations and the trade-offs between implementations.

Introduction to CompactRIO

CompactRIO is a rugged, reconfigurable embedded system containing three components – a real-time controller, a reconfigurable field-programmable gate array (FPGA), and industrial I/O modules.

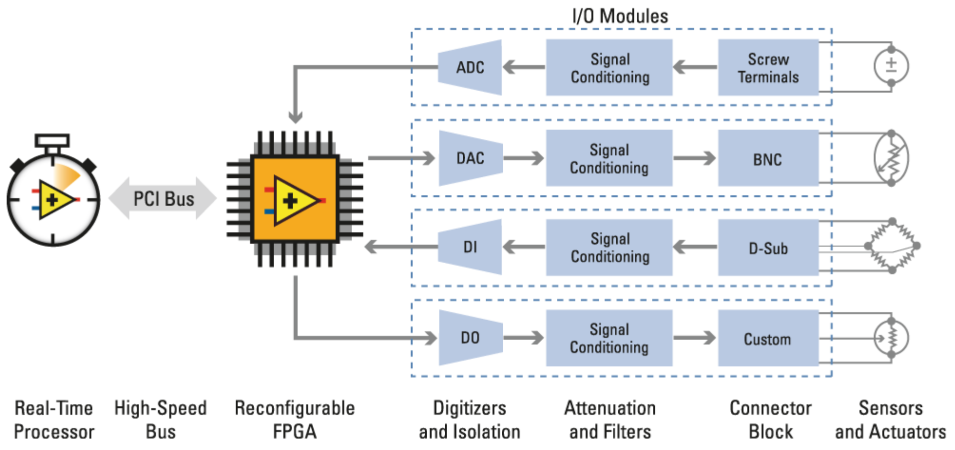


Figure 1.6. Reconfigurable Embedded System Architecture

Real-Time Controller

The real-time controller contains an industrial processor that reliably and deterministically executes LabVIEW Real-Time applications and offers multirate control, execution tracing, onboard data logging, and communication with peripherals. Additional options include redundant 9 to 30 VDC supply inputs, a real-time clock, hardware watchdog timers, dual Ethernet ports, up to 2 GB of data storage, and built-in USB and RS232.

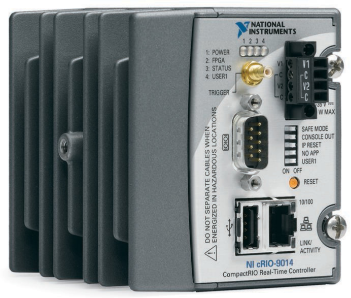


Figure 1.7. NI cRIO-9014 Real-Time Controller

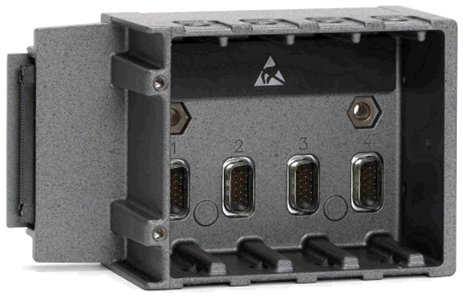


Figure 1.8. Reconfigurable FPGA Chassis

Reconfigurable FPGA Chassis

The reconfigurable FPGA chassis is the center of the embedded system architecture. The reconfigurable I/O (RIO) FPGA is directly connected to the I/O modules for high-performance access to the I/O circuitry of each module and unlimited timing, triggering, and synchronization flexibility. Because each module is connected directly to the FPGA rather than through a bus, you experience almost no control latency for system response compared to other industrial controllers. By default, this FPGA automatically communicates with I/O modules and provides deterministic I/O to the real-time processor. Out of the box, the FPGA enables programs on the real-time controller to access I/O with less than 500 ns of jitter between loops. You can also directly program this FPGA to run custom code. Because of the FPGA speed, this chassis is frequently used to create controller systems that incorporate high-speed buffered I/O, very fast control loops, or custom signal filtering. For instance, using the FPGA, a single chassis can execute more than 20 analog proportional integral derivative (PID) control loops simultaneously at a rate of 100 kHz. Additionally, because the FPGA runs all code in hardware, it provides the high reliability and determinism that is ideal for hardware-based interlocks, custom timing and triggering, or eliminating the custom circuitry normally required with custom sensors.

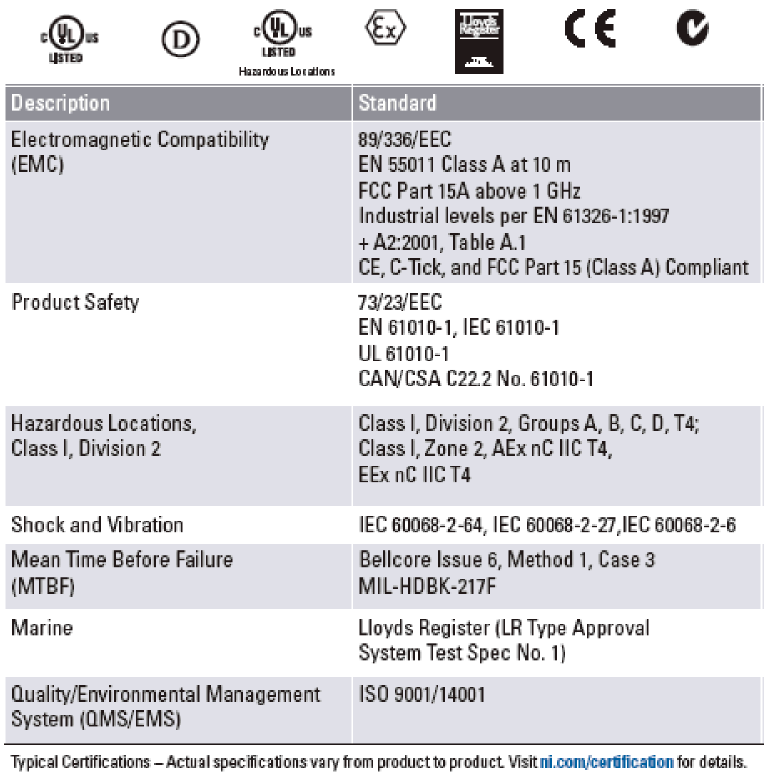
Industrial I/O Modules

I/O modules contain isolation, conversion circuitry, signal conditioning, and built-in connectivity for direct connection to industrial sensors/actuators. By offering a variety of wiring options and integrating the connector junction box into the modules, the CompactRIO system significantly reduces space requirements and field-wiring costs. You can choose from more than 50 NI C Series I/O modules for CompactRIO to connect to almost any sensor or actuator. Module types include thermocouple inputs; ±10 V simultaneous sampling, 24-bit analog I/O; 24 V industrial digital I/O with up to 1 A current drive; differential/TTL digital inputs; 24-bit IEPE accelerometer inputs; strain measurements; RTD measurements; analog outputs; power measurements; controller area network (CAN) connectivity; and secure digital (SD) cards for logging. Additionally, the platform is open, so you can build your own modules or purchase modules from other vendors. With the NI cRIO-9951 CompactRIO Module Development Kit, you can develop custom modules to meet application-specific needs. The kit provides access to the low-level electrical CompactRIO embedded system architecture for designing specialized I/O, communication, and control modules. It includes LabVIEW FPGA libraries to interface with your custom module circuitry.

Figure 1.9. You can choose from more than 50 I/O modules for CompactRIO to connect to almost any sensor or actuator.

CompactRIO Specifications

Many CompactRIO customers build systems that are sold and deployed around the world. To help simplify the process of designing systems for global deployment, CompactRIO has numerous certifications and has passed testing by third-party agencies.


LISTED
LISTED
Hazardous Locations

Description	Standard
Electromagnetic Compatibility (EMC)	89/336/EEC EN 55011 Class A at 10 m FCC Part 15A above 1 GHz Industrial levels per EN 61326-1:1997 + A2:2001, Table A.1 CE, C-Tick, and FCC Part 15 (Class A) Compliant
Product Safety	73/23/EEC EN 61010-1, IEC 61010-1 UL 61010-1 CAN/CSA C22.2 No. 61010-1
Hazardous Locations, Class I, Division 2	Class I, Division 2, Groups A, B, C, D, T4; Class I, Zone 2, AEx nC IIC T4, EEx nC IIC T4
Shock and Vibration	IEC 60068-2-64, IEC 60068-2-27,IEC 60068-2-6
Mean Time Before Failure (MTBF)	Bellcore Issue 6, Method 1, Case 3 MIL-HDBK-217F
Marine	Lloyds Register (LR Type Approval System Test Spec No. 1)
Quality/Environmental Management System (QMS/EMS)	ISO 9001/14001

Typical Certifications – Actual specifications vary from product to product. Visit ni.com/certification for details.

Figure 1.10. CompactRIO Specifications

CHAPTER 2
Basic Architecture for Control

Basic Controller Architecture Background

Building complex systems requires an architecture that allows code reuse, scalability, and execution management. The next two sections describe how to build a basic architecture for control applications and how to perform a simple PID loop using this architecture.

A basic controller architecture has three main states:

1. Initialization (Housekeeping)
2. Control (IO and Comm Drivers, Memory Table, Control and Meas Tasks)
3. Shutdown (Housekeeping)

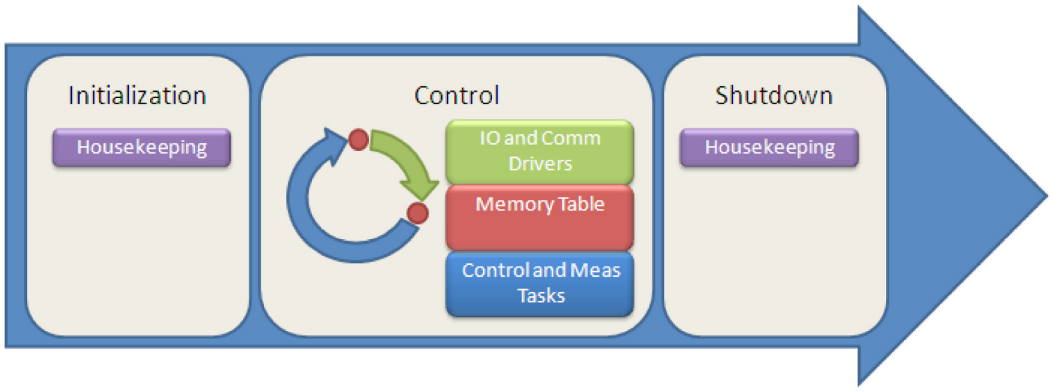


Figure 2.1. The Three Main States of a Basic Controller Architecture

Initialization Routine

Housekeeping

Before executing the main control loop, the program needs to perform an initialization routine. The initialization routine prepares the controller for execution and is not the place for logic related to the machine such as logic for machine startup or initialization. That logic should go in the main control loop. This initialization routine

1. Sets all internal variables to default states.
2. Creates any programming structures necessary for operation. This may include queues, real-time first-in-first-out memory buffers (FIFOs), VI refnums, and FPGA bit file downloading.
3. Performs any additional user-defined logic to prepare the controller for operation such as preparing log files.

Control Routine

IO and Comm Drivers
Memory Table
Control and Meas Tasks

I/O, Communications, and the Memory Table

Many programmers are familiar with direct I/O access, during which subroutines directly send and receive inputs and outputs from the hardware. This method is ideal for waveform acquisition and signal processing and for smaller single-point applications. However, control applications normally use single-point reads and writes and can become very large with multiple states – all of which need access to the I/O. Accessing I/O introduces overhead in the system and can slow it down. Additionally, managing multiple I/O accesses throughout all levels of a program makes it very difficult to change I/O and implement features such as simulation or forcing. To avoid these problems, the control routine uses a scanning I/O architecture. In this type of architecture, you access the physical hardware only once per loop iteration using I/O and communication drivers (labeled as IO and Comm Drivers in Figure 2.1). Input and output values are stored in a memory table, and control and measurement tasks access the memory space instead of directly accessing the hardware. This architecture provides numerous benefits:

- I/O abstraction so you can reuse subVIs and functions (no hard coding of I/O)
- Low overhead
- Deterministic operation
- Support for simulation
- Support for "forcing"
- Elimination of the risk of I/O changes during logic execution

Control and Measurement Tasks

Control and measurement tasks are the machine-specific logic that defines the control application. This may be process control or more sophisticated machine control. In many cases, it is based on a state machine to handle complex logic with multiple states. A later section explores how to use state machines to design the logic. To execute in the control architecture, the main control task must

- Execute in less time than the I/O scan rate
- Access I/O through the I/O memory table instead of through direct I/O reads and writes
- Not use "while loops" except to retain state information in shift registers
- Not use "for loops" except in algorithms
- Not use "waits" and instead use timer functions or "Tick Count" for timing logic
- Not perform waveform, logging, or nondeterministic operations (use parallel, lower-priority loops for these operations)

The user logic can

- Include single-point operations such as PID or point-by-point analysis
- Use a state machine to structure the code

You can diagram the control routine as one loop where I/O is read and written and a control task runs with communication via a memory table; but, in reality, it is multiple synchronized loops, and there may be more than one control or measurement task.

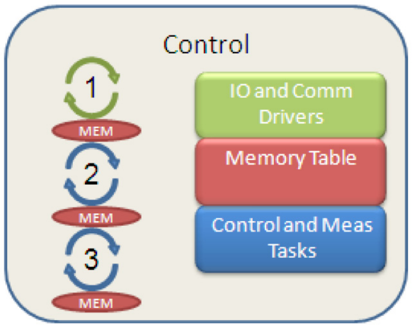


Figure 2.2. The Three Main States of a Basic Controller Architecture

Shutdown Routine Housekeeping

When the controller needs to stop running because of a command or a fault condition, it stops running the main control loop and runs a shutdown routine. The shutdown routine shuts down the controller and puts it in a safe state. Use it only for controller shutdown – it is not the place for machine shutdown routines, which should go in the main control loop. The shutdown routine

1. Sets all outputs to safe states
2. Stops any parallel loops that are running
3. Performs any additional logic such as notifying the operator of any controller fault or logging state information

Basic Controller Architecture Example in LabVIEW

Example

LabVIEW Example Code
is provided for this section

To demonstrate this control architecture, build a basic PID control application. This simple application controls a temperature chamber to maintain 350 °F. Featuring one analog input from a thermocouple and one pulse-width modulation (PWM) digital output that is connected to a heater, the application uses a PID algorithm for control. This overly simplistic application is used here to explain the architecture components without adding the complexity of an intricate control example. More detailed control examples using this architecture are explored later in this document.

To build this application in LabVIEW, use five of the controller architecture components:

1. Initialization routine
2. Shutdown routine
3. A simple process control task
4. I/O variables in the memory table
5. The RIO Scan Interface to access I/O

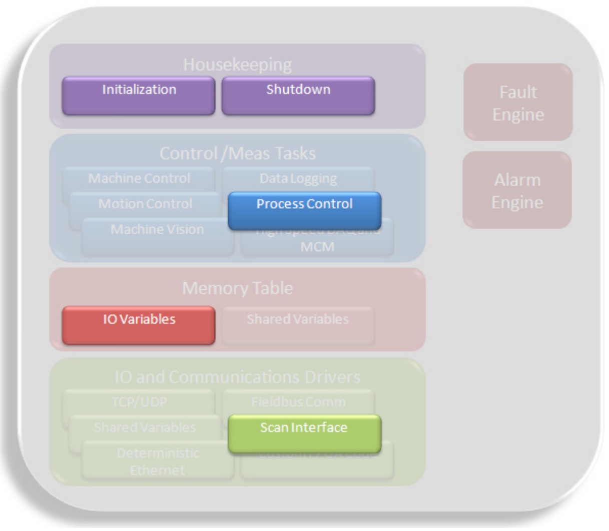


Figure 2.3. Example PID Controller Architecture

Initialization and Shutdown Routines

1. First add the initialization routine and a shutdown routine. The initialization routine needs to configure the controller so it is ready to run any logic, and the shutdown routine needs to perform any actions based on a shutdown.

2. To manage this controller sequence, create a sequence structure with three frames: one for initialization routines, one for the control and measurement tasks, and one for the shutdown routine.

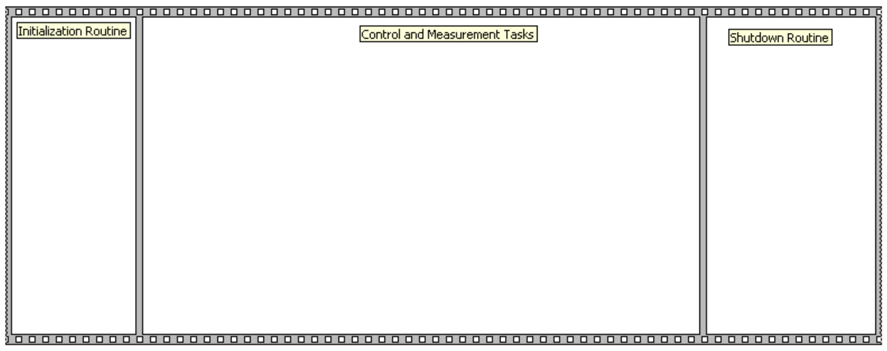


Figure 2.4. Manage this controller sequence with three frames: initialization routines, control and measurement tasks, and shutdown routine.

3. Add any initialization or shutdown logic. In this application, no initialization is required for the controller. By default, the controller leaves output values at the last state. In this application, at shutdown, you need to set outputs to an off state. You also can add other logic at shutdown such as error logging into the structure.

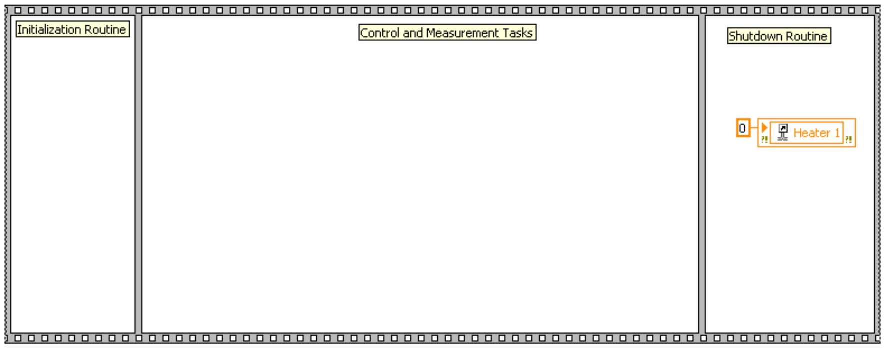
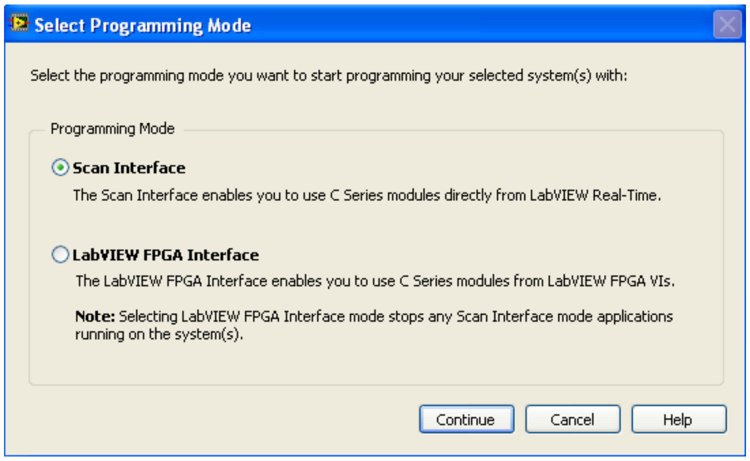


Figure 2.5. Add any initialization or shutdown logic.

You now have a complete initialization and shutdown routine. Now you need to add your control and measurement tasks.

I/O Scan and Memory Table Scan Interface IO Variables

Starting with LabVIEW 8.6, you have a programming option for CompactRIO called the RIO Scan Interface. When you discover your CompactRIO controller from the LabVIEW Project, you have the option to program the controller using the Scan Interface or a LabVIEW FPGA interface (if you do not have LabVIEW FPGA installed, LabVIEW defaults to the scan interface).


Select Programming Mode

Select the programming mode you want to start programming your selected system(s) with:

Programming Mode

[✓ **Scan Interface**]
The Scan Interface enables you to use C Series modules directly from LabVIEW Real-Time.

LabVIEW FPGA Interface
The LabVIEW FPGA Interface enables you to use C Series modules from LabVIEW FPGA VIs.

Note: Selecting LabVIEW FPGA Interface mode stops any Scan Interface mode applications running on the system(s).

Figure 2.6. Starting in LabVIEW 8.6, you can program CompactRIO controllers using the Scan Interface.

When the controller is accessing I/O via the scan interface, module I/O is automatically read from the modules and placed in a memory table on the CompactRIO controller. The default rate for the I/O scan is 10 ms and can be configured under the controller properties. You can access the I/O using I/O variables aliases.

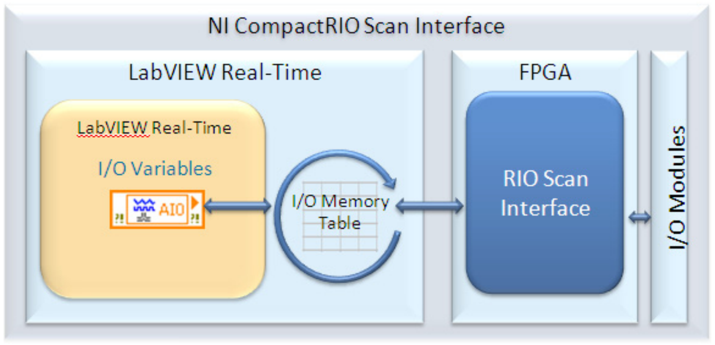
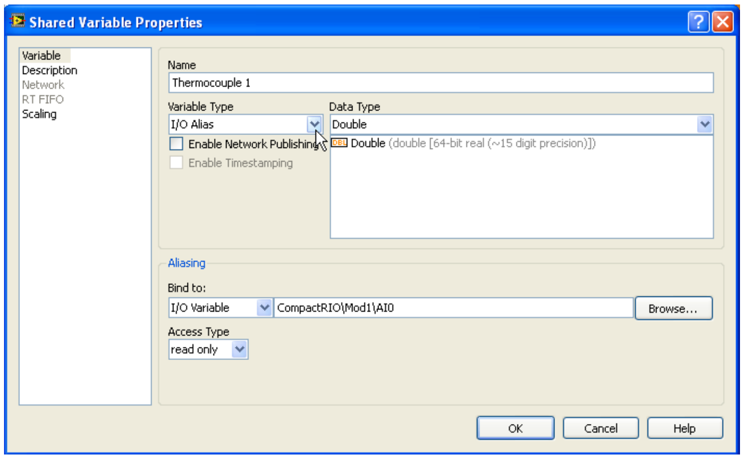


Figure 2.7. Block Diagram Description of the CompactRIO Scan Interface Software Components

In your system, you have a thermocouple input module and a PWM output module. You can both configure and access these via the scan interface. To read and write these values from LabVIEW, create I/O aliases to the items. An I/O alias refers to physical I/O that you can use to obtain additional scaling and maintain code portability.


Shared Variable Properties

Variable
Description
Network
RT FIFO
Scaling

Name: Thermocouple 1
Variable Type: I/O Alias
Data Type: Double
☐ Enable Network Publishing
☐ Enable Timestamping
Double (double [64-bit real (~15 digit precision)])

Aliasing
Bind to: I/O Variable — CompactRIO\Mod1\AI0 — Browse...
Access Type: read only

OK Cancel Help

Figure 2.8. Creating an I/O Alias

1. To create an I/O alias, right-click on the controller and select a new variable. Select the variable type as I/O Alias and bind it to the physical I/O.

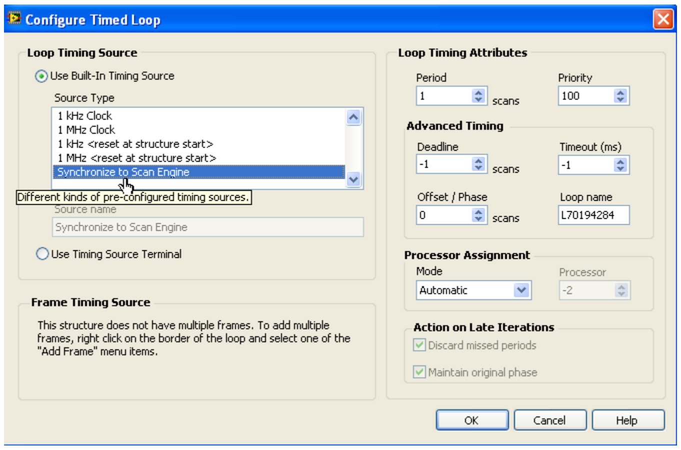
2. For this example, create two I/O aliases for Thermocouple 1 (bound to a TC module) and Heater 1 (bound to a digital output module configured for PWM output), and put both into a library called IO Library.

Control and Measurement Tasks

Process Control

Schedule each control and measurement task using a timed loop. You should synchronize the timed loop to the I/O scan (NI Scan Engine) to provide proper synchronization between the control task and the I/O.

1. Create a timed loop and configure it to synchronize to the scan engine. Leave the period at 1 so the loop runs every time the I/O scan runs.


Configure Timed Loop

Loop Timing Source
◉ Use Built-In Timing Source
Source Type
1 kHz Clock
1 MHz Clock
1 kHz <reset at structure start>
1 MHz <reset at structure start>
Synchronize to Scan Engine
Different kinds of pre-configured timing sources.
Source name
Synchronize to Scan Engine
○ Use Timing Source Terminal

Frame Timing Source
This structure does not have multiple frames. To add multiple frames, right click on the border of the loop and select one of the "Add Frame" menu items.

Loop Timing Attributes
Period: 1 scans
Priority: 100

Advanced Timing
Deadline: -1 scans
Timeout (ms): -1
Offset / Phase: 0 scans
Loop name: L70194284

Processor Assignment
Mode: Automatic
Processor: -2

Action on Late Iterations
✓ Discard missed periods
✓ Maintain original phase

OK Cancel Help

Figure 2.9. Synchronizing the NI Scan Engine

2. Write the control logic to read the inputs from the I/O aliases, run the logic, and write to the I/O aliases. Normally, you create subVIs for encapsulation to enable code reuse; however, because this example is intended to show the overall architecture, the code is trivial. Further encapsulation of the code is redundant; in later examples, learn about appropriate code encapsulation. For this simple example, drop a PID VI on the block diagram and wire constants so the output range is [100, 0], the PID gains are [10, 0.1, 0], and the setpoint is 350. Instead of constants, these can also be variables that you can reconfigure while the program is executing. Wire the "Temperature 1" I/O alias to the "Process Variable" terminal and wire the "Heater 1" I/O alias to the "Output" terminal. Add the appropriate error-handling components.

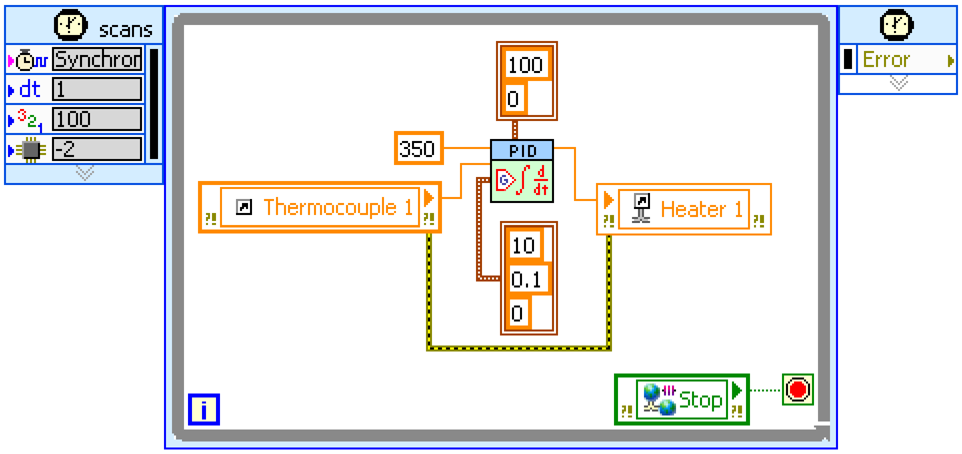


Figure 2.10. In this example, use a network-published shared variable to stop the loop.

Now you can run the temperature control program. It has start-up and shutdown procedures, a scanning architecture, reusable subVIs that are not hard-coded to I/O, and error handling. This is the fundamental architecture for machine control used throughout this document.

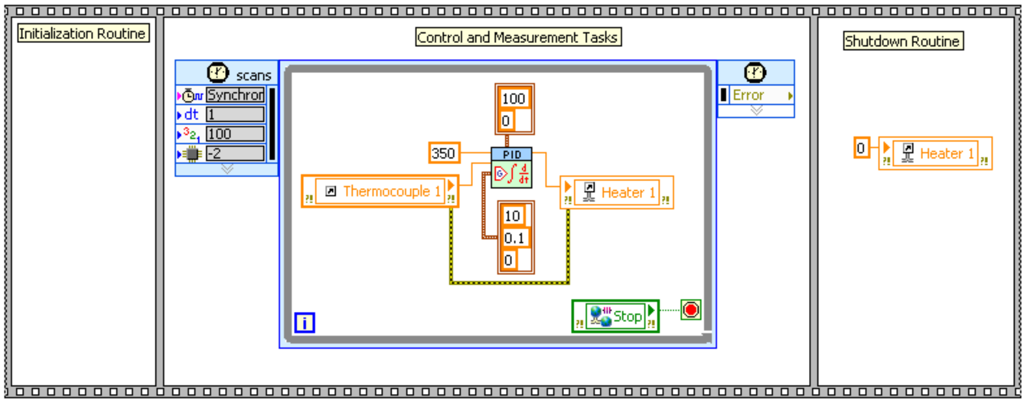


Figure 2.11. Fundamental Architecture for Machine Control

State-Based Designs

With this fundamental architecture, you can build sophisticated machine control applications. However, as the logic gets more complex, it is important to use a proper architecture to organize your design. By establishing a software architecture, you can create extensible and easily maintainable applications. Architecting systems to be represented by a series of states is a common method for designing extensible and manageable code.

State Machine Overview

A state machine is a common and useful software architecture. You can use the state machine design pattern to implement any algorithm that can be explicitly described by a state diagram or flow chart. A state machine usually illustrates a moderately complex decision-making algorithm, such as a diagnostic routine or a process monitor.

More precisely defined as a finite state machine, a state machine consists of a set of states and a transition function that maps to the next state. Each state machine should be designed to execute actions upon entry while it exists in the state or on exit. Because state machines are used as part of a larger machine control architecture, they cannot use wait statements and loops except to retain states or perform algorithms such as a for loop used for array manipulation.

Use state machines in applications where distinguishable states exist. If you can deconstruct an application into several regions of operation, a state machine is a good architectural choice. Each state can lead to one or multiple states or end the process flow. A state machine relies on user input or in-state calculation to determine which state to go to next. Many applications require an initialization state, followed by a default state, where you can perform a variety of actions. These actions depend on previous and current inputs as well as states. A shutdown state is commonly used to perform cleanup actions.

Example Scenario of Developing with State Machines

To learn how an application benefits from the state machine architecture, design a control system for a chemical reacting vessel. In this application, the controller needs to:

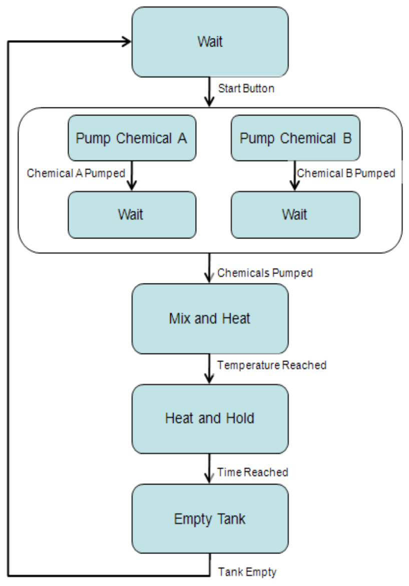
1. Wait for an operator start command from a push button.
2. Meter two chemical flows into a tank based on output from a flow totalizer (two parallel processes – one for each chemical flow).
3. After filling the tank, turn on a stirrer and raise the temperature in the tank. Once the temperature has reached 200 °F, turn off the stirrers and hold the temperature constant for 10 seconds.
4. Pump the contents to a holding tank.
5. Go back to the wait state.

Note that for simplicity in this application, the chemical flow rates have been hard-coded to 850, the temperature to 200 °F, and the time to 10 seconds. This was done to further simplify the application. In a real application, you can load these values from a recipe or an operator can enter them.

State Machine Example in LabVIEW

LabVIEW Example Code
is provided for this section

The first step in building this application is to map out the logic and I/O points. Because this application involves a sequence of steps, a flowchart is a good tool for planning the application. Below is a flowchart for this application and a list of I/O signals.



I/O Signals	I/O Name
Operator push button	Input_Operator_PB
Pump A	Output_PumpA
Pump B	Output_PumpB
Chemical A Flow	Input_ChemA_Flow
Chemical B Flow	Input_ChemB_Flow
Stirrer	Output_Stirrer
Heater	Output_Heater
Thermocouple	Input_TC
Drain Pump	Ouput_PumpDrain
Tank Empty Level Sensor	Input_TankEmpty_LS

Figure 2.12. Application Flowchart and I/O Signal List

Each state in a state machine performs a unique action and calls other states. State transitions depend on whether some condition or sequence occurs. Translating the state transition diagram into a LabVIEW block diagram requires the following infrastructure components:

- Case structure – contains a case for each state, and the code to execute for each state
- Shift register – contains state transition information
- State functionality code – implements the function of the state
- Transition code – determines the next state in the sequence

Once you have defined the states within a system, use a case structure in LabVIEW to represent and contain the logic for each state.

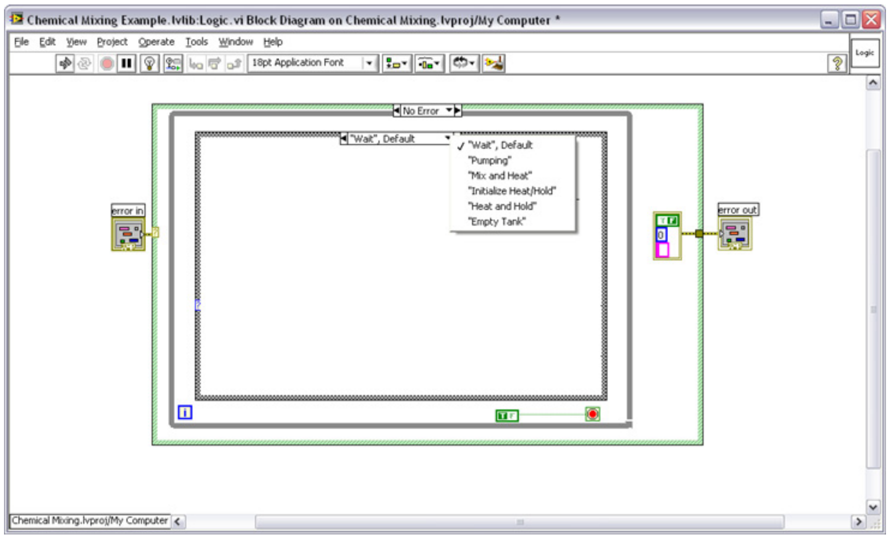

Chemical Mixing Example.lvlib:Logic.vi Block Diagram on Chemical Mixing.lvproj/My Computer *
File Edit View Project Operate Tools Window Help
18pt Application Font
No Error
"Wait", Default
✓ "Wait", Default
"Pumping"
"Mix and Heat"
"Initialize Heat/Hold"
"Heat and Hold"
"Empty Tank"
error in
error out
Chemical Mixing.lvproj/My Computer

Figure 2.13. Create a case for each state of the machine.

By adding a shift register to the application, you can keep track of and pass current state information each time the state machine executes. The case structure input terminal connects to the shift register.

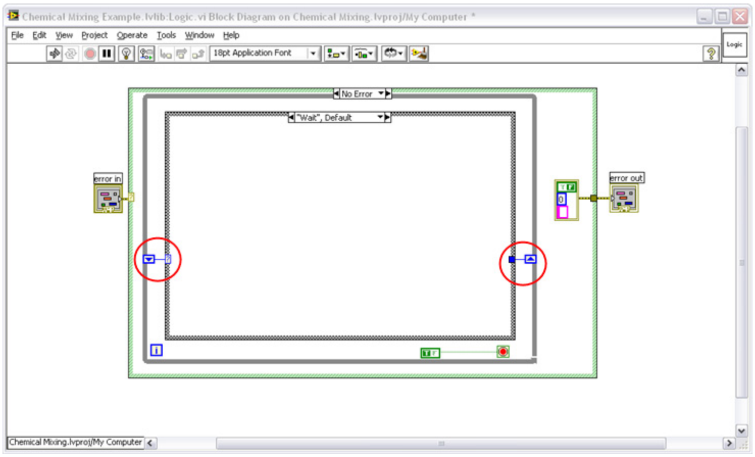

Chemical Mixing Example.lvlib:Logic.vi Block Diagram on Chemical Mixing.lvproj/My Computer *
File Edit View Project Operate Tools Window Help
18pt Application Font
No Error
"Wait", Default
error in
error out
Chemical Mixing.lvproj/My Computer

Figure 2.14. Add shift registers to pass state data.

Each state within the case structure now can contain some LabVIEW code that executes each time the state becomes active. The code in Figure 2.15 executes a PID function for mixing and heating your chemical reacting vessel.

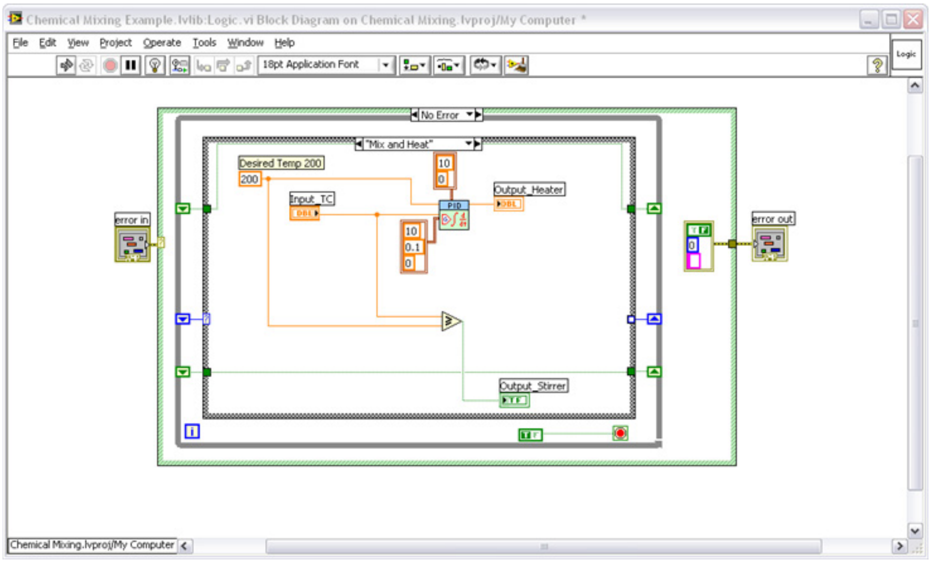

Chemical Mixing Example.lvlib:Logic.vi Block Diagram on Chemical Mixing.lvproj/My Computer *
File Edit View Project Operate Tools Window Help
18pt Application Font
No Error
"Mix and Heat"
Desired Temp 200
200
Input_TC
Output_Heater
Output_Stirrer
error in
error out
Chemical Mixing.lvproj/My Computer

Figure 2.15. Insert your logic into each state.

Each state must then determine which state to transition to next depending on the conditions within the system. Add transition logic to determine which state should be executed next.

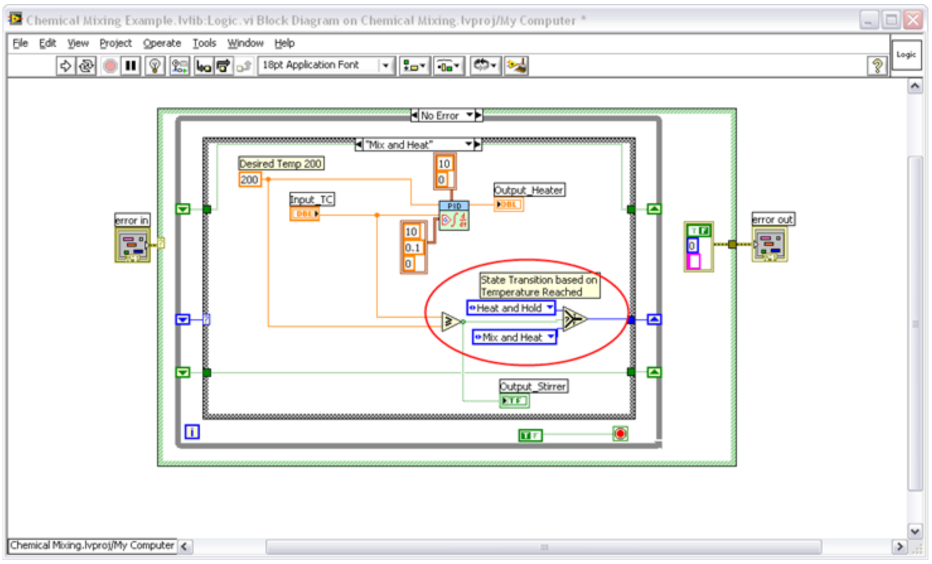

Chemical Mixing Example.lvlib:Logic.vi Block Diagram on Chemical Mixing.lvproj/My Computer *
File Edit View Project Operate Tools Window Help
18pt Application Font
No Error
"Mix and Heat"
Desired Temp 200
200
Input_TC
Output_Heater
State Transition based on Temperature Reached
Heat and Hold
Mix and Heat
Output_Stirrer
error in
error out
Chemical Mixing.lvproj/My Computer

Figure 2.16. Use a selector to determine the next logic case.

For this example, use simulated I/O instead of physical I/O so you can test your logic. To do this, use a global variable instead of hardware read/write VIs. This is a convenient way to test your logic with interactive controls and indicators before deploying to actual hardware. The ability to easily simulate I/O is one of the benefits of this architecture.

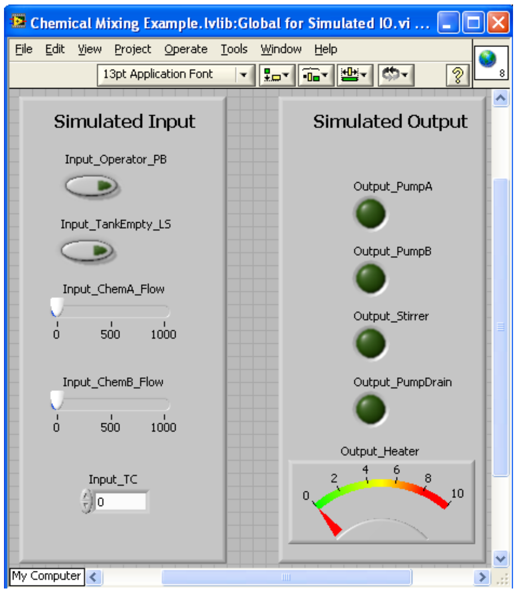


Figure 2.17. Global variables are a convenient method to test logic without hardware.

Because one state in your machine has parallel processes, you need to build a second state machine to represent the parallel logic and call parallel processes in that state. Exit the state only when both parallel processes have completed.

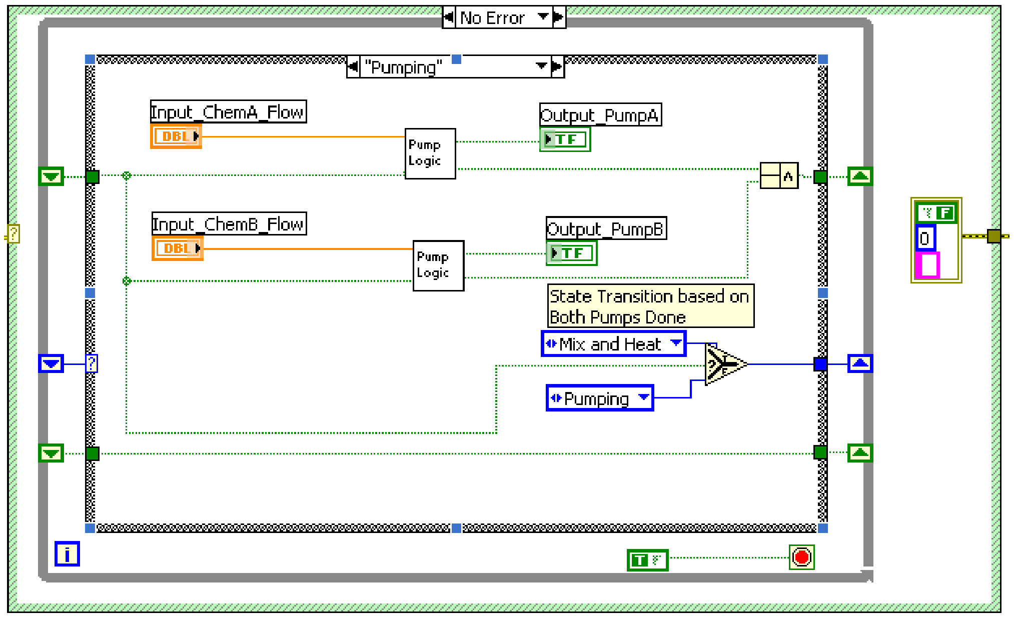


Figure 2.18. Parallel logic can occur in one state.

Introduction to Statecharts

State machines are just one method for representing state-based diagrams in software. As systems become more complex, you need to move to higher levels of abstraction to ensure a maintainable software design. Statecharts offer a superset of state machine functionality and features that improve application scalability.

Statecharts offer a higher level graphical programming tool; providing a system-level view that describes the complete function of a system or application. The use of statecharts helps organize software applications in a manner that reduces unexpected behavior by ensuring all possible states are accounted for. The statechart programming model is especially useful for reactive systems, which are characterized by how they respond to inputs. Statecharts are similar to graphical dataflow programs in that they are self-documenting and promote the easy transfer of knowledge between developers. A new member of a design team can look at a statechart diagram and quickly grasp the elements of a system.

To begin understanding statecharts, it is best to start with the classic state diagram. The classic state diagram consists of two main constructs: states and transitions. In Figure 2.19, the state diagram describes a simple soda vending machine with five states and seven transitions to illustrate how the machine operates. The machine starts in the "idle" state and transitions to the "count coins" state when coins are inserted. The state diagram shows additional states and transitions when the machine waits for a selection, then dispenses a soda, and finally gives change.

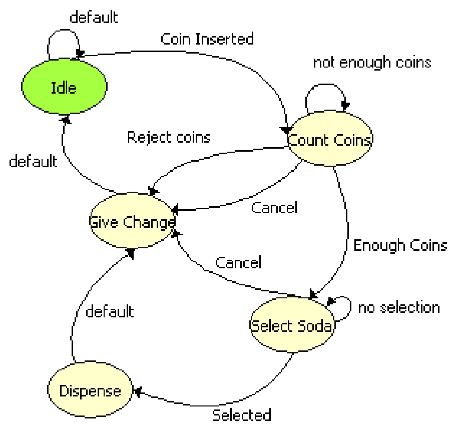


Figure 2.19. State Diagram of a Simple Soda Vending Machine

The figure shows a statechart that describes the behavior of the same machine. In the statechart, you can nest the "count coins" and "dispense" states within a superstate by using a statechart feature called hierarchy, allowing states to be embedded within another. With hierarchy, you can simplify your designs. Now you have to define only one transition (T3) from either of these two states to the "give change" state. You can configure the T3 transition to respond to three events: soda dispensed, change requested, or coins rejected. Notice how the "select soda" state in the classic state diagram has been removed. This is accomplished using a "guard" condition to transition T2. With guard conditions, you can embed logic within a transition. Guard conditions must evaluate to "true" for the transition to occur. If the result of the guard condition is "false," the event is ignored and the transition does not take place.

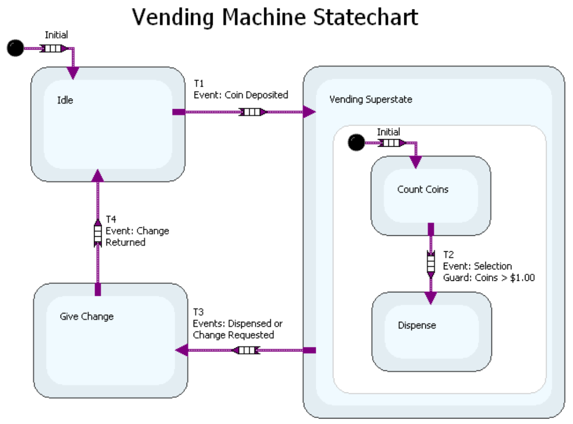


Figure 2.20. Statechart of a Simple Soda Vending Machine

The previous example demonstrates a very basic vending machine. You can add complexity to your system by adding a requirement for monitoring the temperature of your vending machine while concurrently counting coins and dispensing drinks. With statecharts, you can easily expand the functionality of a software design. They have a notion of concurrency, which allows a statechart to reside in multiple states at the same time. These states are said to be orthogonal or and-states. By applying concurrency to your statechart, you can encapsulate the dispensing logic and the temperature control into an and-state. And-states describe a system that is simultaneously in two states that are independent of each other. The T7 transition shows how statecharts can define an exit that applies to both substatecharts.

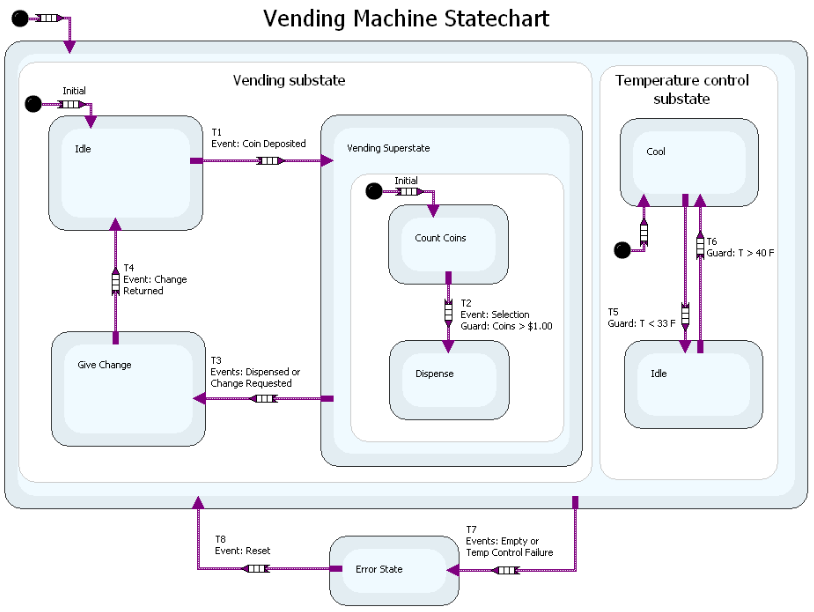


Figure 2.21. The T7 transition shows how statecharts can define an exit that applies to both substatecharts.

In addition to hierarchy and concurrency, statecharts have features that make them valuable for complex systems. They have a concept of history, allowing a superstate to "remember" which substate within it was previously active. For example, consider a superstate that describes a machine that pours a substance and then heats it. A halt event may pause the execution of the machine while it is pouring. When a resume event occurs, the machine remembers to resume pouring.

LabVIEW Statechart Module Tutorial

The LabVIEW Statechart Module is an editor for LabVIEW that you can use to quickly build full state-based machine logic. It is hierarchical, so you can use multiple statecharts and LabVIEW VIs together in a complex application. LabVIEW statecharts run on Windows, real-time targets, and FPGA targets. They comprise **regions, states, pseudostates, transitions,** and **connectors**.

Statechart Regions

A **region** is an area that contains **states**. The top-level statechart diagram is a region within which states are placed. Additionally, you can create regions within states to take advantage of hierarchical designs by creating states within another state. This ability is illustrated in Figure 2.22, where a substate has been created within a state using a region. Each region must contain an **initial** pseudostate.

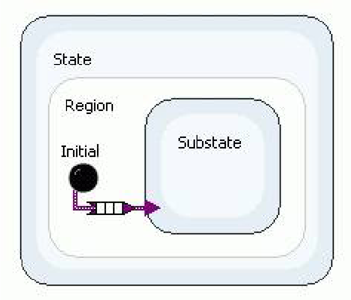


Figure 2.22. Create a substate within a state using a region.

Statechart States
A state is a condition of a statechart. You must place states within regions and have at least one incoming transition.

Figure 2.23. A state is a condition of a statechart.

Each state has an associated **entry** and **exit action**. An **entry action** is LabVIEW code that executes when entering a state. An **exit action** is code that executes when you leave a state (just before transitioning to the next state). Each state has only one entry and exit action. Both are optional. The **entry** and/or **exit** executes the action every time the state is entered or exited when present.

You can access this code through the **Configure State** dialog box.

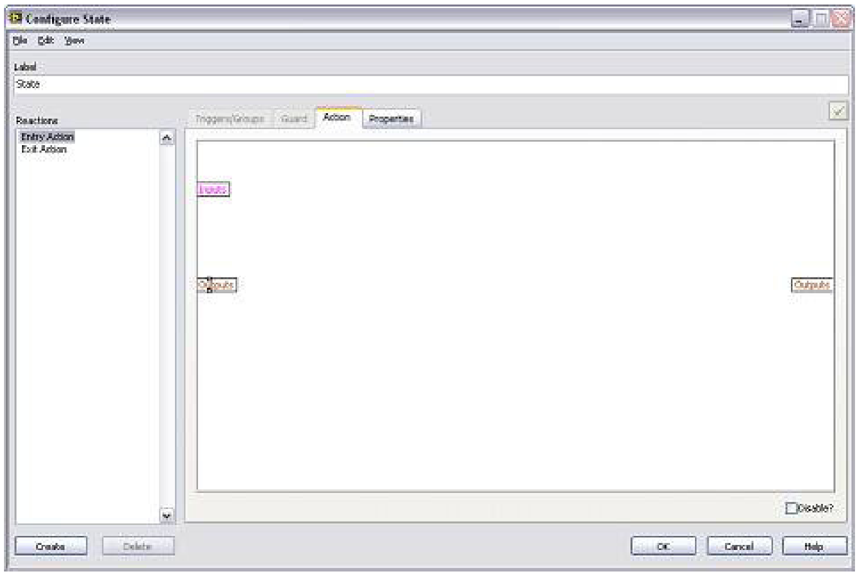


Figure 2.24. You can access entry and exit code through the Configure State dialog box.

You can further configure states to have **static reactions**, which are the actions a state performs when it is not taking any incoming or outgoing transitions. An individual state can have multiple static reactions that can execute at each iteration of the statechart.

Each static reaction comprises three components – trigger, guard, and action.

A **trigger** is an event or signal that causes a statechart to react. In **synchronous statecharts**, triggers are automatically passed to the statechart at periodic intervals. By default, the trigger value is set to **NULL.**

A **guard** is a piece of code that is evaluated before performing the action of the state. If the guard evaluates to true, the action code executes. If it evaluates to false, the action is not executed.

If the statechart receives a trigger that is to be handled by a particular static reaction, and the guard code evaluates to true, the reaction performs the **action** code. The action is LabVIEW code that performs the desired logic of the state. This can be reading inputs or internal state information and modifying outputs accordingly.

You can create static reactions through the **Configure State** dialog by creating a new reaction. Once you create a new reaction, you can associate it with a trigger and implement guard and action code. You can configure only static reactions to have a trigger and guard.

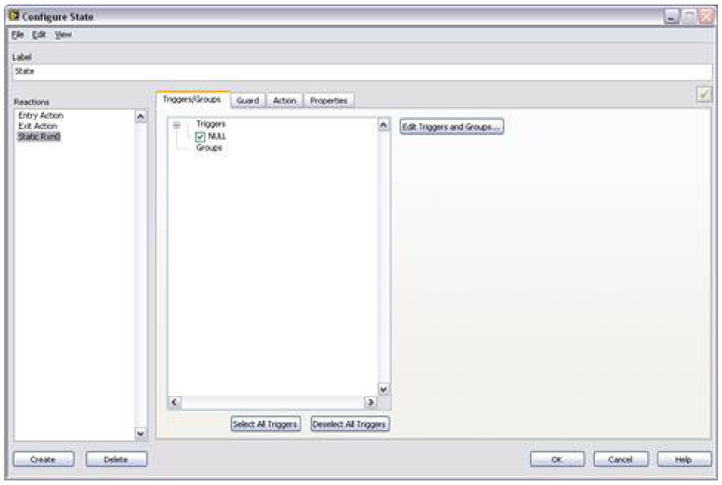


Figure 2.25. You can create static reactions through the Configure State dialog by creating a new reaction.

Orthogonal Regions and Concurrency

When a state contains two or more regions, the regions are said to be orthogonal. Regions 1 and 2 in Figure 2.26 are orthogonal.

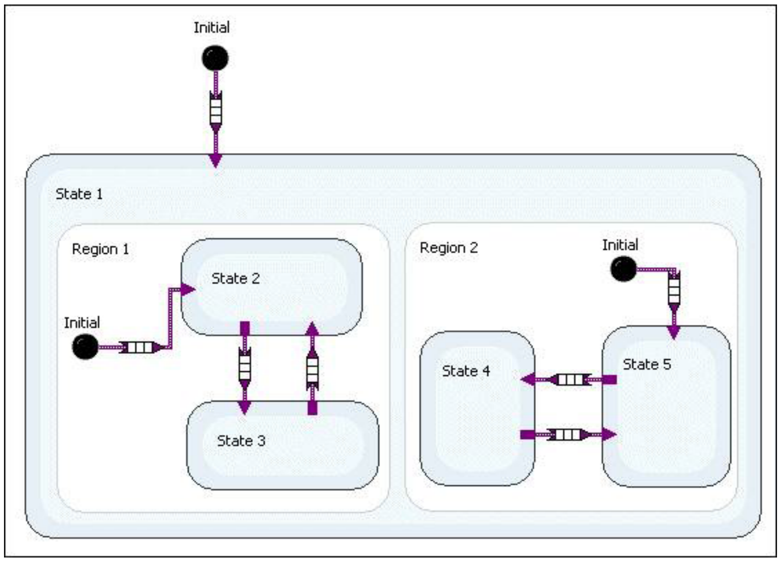


Figure 2.26. Regions 1 and 2 in are orthogonal.

Substates in orthogonal regions are concurrent, which means that while the superstate is active, the statechart can be in only one substate from each orthogonal region during each statechart iteration.

Transitions

Transitions define the conditions that statecharts move between states.

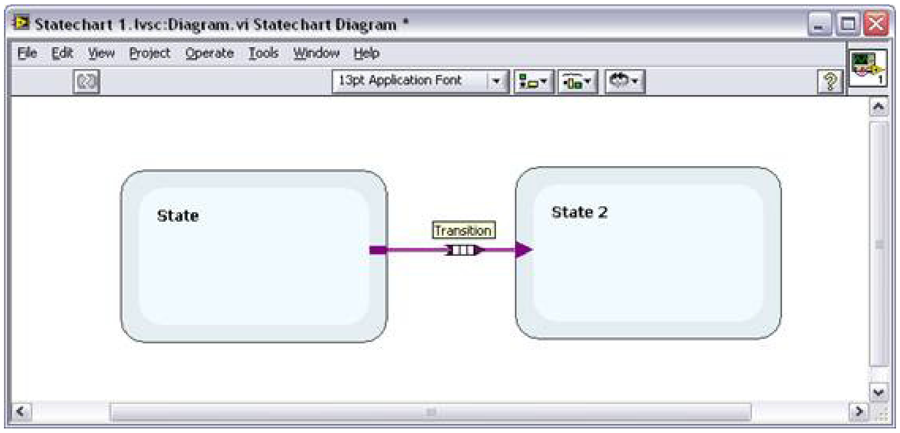

Statechart 1.lvsc:Diagram.vi Statechart Diagram *
File Edit View Project Operate Tools Window Help
13pt Application Font
State
Transition
State 2

Figure 2.27. Transitions define the conditions that statecharts move between states.

Transitions consist of ports and transition nodes. Ports are the connections between states, and transition nodes define the behavior of the transition using triggers, guards, and actions. You configure transition nodes through the Configure Transition dialog.

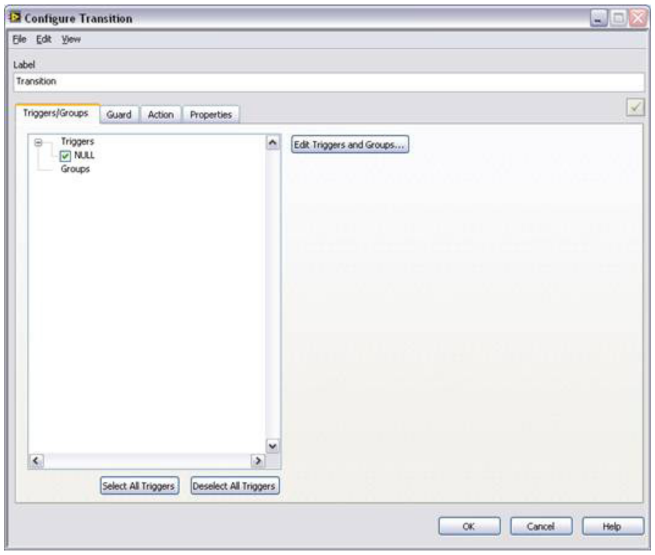


Figure 2.28. Configure transition nodes through the Configure Transition dialog.

Triggers, guards, and actions behave the same way in transitions that they do in states. A transition responds to a trigger, and if the guard code evaluates to true, the action is executed and the statechart moves to the next state. If the guard code does not evaluate to true, the action code is not executed and the statechart does not move to the state indicated by that transition.

Pseudostates

A pseudostate is a statechart object that represents a state. The LabVIEW Statechart Module includes the following pseudostates:

- Initial state – Represents the first state that occurs when entering a region. An initial state must be present in each region.
- Terminal state – Represents the final state of a region and ends the execution of all states within that region.

- **Shallow history** – Specifies that when the statechart leaves and returns to a region, the statechart enters the highest-level substates that were active when the statechart left the region.
- **Deep history** – Specifies that when the statechart leaves and returns to a region, the statechart enters the lowest-level substates that were active when the statechart left the region.

Connectors
A connector is a statechart object that connects multiple transition segments. The LabVIEW Statechart Module includes the following connectors:

- **Fork** – Splits one transition segment into multiple segments.
- **Join** – Merges multiple transition segments into one segment.
- **Junction** – Connects multiple transition segments.

Statechart Example in LabVIEW

Example LabVIEW Example Code
is provided for this section

To demonstrate the benefits of the LabVIEW Statechart Module, use the previous example built with state machines:

1. Wait for an operator start command from a push button.
2. Meter two chemical flows into a tank based on output from a flow totalizer (two parallel processes – one for each chemical flow).
3. After filling the tank, turn on a stirrer and raise the temperature in the tank. Once the temperature has reached 200 °F, the system turns off the stirrers and holds the temperature constant for 10 seconds.
4. Pump the contents to a holding tank.
5. Go back to wait state.

Note that for simplicity in this application, the chemical flow rates have been hard-coded to 850, the temperature to 200 °F, and the time to 10 seconds. In a real application, you can load these values from a recipe or an operator can enter them.

To build this application, first create a library with I/O aliases to each of the I/O signals.

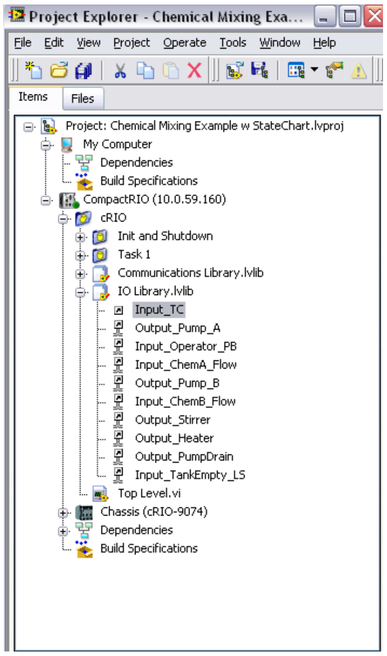


Figure 2.29. Create a library with I/O aliases to each of the I/O signals.

Next create the shutdown task to set the default output states for the output I/O aliases.

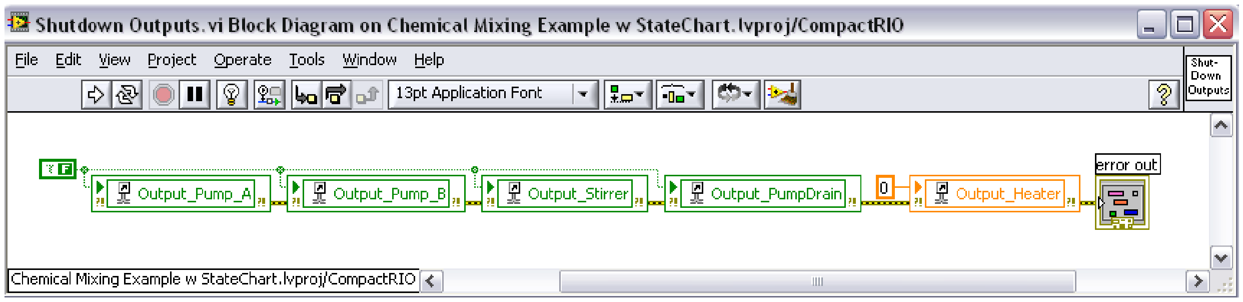

Shutdown Outputs.vi Block Diagram on Chemical Mixing Example w StateChart.lvproj/CompactRIO
File Edit View Project Operate Tools Window Help
13pt Application Font
Shut-Down Outputs
F
Output_Pump_A
Output_Pump_B
Output_Stirrer
Output_PumpDrain
0
Output_Heater
error out
Chemical Mixing Example w StateChart.lvproj/CompactRIO

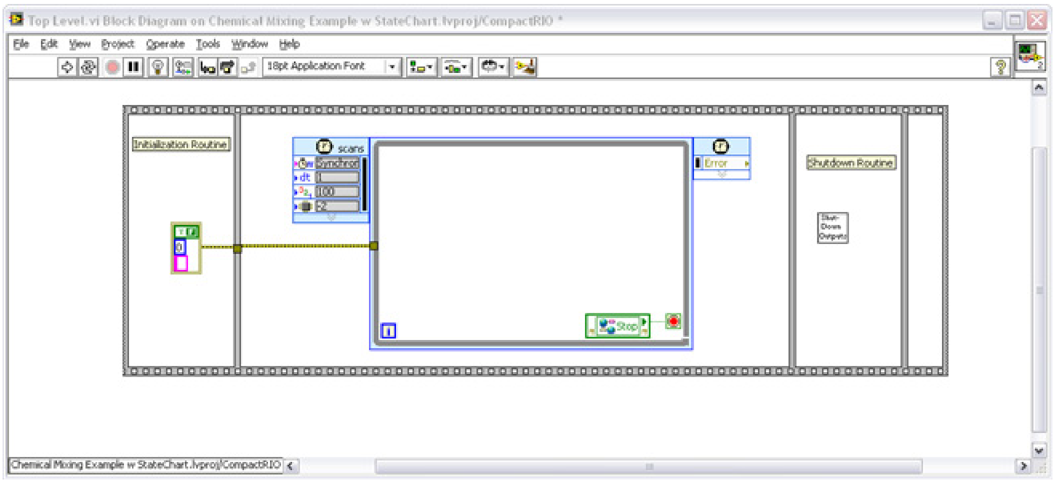
Figure 2.30. Create the shutdown task to set the default output states for the output I/O aliases.

The process for developing a statechart-based application involves the following steps:

1. Design the Caller VI
2. Define the inputs, outputs, triggers
3. Develop the statechart diagram
4. Place the statechart in the Caller VI

Step 1. Design the Caller VI

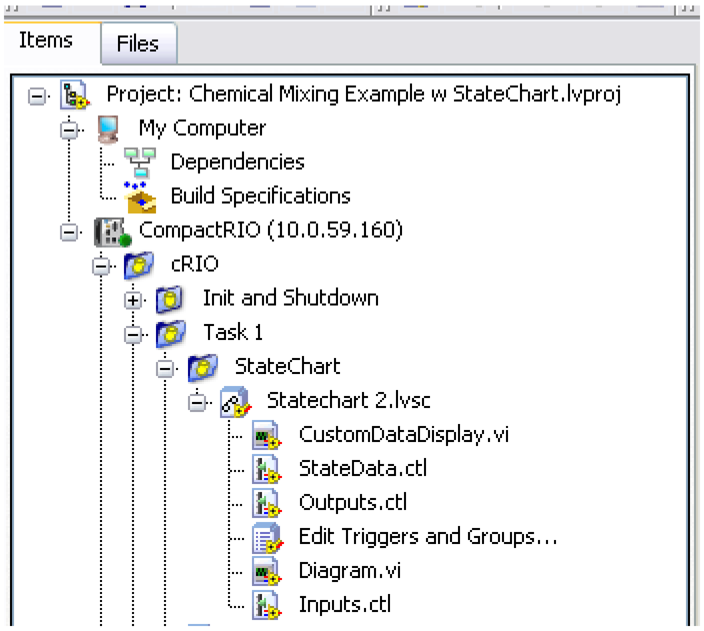
The top-level VI in this application is the Caller VI. It features a timed loop that continuously calls your statechart. Additionally, the top-level VI includes code sections for startup and shutdown. This is encapsulated within a sequence structure.



Top Level.vi Block Diagram on Chemical Mixing Example w StateChart.lvproj/CompactRIO *
File Edit View Project Operate Tools Window Help
18pt Application Font
Initialization Routine
scans
Synchron
dt
100
Error
Shutdown Routine
Stop
Chemical Mixing Example w StateChart.lvproj/CompactRIO

Figure 2.31. The top-level VI includes a timed loop that continuously calls your statechart. It also includes code sections for startup and shutdown, which are encapsulated within a sequence structure.

Now add a new statechart into the LabVIEW Project. Each LabVIEW statechart has several components that you can use to configure the context of the design.



Items Files

- Project: Chemical Mixing Example w StateChart.lvproj
  - My Computer
    - Dependencies
    - Build Specifications
  - CompactRIO (10.0.59.160)
    - cRIO
      - Init and Shutdown
      - Task 1
        - StateChart
          - Statechart 2.lvsc
            - CustomDataDisplay.vi
            - StateData.ctl
            - Outputs.ctl
            - Edit Triggers and Groups...
            - Diagram.vi
            - Inputs.ctl

Figure 2.32. Adding a New Statechart into the LabVIEW Project

The diagram.vi file contains the actual statechart diagram. The inputs.ctl and outputs.ctl are clusters that define the inputs and outputs to the statechart. The statedata.ctl is for internal state information only used in the statechart. For this example, do not use the triggers, statedata.ctl, or customdatadisplay.vi.

Step 2. Define the Inputs, Outputs, and Triggers

Open, modify, and save inputs.ctl and outputs.ctl to create an input and output for each I/O point. The outputs.ctl contains an error cluster in the event that your statechart throws an error condition.

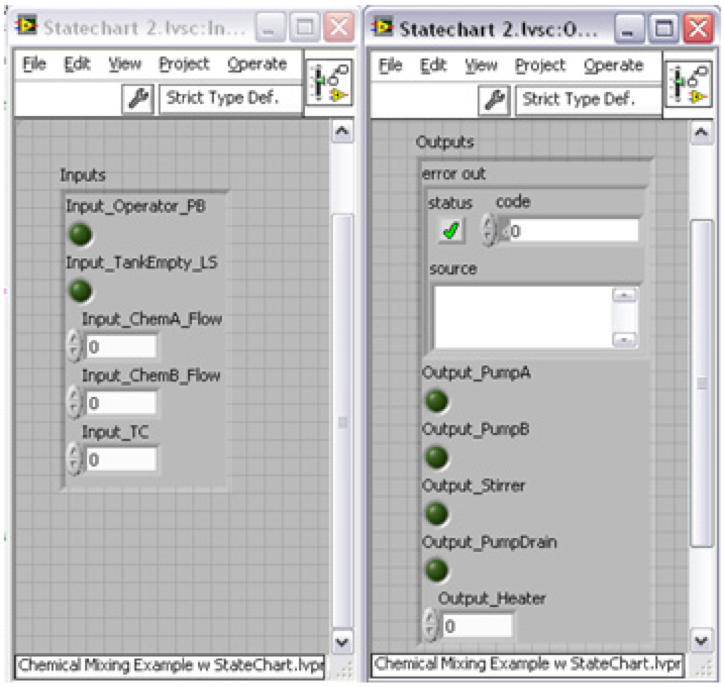


Figure 2.33. Opening, Modifying, and Saving inputs.ctl and outputs.ctl to Create an Input and Output for Each I/O Point.

Step 3. Develop a Statechart Diagram

Now open the diagram.vi file. Within this diagram, you create the states of the system and the transitions between them. Create the appropriate states, regions, and transitions to represent your logic. Each state and transition contains LabVIEW code that executes when active. The statechart is an asynchronous statechart that uses guards to determine when to transition between states. One of the main benefits of statecharts is how they visually represent the behavior of the system and, therefore, self-document the software.

Step 4. Place the Statechart in the Caller VI

Once you are done with your diagram, click on the icon on the upper left to have LabVIEW generate code for the statechart.

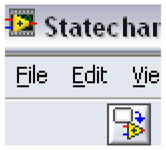


Figure 2.34. Click on the icon on the upper left corner to generate code for the statechart using LabVIEW.

In your main application, drag the statechart to the logic portion of the code. Because the statechart requires inputs and outputs through clusters, you need to create subVIs to read the I/O aliases and pass them in and out of statechart clusters. Your subVI for outputs checks error conditions before writing to the variables. If an error occurs, the error is propagated through without writing to the variable location.

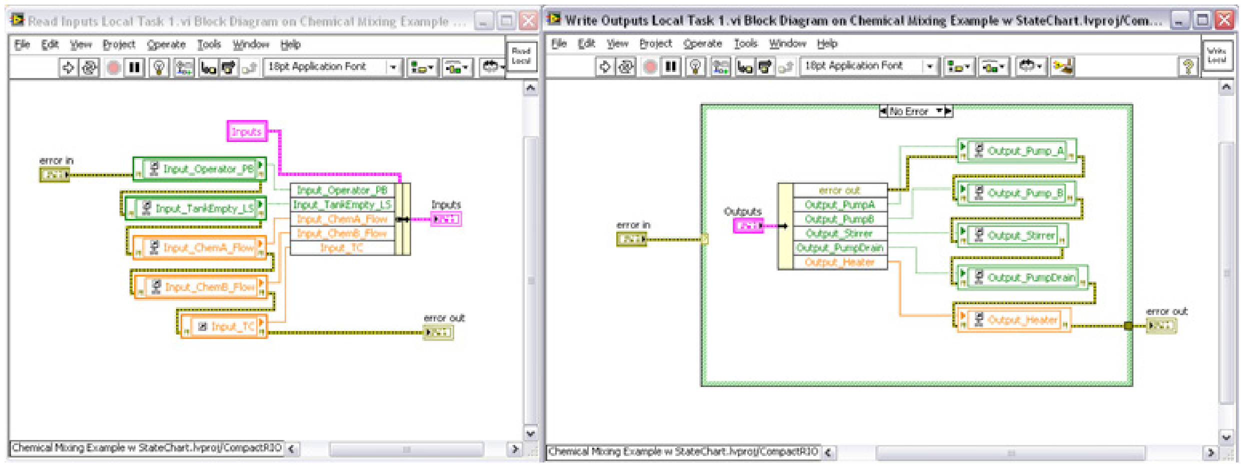

Read Inputs Local Task 1.vi Block Diagram on Chemical Mixing Example ...
File Edit View Project Operate Tools Window Help
18pt Application Font
Inputs
error in
Input_Operator_PB
Input_TankEmpty_LS
Input_ChemA_Flow
Input_ChemB_Flow
Input_TC
Inputs
error out
Chemical Mixing Example w StateChart.lvproj/CompactRIO
Write Outputs Local Task 1.vi Block Diagram on Chemical Mixing Example w StateChart.lvproj/Com...
No Error
error in
Outputs
error out
Output_PumpA
Output_PumpB
Output_Stirrer
Output_PumpDrain
Output_Heater
Output_Pump_A
Output_Pump_B
Output_Stirrer
Output_PumpDrain
Output_Heater
error out
Chemical Mixing Example w StateChart.lvproj/CompactRIO

Figure 2.35. Create subVIs to read the I/O aliases and pass them in and out of statechart clusters.

Finally drop and wire everything onto your main VI. The statechart is placed within a conditional structure that checks whether an error has occurred. If an error has occurred, the execution of the statechart is skipped. This allows reliable error-checking results, ensuring proper behavior and execution within your control system. You enable statechart debugging by right-clicking and going to properties. By doing this, you can visually debug the statechart through LabVIEW execution highlighting and through standard debugging elements such as breakpoints, probes (variable watch windows), and single-stepping. Be sure to disable debugging before deployment for best performance.

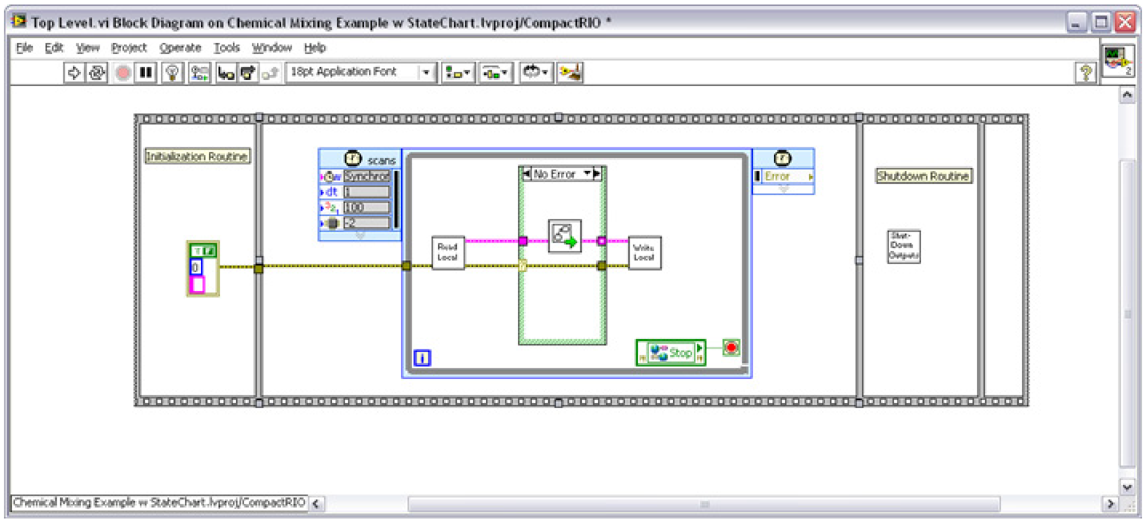

Top Level.vi Block Diagram on Chemical Mixing Example w StateChart.lvproj/CompactRIO *
File Edit View Project Operate Tools Window Help
18pt Application Font
Initialization Routine
scans
Synchron
dt 1
100
-2
No Error
Read Local
Write Local
Stop
Error
Shutdown Routine
Shut-Down Outputs
Chemical Mixing Example w StateChart.lvproj/CompactRIO

Figure 2.36. If there is no error, the statechart executes.

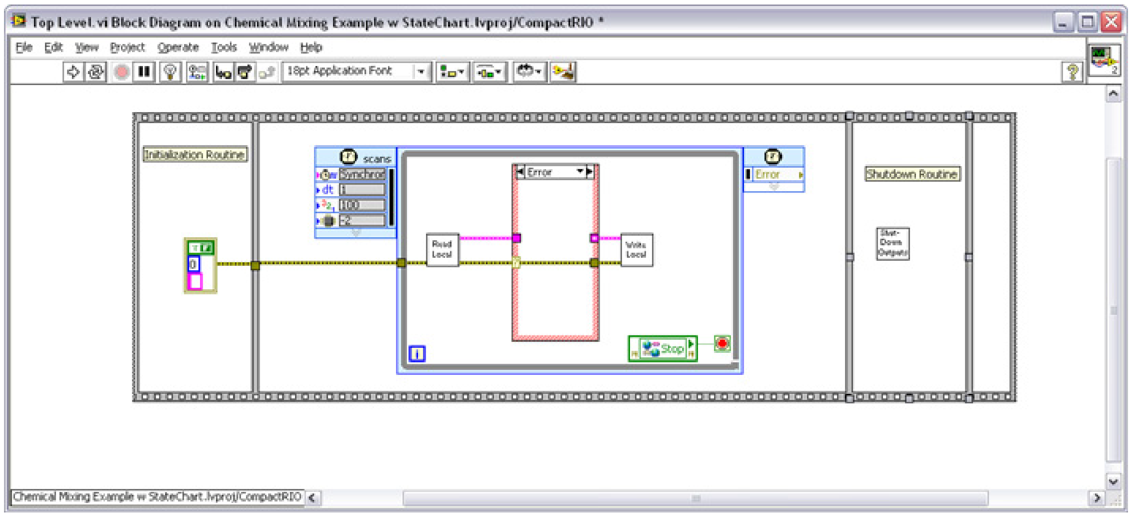

Top Level.vi Block Diagram on Chemical Mixing Example w StateChart.lvproj/CompactRIO *
File Edit View Project Operate Tools Window Help
18pt Application Font
Initialization Routine
scans
Synchron
dt
100
-2
Error
Read Local
Write Local
Stop
Error
Shutdown Routine
Shut-Down Outputs
Chemical Mixing Example w StateChart.lvproj/CompactRIO

Figure 2.37. If an error occurs, the statechart does not execute.

Getting Started – Modifying an Example

LabVIEW Example Code
is provided for this section

The easiest way to get started with this design is to modify an existing example. In this section, walk through modifying the previous chemical mixing example where you used a statechart to build your own application. There are four main steps:

1. Modify the IO Library to create IOV aliases for the physical I/O for your application
2. Modify the shutdown routine to write the shutdown values for your physical outputs
3. Modify Task 1 to read and write I/O from the statechart
4. Modify/rewrite the statechart to fit your application

Step 1. Modify the IO Library

Open the Chemical Mixing.lvproj. Because your application uses different I/O, you need to modify the IO Library to create IOV aliases for your physical I/O. If you have finalized your wiring, you can map the IOV aliases to the physical I/O now. If you have not finalized the wiring, you can remap the IOV aliases later. Expand the IO Library in the project. You can edit the variables one at a time by double-clicking on the alias. For a faster method, use the Multiple Variable Editor. You can open the editor by right-clicking on the library and selecting "Multiple Variable Editor…"

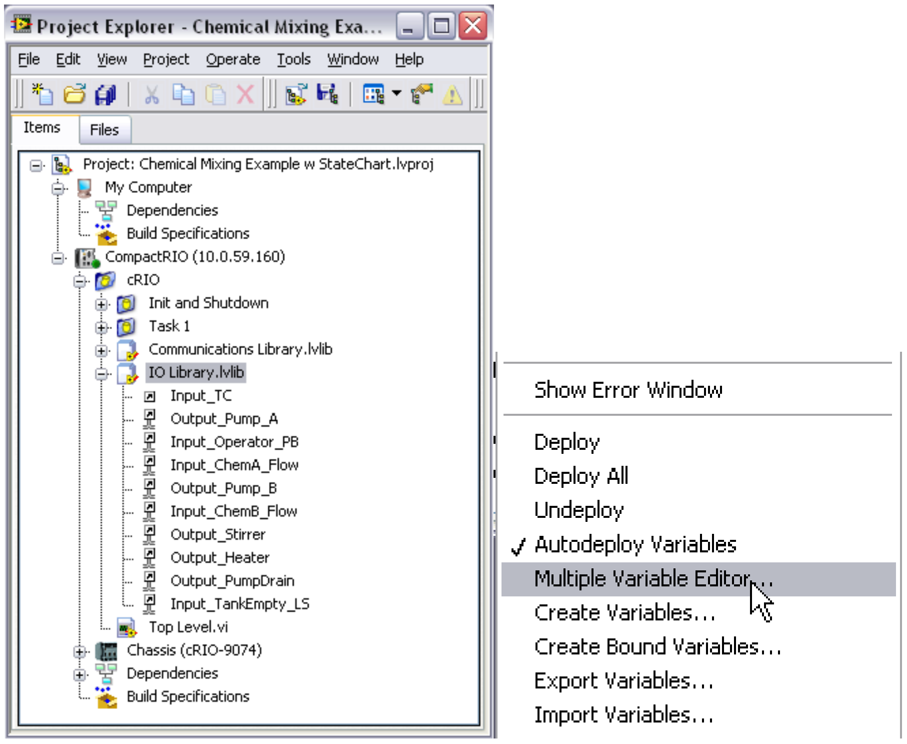
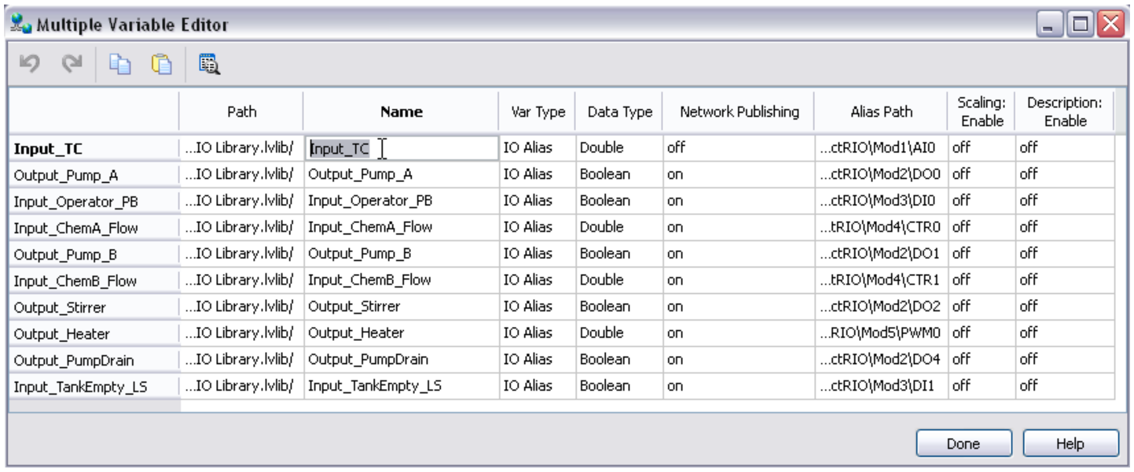


Figure 2.38. You can edit variables faster with the Multiple Variable Editor.

1. In the Multiple Variable Editor, change the names, data types, and physical bindings (alias path) of variables. You can also quickly create new variables by copying and pasting existing variables.



	Path	Name	Var Type	Data Type	Network Publishing	Alias Path	Scaling: Enable	Description: Enable
Input_TC	...IO Library.lvlib/	Input_TC	IO Alias	Double	off	...ctRIO\Mod1\AI0	off	off
Output_Pump_A	...IO Library.lvlib/	Output_Pump_A	IO Alias	Boolean	on	...ctRIO\Mod2\DO0	off	off
Input_Operator_PB	...IO Library.lvlib/	Input_Operator_PB	IO Alias	Boolean	on	...ctRIO\Mod3\DI0	off	off
Input_ChemA_Flow	...IO Library.lvlib/	Input_ChemA_Flow	IO Alias	Double	on	...tRIO\Mod4\CTR0	off	off
Output_Pump_B	...IO Library.lvlib/	Output_Pump_B	IO Alias	Boolean	on	...ctRIO\Mod2\DO1	off	off
Input_ChemB_Flow	...IO Library.lvlib/	Input_ChemB_Flow	IO Alias	Double	on	...tRIO\Mod4\CTR1	off	off
Output_Stirrer	...IO Library.lvlib/	Output_Stirrer	IO Alias	Boolean	on	...ctRIO\Mod2\DO2	off	off
Output_Heater	...IO Library.lvlib/	Output_Heater	IO Alias	Double	on	...RIO\Mod5\PWM0	off	off
Output_PumpDrain	...IO Library.lvlib/	Output_PumpDrain	IO Alias	Boolean	on	...ctRIO\Mod2\DO4	off	off
Input_TankEmpty_LS	...IO Library.lvlib/	Input_TankEmpty_LS	IO Alias	Boolean	on	...ctRIO\Mod3\DI1	off	off

Figure 2.39. Multiple Variable Editor Options

Step 2. Modify the Shutdown Routine

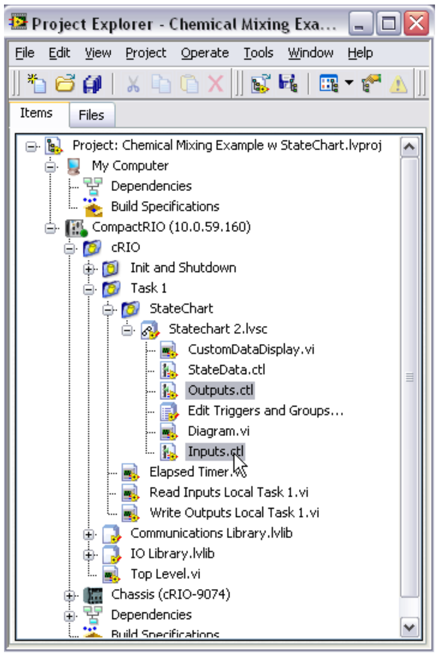
Because your application uses different I/O, you need to modify the shutdown routine to set the shutdown values for your outputs.

- Open the Shutdown Outputs.vi and modify it to set the default output values for the IOV aliases you created.

Step 3. Modify Task 1 to Map the I/O

Now you need to modify your logic. Because each logic task creates a local copy of I/O for its execution, you need to remap the I/O.

In the statechart folder, open the outputs.ctl and inputs.ctl files. Modify these to match the I/O for your application.


Project Explorer - Chemical Mixing Exa...
File Edit View Project Operate Tools Window Help
Items Files

```
Project: Chemical Mixing Example w StateChart.lvproj
  My Computer
    Dependencies
    Build Specifications
  CompactRIO (10.0.59.160)
    cRIO
      Init and Shutdown
      Task 1
        StateChart
          Statechart 2.lvsc
            CustomDataDisplay.vi
            StateData.ctl
            Outputs.ctl
            Edit Triggers and Groups...
            Diagram.vi
            Inputs.ctl
        Elapsed Timer.vi
        Read Inputs Local Task 1.vi
        Write Outputs Local Task 1.vi
      Communications Library.lvlib
      IO Library.lvlib
      Top Level.vi
    Chassis (cRIO-9074)
    Dependencies
    Build Specifications
```

Figure 2.40. Open the outputs.ctl and inputs.ctl files. Modify these to match the I/O for your application.

- Update the Write Outputs Local Task 1.vi and Read Outputs Local Task 1.vi to read and write your I/O.

Step 4. Modify/Rewrite the Statechart

Now you need to enter your logic. You can do this by modifying/rewriting the statechart to perform your application.

CHAPTER 3
Software Techniques for Scalable Systems

Reusable Functions

When designing machine control code, it is ideal to make sections of code reusable. This saves you development time because you can modularize your code within a project and build a library of code that you can use in future projects. In other development environments, these reusable pieces of code are called functions or function blocks. To be reusable, code has three primary requirements:

1. There must be a method to call the code and provide input and output data
2. The code must maintain its own memory space so it can retain state (this may not be required on some functions)
3. The code must be capable of having multiple instances in one program

Building Reusable Code in LabVIEW

In LabVIEW, reusable sections of code are called subVIs. LabVIEW is a hierarchical language designed to make code reuse easy by using reentrant subVIs. To create the three components of reusable code:

1. There must be a method to call the code and provide input and output data.
   - In LabVIEW, you can do this by creating any inputs and outputs on the front panel and connecting these controls and indicators to the connector pane.

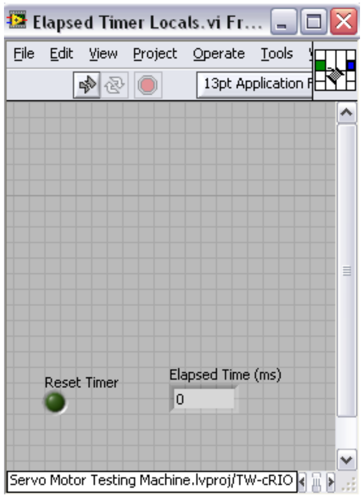

Elapsed Timer Locals.vi Fr...
File Edit View Project Operate Tools
13pt Application F
Reset Timer
Elapsed Time (ms)
0
Servo Motor Testing Machine.lvproj/TW-cRIO

Figure 3.1. On the connector pane, wire inputs and outputs to front panel controls and indicators.

2. The code must maintain its own memory space so it can retain state (this may not be required on some functions).
   - In LabVIEW, you can do this one of two ways. You can use a while loop with uninitialized shift registers to hold memory or you can create local variables. Local variables have slightly more overhead but are more flexible and easier to understand. You can create local variables from front panel controls and indicators. Right click on the control and create a local variable. This variable can be referenced multiple times on the block diagram.

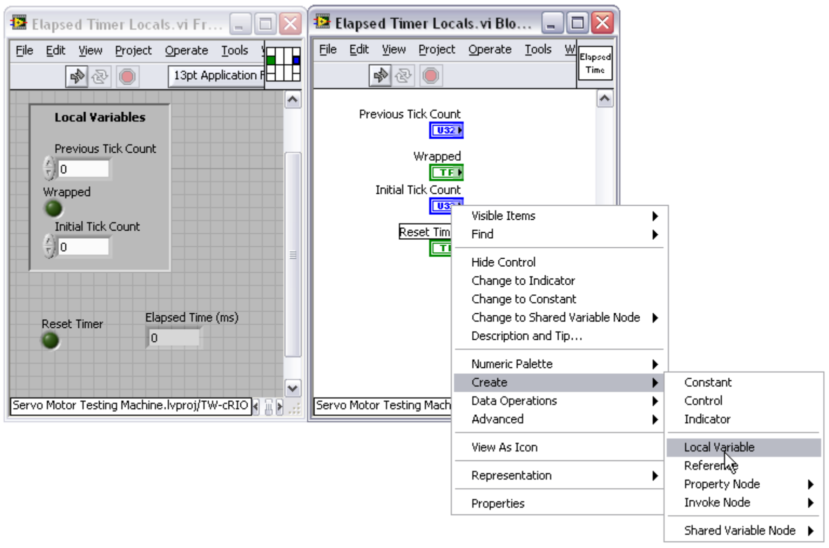
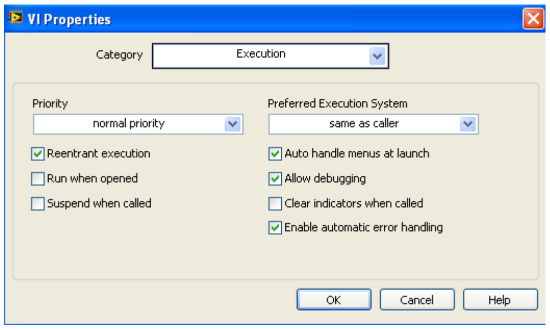


Figure 3.2. Right-click on a control or indicator to create local variables.

3. The code must be capable of having multiple instances in one program.
   - In LabVIEW, do this by making the VI reentrant. A reentrant VI has a separate memory space for each instance it is called in a program. To make a subVI reentrant, go to the VI Properties page (under File), select Execution on the Category pull-down menu, and check the box for Reentrant execution.



VI Properties

Category: Execution

Priority: normal priority

Preferred Execution System: same as caller

- [✓] Reentrant execution
- [ ] Run when opened
- [ ] Suspend when called
- [✓] Auto handle menus at launch
- [✓] Allow debugging
- [ ] Clear indicators when called
- [✓] Enable automatic error handling

OK Cancel Help

Figure 3.3. Making a VI Reentrant

Example of Building Reusable Code in LabVIEW

Example LabVIEW Example Code is provided for this section

If you look at the previous chemical mixing example, one of the requirements was to hold the mixture at a set temperature for a specific period of time. Because a control application must remain responsive, you cannot use "wait" statements to control the timing of an application. If you do, the rest of the control algorithm does not run while you are waiting, and you have an unresponsive application. Because you cannot put a wait statement into the loop, you need a method where at every iteration, you can check the elapsed time in that state. This is a common requirement and is an ideal application for reusable code.

To learn how to build custom reusable code, create a subVI to determine elapsed time. The function should output the elapsed time and have an input to reset the timer. In LabVIEW, there is a Tick Count function that reads a microsecond counter. The Tick Count function outputs a U32 value of microseconds. The following is the logic for the elapsed time subVI:

- Check to see if this is the first time this instance of the VI has been run or if the reset counter input is true. If so, read the Tick Count and store that as the initial tick count, set the elapsed time output to 0, and write a false to the wrapped register.
- Check to see if the Tick Count output has wrapped (if the output exceeds the available space in a U32, it starts over at 0) by comparing to the last Tick Count output. If the Tick Count has wrapped, set the wrapped register to true.
- Subtract the current Tick Count from the original Tick Count. If the count has wrapped, convert to U64, add 2^32 -1, subtract the original tick count, and convert back to U32.

1. There must be a method to call the code and provide input and output data.
   - Create a new VI. On the front panel create a control for "reset" and an indicator for "elapsed time." Connect these to the connector pane.

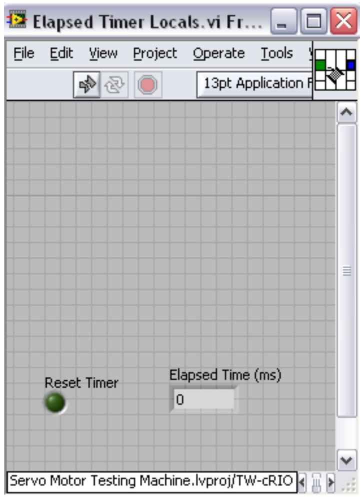

Elapsed Timer Locals.vi Fr...
File Edit View Project Operate Tools
13pt Application F
Reset Timer
Elapsed Time (ms)
0
Servo Motor Testing Machine.lvproj/TW-cRIO

Figure 3.4. Create a new VI to call the code and provide input and output data.

2. The code must maintain its own memory space.
   - On the front panel, create controls and indicators for the three locals: Previous Tick Count, Wrapped, and Initial Tick Count.

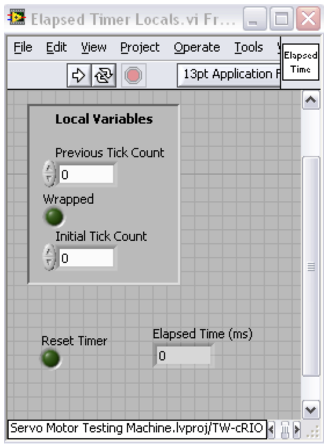
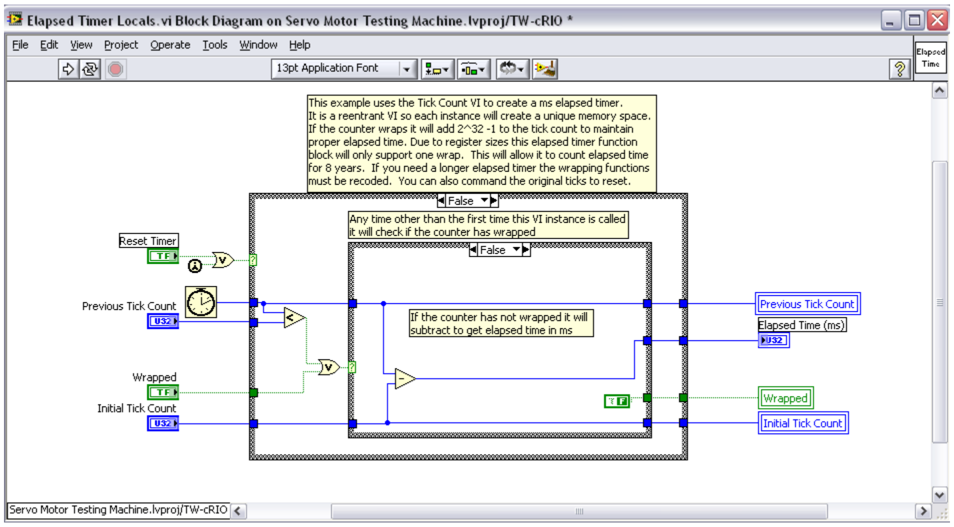


Figure 3.5. Create controls and indicators for the three locals: Previous Tick Count, Wrapped, and Initial Tick Count.

3. The code must be capable of having multiple instances in one program.
   - Go to the Properties window and make the VI reentrant.

Now on the block diagram, simply write the logic for the elapsed timer. When you need to access local data, right-click on the control or indicator and create a local variable. You can now debug this code and then reuse it throughout multiple control programs.


Elapsed Timer Locals.vi Block Diagram on Servo Motor Testing Machine.lvproj/TW-cRIO *
File Edit View Project Operate Tools Window Help
13pt Application Font
Elapsed Time
False
False
Reset Timer
Previous Tick Count
Wrapped
Initial Tick Count
Previous Tick Count
Elapsed Time (ms)
Wrapped
Initial Tick Count
Servo Motor Testing Machine.lvproj/TW-cRIO

This example uses the Tick Count VI to create a ms elapsed timer.
It is a reentrant VI so each instance will create a unique memory space.
If the counter wraps it will add 2^32 -1 to the tick count to maintain
proper elapsed time. Due to register sizes this elapsed timer function
block will only support one wrap. This will allow it to count elapsed time
for 8 years. If you need a longer elapsed timer the wrapping functions
must be recoded. You can also command the original ticks to reset.

Any time other than the first time this VI instance is called
it will check if the counter has wrapped

If the counter has not wrapped it will
subtract to get elapsed time in ms

Figure 3.6. Finished Code

Other Reusable Code in LabVIEW

NI ships LabVIEW with an extensive library of reusable code that you can access from the **Functions** palette. This code provides hundreds of built-in functions for control, analysis, communications, file I/O, and more.

IEC 61131 Function Blocks

LabVIEW 8.6 introduced a new type of reusable code called function blocks. These function blocks are based on the IEC 61131-3 international standard for programming industrial control systems. They are written in LabVIEW, designed for use in real-time applications, and have the ability to publish their parameters in the memory table (as shared variables). You can use these function blocks with all other LabVIEW code.

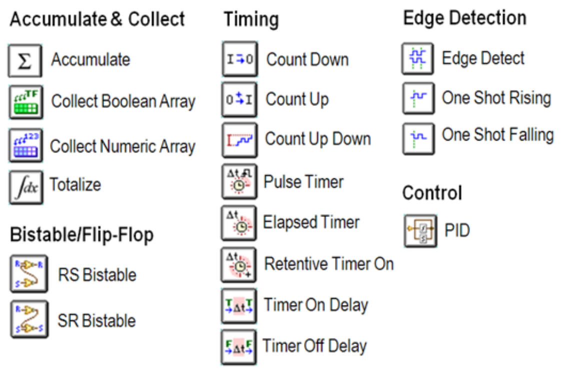
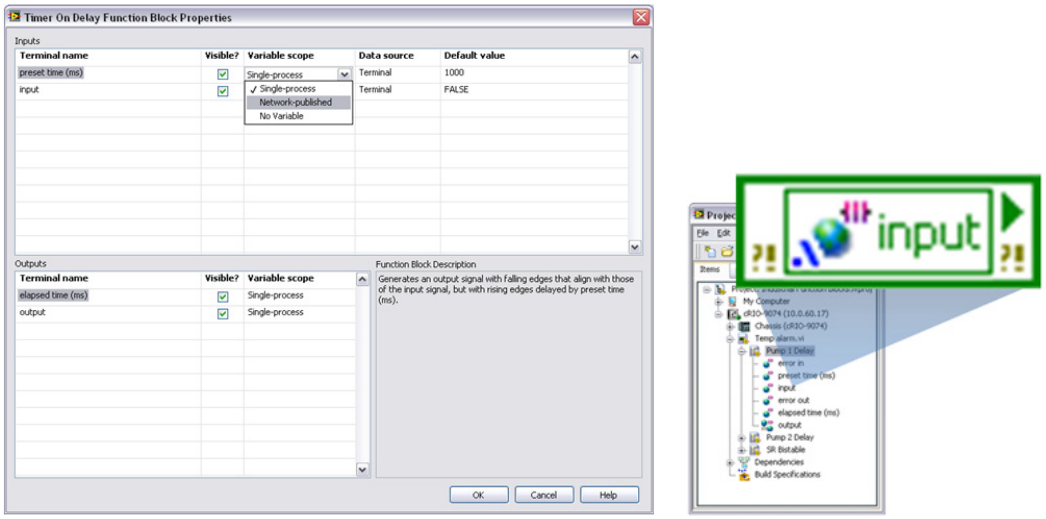


Figure 3.7. New LabVIEW Function Blocks Based on the IEC 61131-3 International Standard for Programming Industrial Control Systems

Configurable Terminal Variables

Function blocks differ from standard subVIs by offering configuration pages and providing the option to directly connect the inputs and outputs to entries in the global memory table that are visible in the LabVIEW Project. You also can access these entries through the network. You can configure the terminals and variables from the function block Properties window.



Timer On Delay Function Block Properties

Inputs

Terminal name	Visible?	Variable scope	Data source	Default value
preset time (ms)	✓	Single-process	Terminal	1000
input	✓	✓ Single-process / Network-published / No Variable	Terminal	FALSE

Outputs

Terminal name	Visible?	Variable scope
elapsed time (ms)	✓	Single-process
output	✓	Single-process

Function Block Description

Generates an output signal with falling edges that align with those of the input signal, but with rising edges delayed by preset time (ms).

Figure 3.8. Configure function blocks from the Properties window and access inputs and outputs from the memory table.

Multiple Tasks (Multiple Loops)

In many applications, the controller runs more than one control and measurement task. For instance, a machine control application may have a task that controls the machine operation using a statechart and a second task that performs machine health monitoring or a task that logs data. The control routine can have multiple tasks that run in parallel and pass data to the memory table.

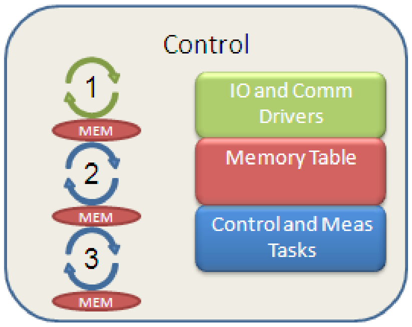


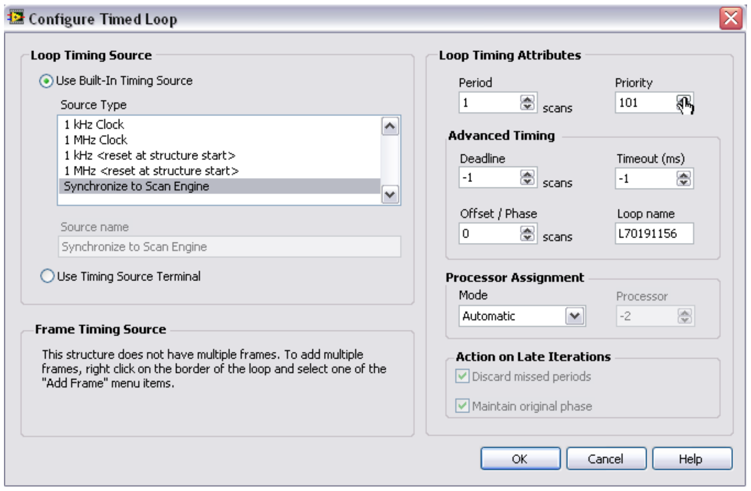
Figure 3.9. The control routine can have multiple tasks that run in parallel and pass data to the memory table.

- To manage the execution of your application, you need to be able to:Set the priority between the tasks
- Synchronize the tasks
- Pass data between the tasks
- Trigger the tasks

Setting Task Priority and Synchronizing Tasks

When running multiple tasks, you need to ensure that your control task has the highest priority. Because LabVIEW execution is based on time, a high-priority loop such as the IO Scan always runs on a set schedule. This ensures low-jitter operation and stable control. However, it also means that if the controller does not finish all the requested operations in time, those operations are interrupted. This may be OK (or even desirable) for a low-priority task like data logging or network communication. But if the control task is interrupted, it may lead to an unstable operation. Therefore, you should design your application to determine the priority of the tasks. You should also use tools like the NI Real-Time Execution Trace Toolkit to benchmark your application to ensure adequate time for background tasks like communications.

To set the priority of a task, you can use a timed loop. The timed loop has a configurable priority relative to other timed loops. The higher the number you enter, the higher the priority. The value for the priority must be a positive integer between 1 and 65,535. The LabVIEW execution system is preemptive, so a higher-priority timed structure that is ready to execute preempts all lower-priority structures that are also ready to execute and other LabVIEW code not running at time-critical priority.


Configure Timed Loop

Loop Timing Source

(●) Use Built-In Timing Source

Source Type
- 1 kHz Clock
- 1 MHz Clock
- 1 kHz <reset at structure start>
- 1 MHz <reset at structure start>
- Synchronize to Scan Engine

Source name: Synchronize to Scan Engine

( ) Use Timing Source Terminal

Frame Timing Source

This structure does not have multiple frames. To add multiple frames, right click on the border of the loop and select one of the "Add Frame" menu items.

Loop Timing Attributes

Period: 1 scans

Priority: 101

Advanced Timing

Deadline: -1 scans

Timeout (ms): -1

Offset / Phase: 0 scans

Loop name: L70191156

Processor Assignment

Mode: Automatic

Processor: -2

Action on Late Iterations

[✓] Discard missed periods

[✓] Maintain original phase

OK Cancel Help

Figure 3.10. Set the priority of a timed loop.

To synchronize multiple tasks, set them all to synchronize to the NI Scan Engine. All loops synchronized to the scan engine run at the I/O scan rate and execute from highest priority to lowest priority. If you have background tasks or nondeterministic tasks that do not need synchronization or priority, you can run those tasks in a standard while loop with a wait function to set timing.

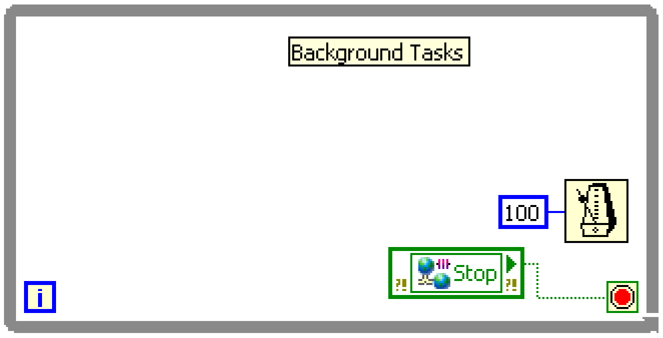


Figure 3.11. While loops operate at normal priority.

Passing Data between Tasks

All tasks can read and write I/O from the memory table. To pass data between tasks, you need to add a new set of data into the memory table, which also features a component called the LabVIEW shared variable. With LabVIEW shared variables, you can share data globally within a controller or across a network. They are configurable, and you can use them to provide functionality within a controller and across the network. For now, focus on using shared variables to put data in the controller memory table.

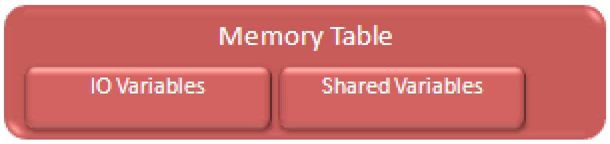
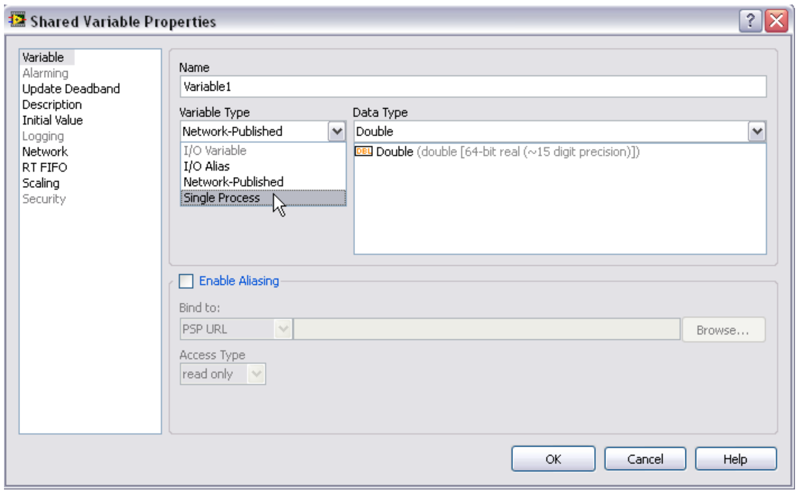


Figure 3.12. Shared variables and I/O variables are elements of the memory table.

Creating a new shared variable is similar to creating an I/O alias. You can organize these variables by using libraries. First create a new library in the project and then create a new variable. Select the data type and set the variable type as Single Process (a single-process shared variable is a global variable).



Name: Variable1

Variable Type: Network-Published

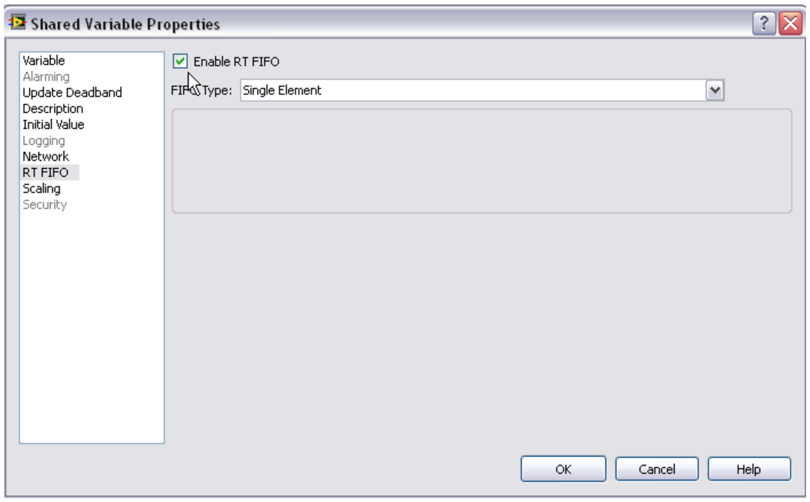
I/O Variable
I/O Alias
Network-Published
Single Process

Data Type: Double

Double (double [64-bit real (~15 digit precision)])

Figure 3.13. Creating a Shared Variable

Next go to the RT FIFO tab. Some variables, such as arrays, cannot be read or written in a single processor operation. When loops become preempted by higher-priority loops, these unfinished operations can cause increased processor usage and jitter. To avoid this, enable the real-time FIFO and set it to Single Element.



☑ Enable RT FIFO

FIFO Type: Single Element

Figure 3.14. By enabling the real-time FIFO, you can read and write to a shared variable from multiple parallel loops without inducing jitter.

You can now read and write to this memory table element from anywhere in your control code, just like I/O aliases.

Triggering Tasks

Sometimes it is ideal to trigger a task from another task. For instance, the top-level machine control code may manage the machine and, in one state, need to start a waveform acquisition and analysis operation. This waveform operation needs to take place in a lower-priority parallel task. To trigger parallel tasks, use a command-based architecture.

Command-Based Architecture for Multiloop Systems

At its simplest, the command-based architecture can be described as a model where two entities, a commander and a worker, are related by a message, the command. The commander relays the action to be performed to the worker via a message and the worker acts on it.

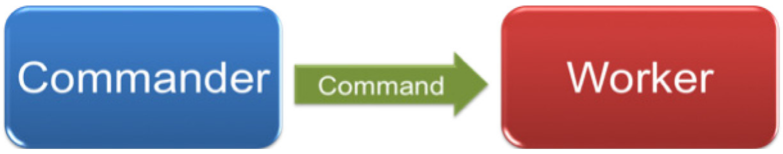


Figure 3.15. A command-based architecture provides a mechanism to send commands between parallel tasks.

Other terms have been used to describe this model, including master-slave, server-client, and producer-consumer models, with slight variations in their meanings and implementations.

There are numerous methods to pass a command between parallel loops in LabVIEW including real-time FIFOs, queues, or shared variables.

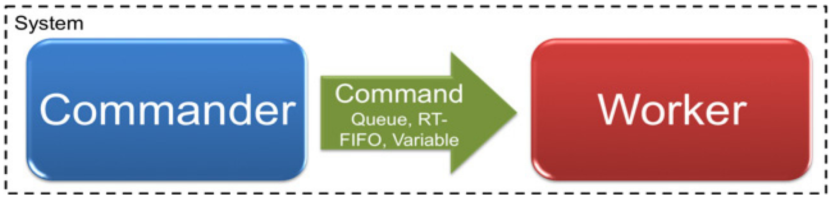
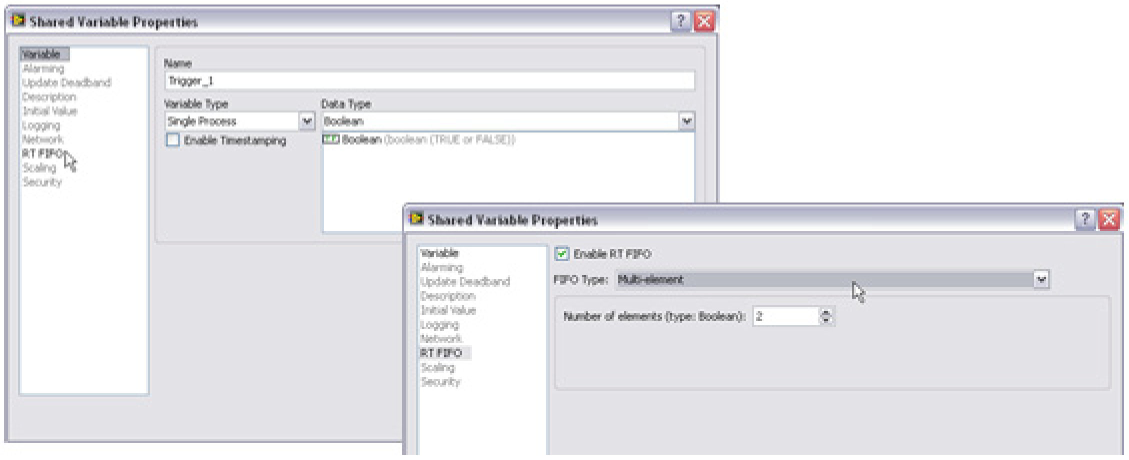


Figure 3.16. You can send a command in LabVIEW with several methods, including queues, real-time FIFOs, and variables.

Shared Variables for Commands

Shared variables offer a scalable mechanism to send commands between parallel loops for operations such as triggering. The shared variable must be real-time safe and provide a mechanism to buffer commands in a FIFO.

To do this, create a single-process shared variable and, on the RT FIFO tab, enable the RT FIFO and set the type to multi-element.



Name: Trigger_1
Variable Type: Single Process
Data Type: Boolean
Enable Timestamping
Boolean (boolean (TRUE or FALSE))

Enable RT FIFO
FIFO Type: Multi-element
Number of elements (type: Boolean): 2

Figure 3.17. A single-process shared variable with RT FIFO set to Multi-element is an effective way to pass commands between loops.

A multi-element FIFO features a buffer where each write adds another element to the buffer and each read removes an element from the buffer. As long as the worker loop checks the FIFO frequently, you can leave the buffer at the default of 2 elements and not miss any triggers/commands.

To check the FIFO, the worker task must read the shared variable and check the error status. If the FIFO is empty, the shared variable returns a warning -2220. If this warning is not returned, then the FIFO was not empty and the returned value is a valid command.

Each time the shared variable is read, one element is removed from the FIFO (assuming the FIFO is not empty). Because of this, you cannot have multiple workers receive a command from the same FIFO, but you can have multiple commanders putting commands into the FIFO.

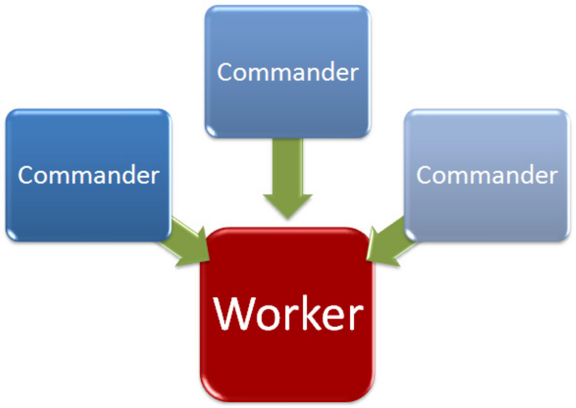


Figure 3.18. You can have multiple writers to a command but you should create only one reader.

Numeric Commands

Commands can also be more complex than a simple trigger. The shared variable supports multiple data types and if it is set to a numeric it can be used to pass a number of different commands. An enumerated type definition, in the form of LabVIEW enum, is a great way to define commands that can be directly passed to the shared variable.

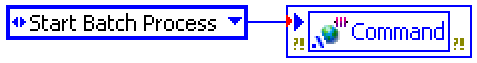


Figure 3.19. Wiring an Enum to a Numeric Shared Variable Command

The enum type definition provides an ordered list of the available commands. If you make it a type definition, changes automatically propagate through the code.

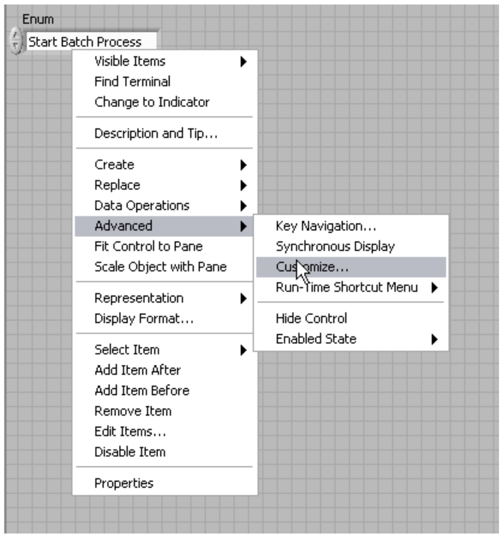


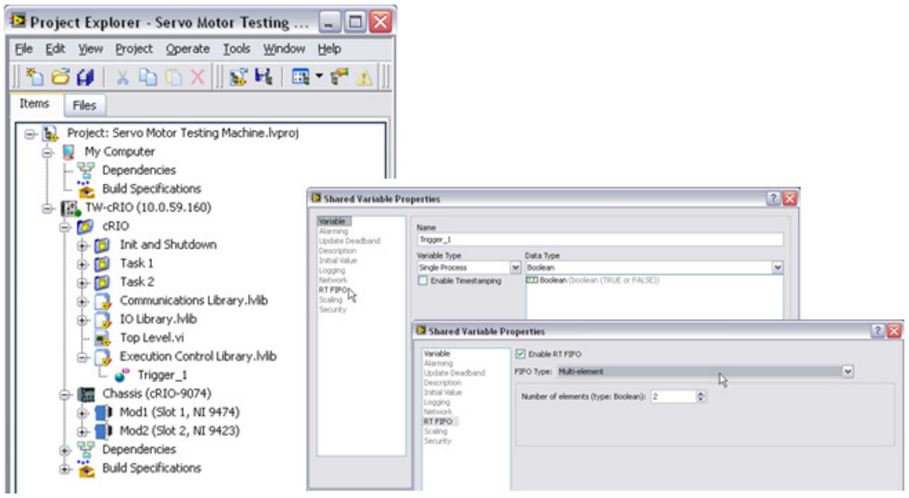
Figure 3.20. If you create a type definition, any changes to the command list automatically propagate through the code.

Example of Using Shared Variables to Trigger a Parallel Loop

LabVIEW Example Code
is provided for this section

Now modify the servo motor testing machine code so that when the statechart sets the Output_Rotating IOV Alias to true, it triggers a parallel loop.

First, create the shared variable. In this example, it is a simple trigger, so create a Boolean variable and save it in a new library called the Execution Control Library.



Name: Trigger_1
Variable Type: Single Process
Data Type: Boolean
☐ Enable Timestamping
Boolean (boolean (TRUE or FALSE))

☑ Enable RT FIFO
FIFO Type: Multi-element
Number of elements (type: Boolean): 2

Figure 3.21. You can use a new library as a container for all commands to keep the code organized.

Then modify the Write Outputs Local Task 1.vi to write a true to Trigger_1 when the value to Output_Rotating transitions from false to true.

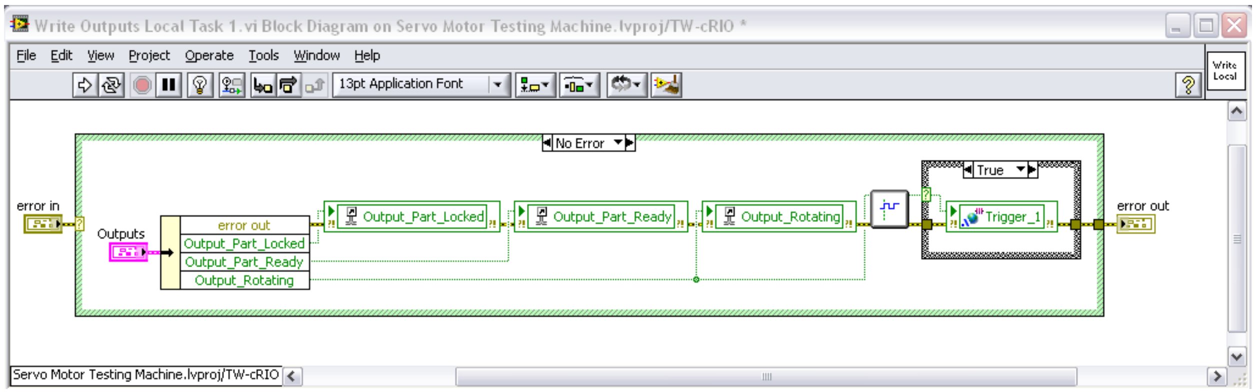

Write Outputs Local Task 1.vi Block Diagram on Servo Motor Testing Machine.lvproj/TW-cRIO *
File Edit View Project Operate Tools Window Help
13pt Application Font
No Error
error in
Outputs
error out
Output_Part_Locked
Output_Part_Ready
Output_Rotating
True
Trigger_1
error out
Servo Motor Testing Machine.lvproj/TW-cRIO

Figure 3.22. Unlike normal I/O, write values to the shared variable used for commands only when you want to trigger the parallel loop.

Finally, create a normal priority parallel loop to be the worker loop that reads Trigger_1 and filters the output based on checking the error cluster for warning -2220. Because you likely want to reuse this code, create a reentrant subVI that checks for the warning and save it as Empty FIFO Filter.vi.

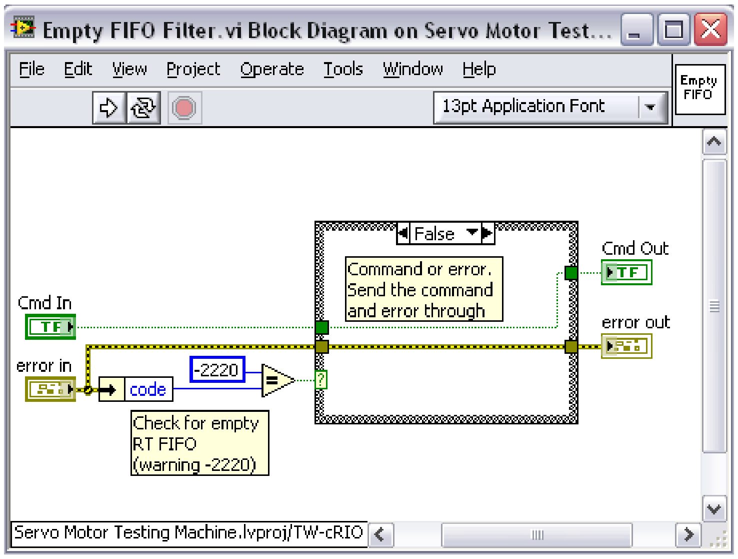

Empty FIFO Filter.vi Block Diagram on Servo Motor Test...
File Edit View Project Operate Tools Window Help
13pt Application Font
Empty FIFO
False
Command or error. Send the command and error through
Cmd Out
Cmd In
error out
error in
-2220
code
Check for empty RT FIFO (warning -2220)
Servo Motor Testing Machine.lvproj/TW-cRIO

Figure 3.23. This reusable section of code filters the output if the real-time FIFO is empty.

The parallel loop triggers logic if it receives a true from the FIFO. If it does not receive a true, it waits and then reads the shared variable again.

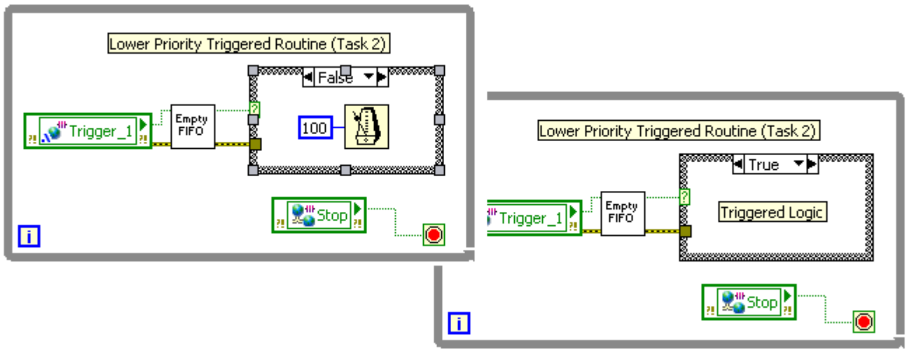

Lower Priority Triggered Routine (Task 2)
False
Trigger_1
Empty FIFO
100
Stop
Lower Priority Triggered Routine (Task 2)
True
Trigger_1
Empty FIFO
Triggered Logic
Stop

Figure 3.24. A wait statement in the false case determines the polling rate for this VI.

This is the final block diagram with the main control routine and a lower-priority task that is triggered.

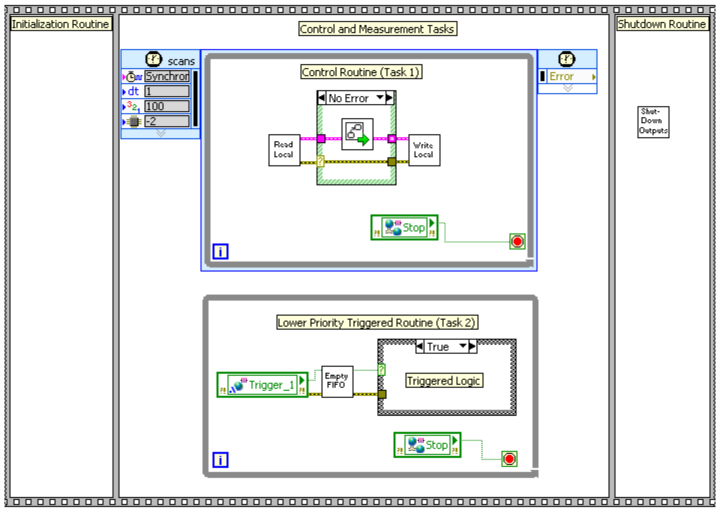


Figure 3.25. An Application with a Commander Loop (Task 1) and a Worker Loop (Task 2)

Adding Data to the I/O Scan

Example

LabVIEW Example Code
is provided for this section

The previous sections examined reading and writing I/O to the NI Scan Engine with I/O aliases and sharing data between tasks using the LabVIEW shared variable. In many applications, it is necessary to read and write to I/O that is not handled by the scan engine. For example, you may need to access the data from an RS232-based flowmeter through the serial port of an CompactRIO controller. Another example is accessing the FPGA on CompactRIO for advanced I/O requirements. This section explores an architecture for adding a custom I/O scan task to your application.

Other than I/O from the Scan Engine, you frequently need to access data from other primary I/O ports including the following:

- FPGA
- Ethernet
- Serial
- USB

Add a Custom I/O Scan Task (Driver Loop)

As discussed previously, you need to use a timed loop to add the task to the application. Place an additional timed loop in the Control and Measurement state.

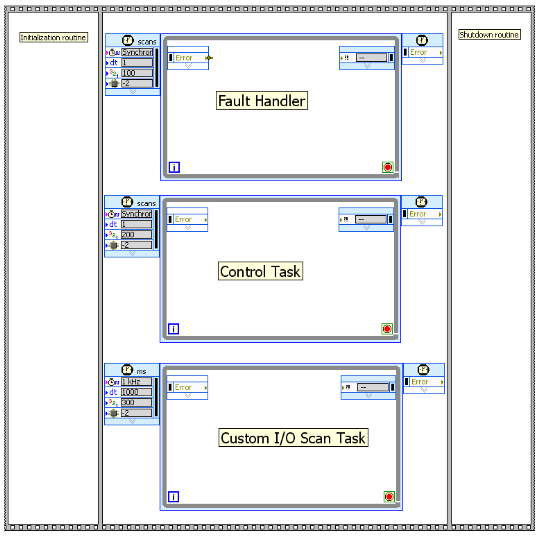


Figure 3.26. Add another timed loop to handle the custom I/O scan task.

Priority and Timing

The priority and timing of the custom I/O scan task depends on which I/O device you access and how you use the data in your application. The two primary scenarios follow.

Synchronous Deterministic I/O

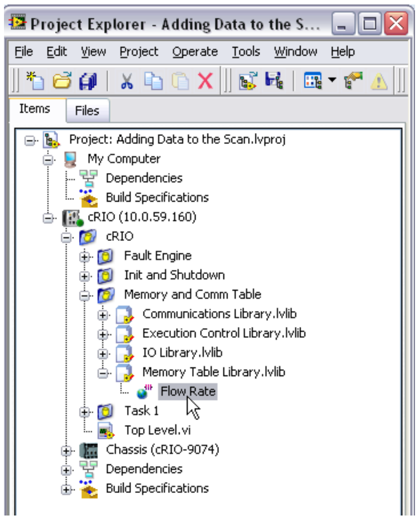
If the I/O driver is deterministic and you are using the data in each iteration of the control task, then you should synchronize the custom I/O scan task to the scan engine with a higher priority than the control task. This causes the custom I/O task to run before each iteration of the control task. It also allows the control task to run with the most recent data from the custom I/O. Accessing I/O through the LabVIEW FPGA Module is the primary use case for this scenario.

Asynchronous Nondeterministic I/O
A second scenario is accessing an I/O device that is nondeterministic such as the serial or Ethernet port. For this use case, you should assign a lower priority to the custom I/O scan task than the control task. This allows the control task to run deterministically and reliably because it is not affected by the possibly high-jitter I/O device in the custom I/O scan task. Set the task timing based on the required update rate. Read values from a serial-based instrument or network communication are common use cases for this scenario.

With the following example, learn more about the asynchronous nondeterministic I/O use case starting with reading the values from an RS232-based flowmeter. A later section addresses accessing I/O through LabVIEW FPGA.

Adding Entries to the Memory Table for Custom I/O

To access the data read and written in the custom I/O scan task throughout the application, create single-process shared variables with real-time FIFOs, as discussed in the section "Passing Data between Tasks."


Project Explorer - Adding Data to the S...
File Edit View Project Operate Tools Window Help
Items
Files

```
Project: Adding Data to the Scan.lvproj
  My Computer
    Dependencies
    Build Specifications
  cRIO (10.0.59.160)
    cRIO
      Fault Engine
      Init and Shutdown
      Memory and Comm Table
        Communications Library.lvlib
        Execution Control Library.lvlib
        IO Library.lvlib
        Memory Table Library.lvlib
          Flow Rate
      Task 1
      Top Level.vi
    Chassis (cRIO-9074)
    Dependencies
    Build Specifications
```

Figure 3.27. Use real-time FIFO shared variables to share the custom I/O data with other tasks.

Add the Custom I/O Scan Logic

After you add another timed loop, set the priority and timing, and create real-time FIFO shared variables for storing the data, you are ready to add the logic to access the I/O device. This includes the following steps:

1. Open a session with the I/O device
2. Read and write I/O
3. Close the session

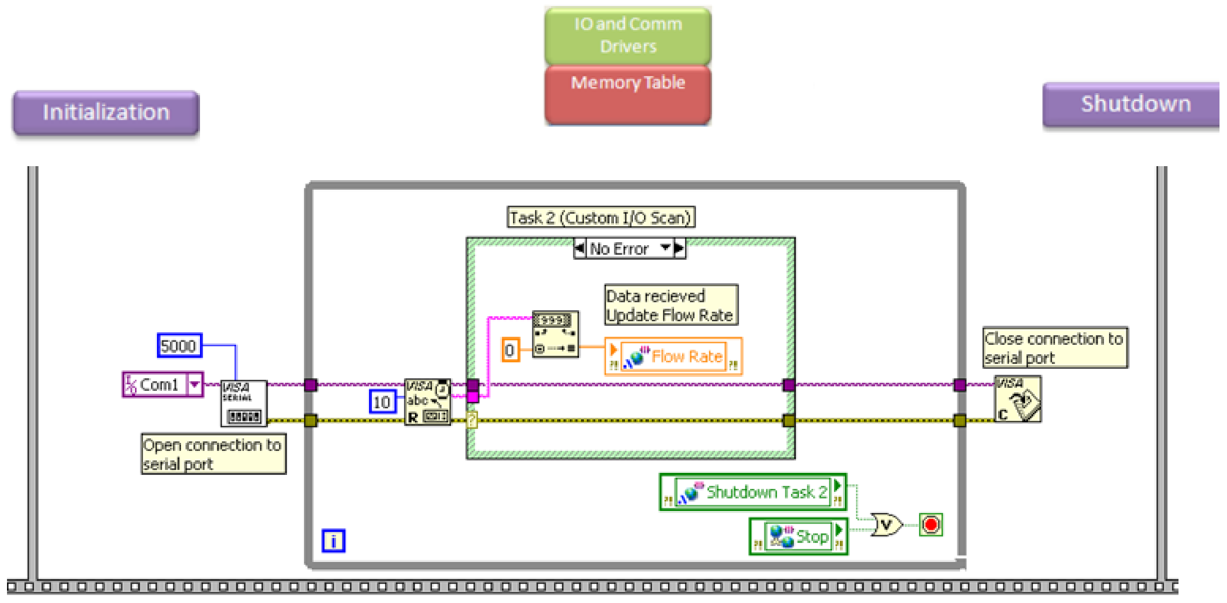


Figure 3.28. In this custom I/O scan task, you open a session, write data periodically to the real-time FIFO variable, and then close the session.

Data Logging

LabVIEW Example Code
is provided for this section

At this point, you have covered an architecture to acquire data and perform analysis and control. Another common requirement for embedded applications is to log data to disk for future analysis and review. In this section, discover how to build a simple data-logging architecture. With this architecture, the user can write the data to file at a user-defined rate, store data in a file format that is easily readable in any spreadsheet or word editing software, and transfer those data files from the CompactRIO controller to a host PC or server.

The first step you should take before coding is to ask a few questions about the application:

1. How much data needs to be stored?
2. Which storage format is required for the data?
3. What is the sampling rate for the logged data – that is, slow, single-point measurements or waveforms?

These questions are important because there are a variety of possible data logging architectures and storage mediums available including: external USB hard drives, external SD Memory cards, streaming to a server via TCP/IP, and so on. The variety of potential architectures is an example of the modular platform advantage of CompactRIO. Figure 3.29 shows a graph of the different storage media and the different data storage architectures.

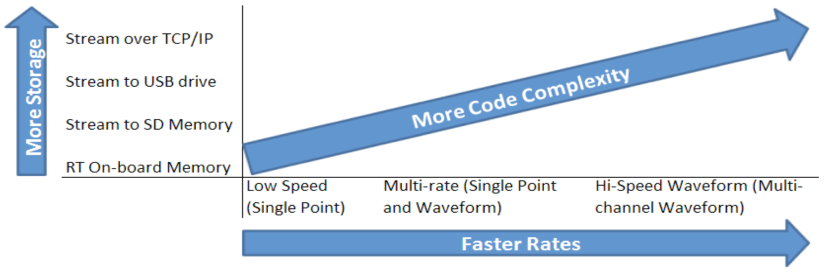


Figure 3.29. Variety of Data-Logging Architectures with CompactRIO

A CompactRIO system has built-in solid-state nonvolatile memory with up to 2 GB of space. This memory is analogous to the hard drive on a PC; however for increased reliability it has no moving parts, supports a wide temperature range, and uses a more reliable file system called Reliance. Reliance is a transactional file system, developed by Datalight, which is tolerant to power interruptions and removes the risk of file corruption that exists when writing to a FAT file system during power down. CompactRIO also offers a carrier module for removable SD memory and some CompactRIO systems have a USB port that can support USB hard drives or memory sticks.

Onboard Real-Time Memory Data Logging with FTP Transfer

In this section, focus on the most common logging architecture used with a control system and write low-speed, single-point data to a file on the onboard memory of the real-time controller. Several documents on ni.com discuss other architectures in detail, but this section focuses on an in-depth look at the low-speed, data-logging architecture consisting of the following components:

1. Read the I/O Memory Table at a user-defined rate
2. Write that data to a file on the real-time controller
3. FTP the data from the real-time controller either automatically or manually

Now walk through the development of the data-logging code that runs on the CompactRIO controller.

Initializing the File

First initialize an ASCII file to which data is written. In this step, it is important to pick a filename that is useful for determining with which test it corresponds.

Real-time targets use letters to indicate different drives just like a desktop computer. CompactRIO systems refer to internal nonvolatile memory by drive letter C and external USB drives by drive letter U. To access a file within the memory of the real-time target, use a path that begins with C:\, similar to your desktop computer. If you wish to log data gathered on the real-time target to a file on the same target, you can use the Write to Text File.vi in LabVIEW and use C:\[folder name]\[filename] for your File input parameter on the VI.

The following example uses a path constant to set the desired folder location to “c:\datalogging” and then converts it to a string. It also uses the “Get Date/Time” function to create a string of the date, in this case, “Feb 23, 01:41:36 PM.” Then, using the “Concatenate String” and “Convert string to path” functions, you create a desirable filename. Finally, input that path into the “Open/Create/Replace File” function.

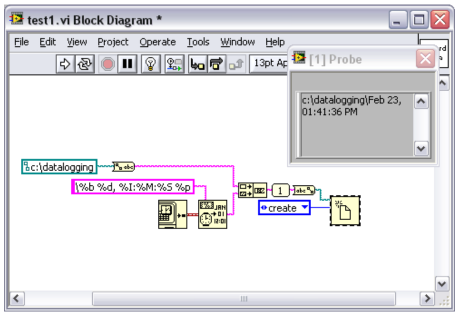

test1.vi Block Diagram *
File Edit View Project Operate Tools Window Help
13pt Ap
[1] Probe
c:\datalogging\Feb 23, 01:41:36 PM
c:\datalogging
%b %d, %I:%M:%S %p
1
create

Figure 3.30. Initializing an ASCII File with Proper Naming Convention for Streaming Data

Writing the Data to File

The next portion of the code, contained in the red box, is where you write the data to file. In this section, the current I/O values stored in the I/O Mapping Table are read at some user-defined rate and written to file. This architecture consists of a separate while loop from the other loops that reads the I/O values and writes them to file. It also consists of a timing function for ensuring that the loop executes at a user-defined rate.

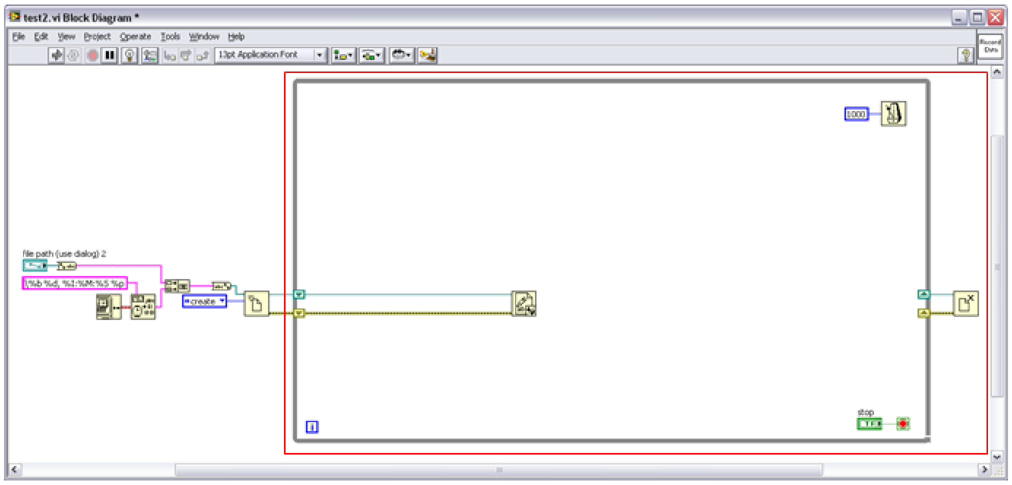


Figure 3.31. Writing the Data to File

Reading and Formatting the Data

The third step, contained in the red box, is to read the current I/O Memory Table values and format them into a string. In this step, place the shared variables, which you should write to file, inside the while loop. Then build the values into an array, convert them into an ASCII string, and write them to file.

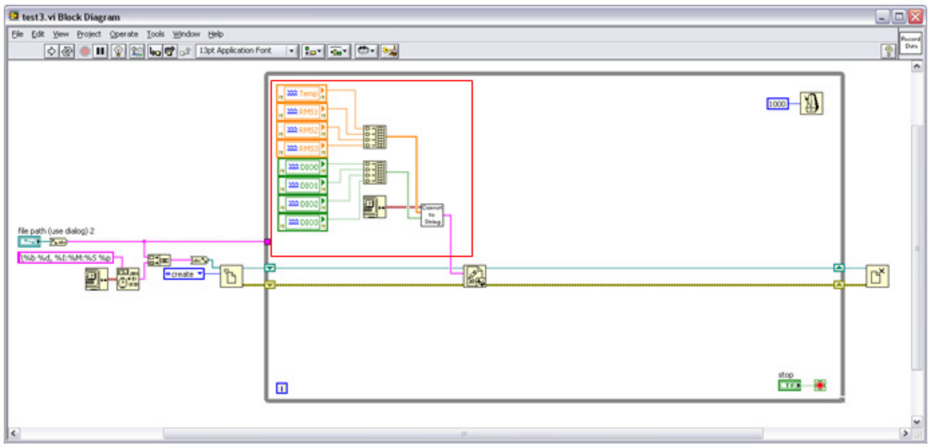


Figure 3.32. Reading and Formatting the Data

These variables can be of any data type; however, you must format all of them into a string before writing them to file. This conversion is housed in this example in a subVI. In this subVI, the analog values of dbl data type are handled separately from the digital logic values. In the end, a single string containing each I/O value, separated by a "tab" or "comma," must be constructed before being written to the ASCII file. The "tab" or "comma" should be placed in between each value to signify the separation of each value, and an "End of Line" character should be used to signify the end of the row of data. This allows spreadsheet editors to extract the row and column data from a file.

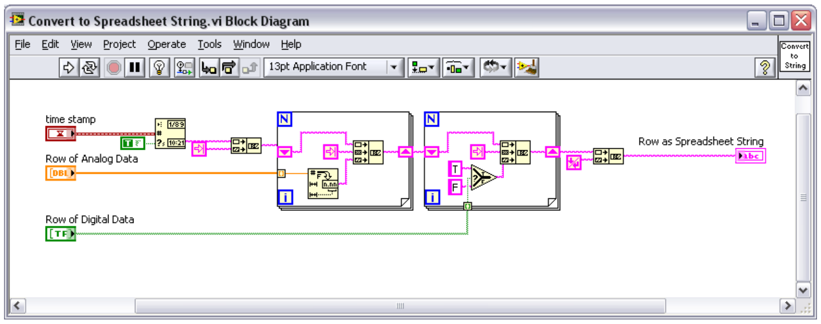

Convert to Spreadsheet String.vi Block Diagram
File Edit View Project Operate Tools Window Help
13pt Application Font
time stamp
Row of Analog Data
Row of Digital Data
Row as Spreadsheet String
N
i
T
F

Figure 3.33. Formatting the Data into an ASCII String

Dynamically Creating New Files

In the final section, code is added in to dynamically close the existing file and create a new file at user defined intervals. This allows the user to then FTP the old data off of the CompactRIO controller to a Host PC for analysis, storage, and/or writing to a SQL server. If the user writes to a single file continuously, it is impossible to pull data off the controller before it finishes, and it leads to storage space limitations. By having a piece of code that automatically closes the existing file and creates a new one at some user-defined rate, it allows the user to access the previously written files before the application completes. To create this functionality, an "Elapsed Time" Express VI was used to monitor how much time had elapsed. When a certain amount of time had elapsed (one hour in this example), it closed the existing VI and created a new one using the same functions as the "Initializing the File" section of this document.

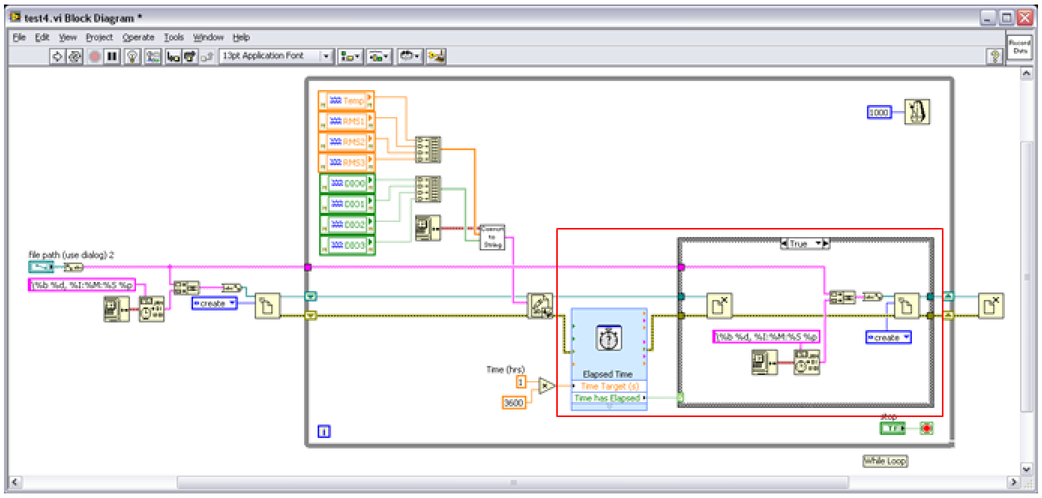


Figure 3.34. Dynamically Creating New Files

Integrating Data-Logging Code with Control Architecture

Now that you have taken an in-depth look at the data-logging architecture, now consider how to integrate the architecture with the control code. Figure 3.35 shows the control code developed in a previous section.

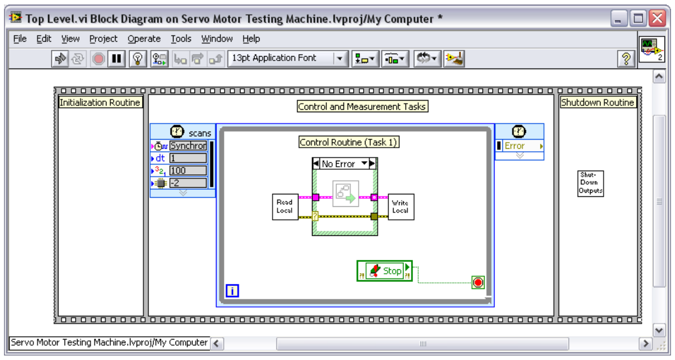

Top Level.vi Block Diagram on Servo Motor Testing Machine.lvproj/My Computer *
File Edit View Project Operate Tools Window Help
13pt Application Font
Initialization Routine
Control and Measurement Tasks
Shutdown Routine
scans
Synchrod
dt
100
Control Routine (Task 1)
No Error
Read Local
Write Local
Error
Stop
Shut-Down Outputs
Servo Motor Testing Machine.lvproj/My Computer

Figure 3.35. Control Architecture without Data Logging

To add data logging to this control architecture, you simply add the data logging code shown above in parallel to the control task.

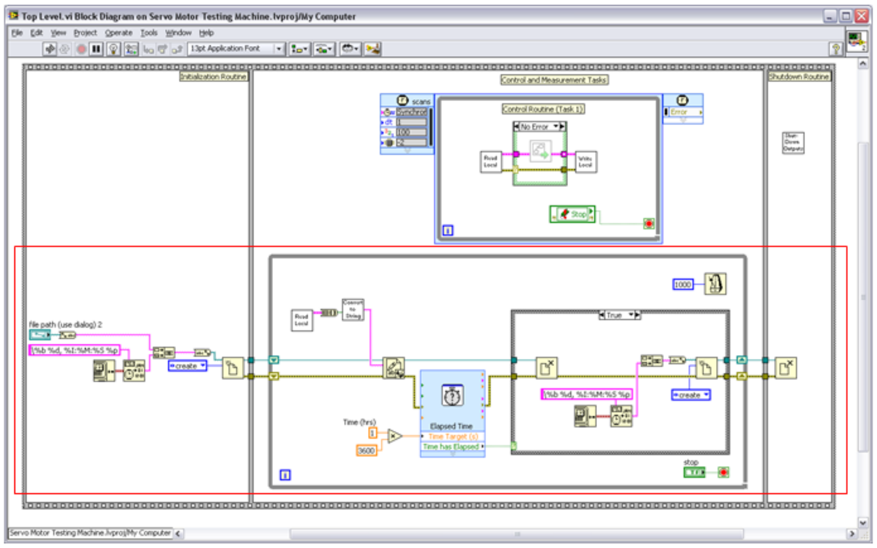
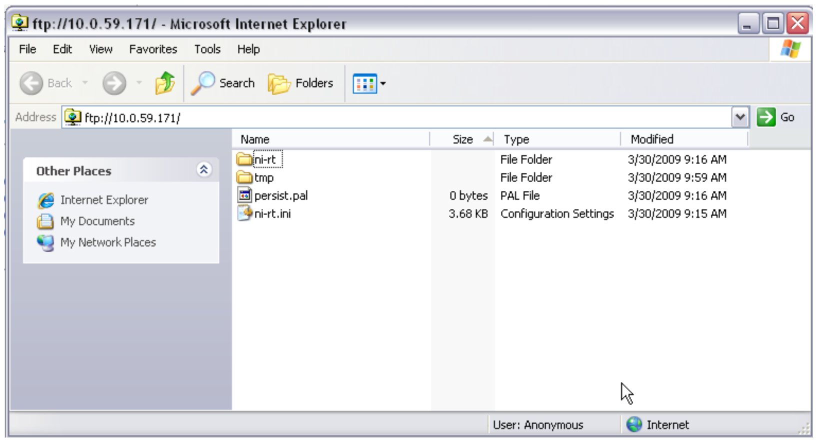


Figure 3.36. Control Architecture with Data Logging

Retrieving the Logged Data

CompactRIO controllers run an FTP server, so you can view files on the controller from any FTP client including Web browsers.

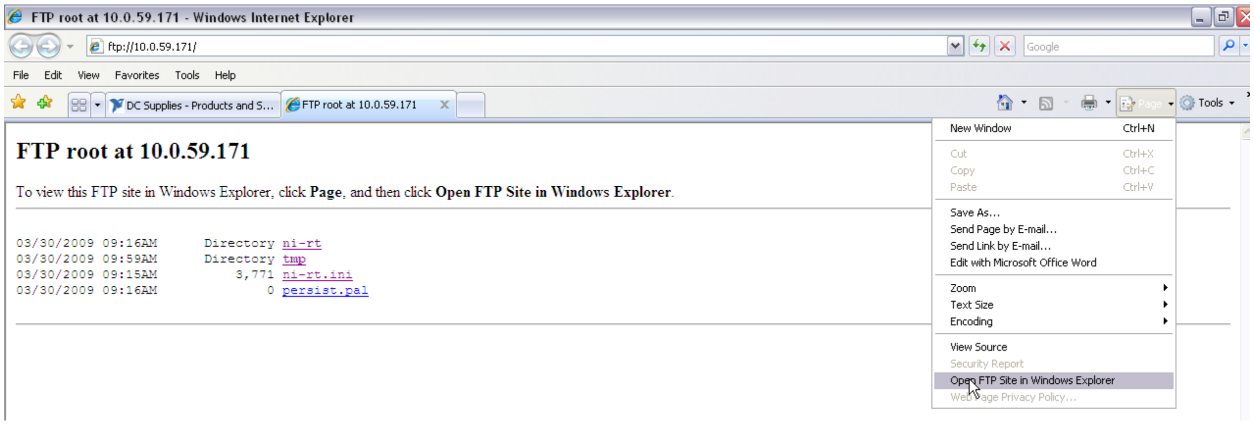

ftp://10.0.59.171/ - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Folders
Address ftp://10.0.59.171/ Go
Other Places
Internet Explorer
My Documents
My Network Places
User: Anonymous
Internet

Name	Size	Type	Modified
ni-rt		File Folder	3/30/2009 9:16 AM
tmp		File Folder	3/30/2009 9:59 AM
persist.pal	0 bytes	PAL File	3/30/2009 9:16 AM
ni-rt.ini	3.68 KB	Configuration Settings	3/30/2009 9:15 AM

Figure 3.37. Using FTP, you can directly retrieve files from the CompactRIO controller.

Web Browsers

Most Web browsers cache the information from the Web pages that you visit, so that they do not have to download the information each time you visit the page. Instead, they load the pages from the cache on your hard drive. This causes you not to see the new files on your CompactRIO controller. The solution is to Reload or Refresh the page in your Web browser or change the cache settings in your Web browser, so that you force the browser to check for new content each time.


FTP root at 10.0.59.171 - Windows Internet Explorer
ftp://10.0.59.171/
Google
File Edit View Favorites Tools Help
DC Supplies - Products and S...
FTP root at 10.0.59.171
Page
Tools
New Window Ctrl+N
Cut Ctrl+X
Copy Ctrl+C
Paste Ctrl+V
Save As...
Send Page by E-mail...
Send Link by E-mail...
Edit with Microsoft Office Word
Zoom
Text Size
Encoding
View Source
Security Report
Open FTP Site in Windows Explorer
Web Page Privacy Policy...

FTP root at 10.0.59.171

To view this FTP site in Windows Explorer, click **Page**, and then click **Open FTP Site in Windows Explorer**.

```
03/30/2009 09:16AM      Directory ni-rt
03/30/2009 09:59AM      Directory tmp
03/30/2009 09:15AM          3,771 ni-rt.ini
03/30/2009 09:16AM              0 persist.pal
```

Figure 3.38. Connecting to a CompactRIO Controller from Windows Explorer

Programmatically

To automatically transfer files from the CompactRIO controller to a Windows PC, you also can use the FTP VIs included in the LabVIEW Internet Toolkit. Run a program on the Windows PC that periodically pulls new files from the controller. Below is a simple example of how to move a file from a real-time target to a host computer. For real-time targets, the default user and password are anonymous and an empty string, respectively. The host is the IP address of the real-time target.

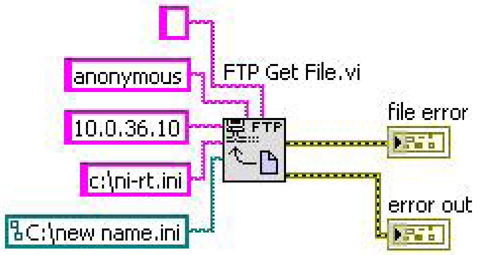


Figure 3.39. The LabVIEW Internet Toolkit provides FTP VIs that you can use to create a Windows application to programmatically pull files off the CompactRIO system.

Errors and Faults

In LabVIEW, the error cluster is an essential tool to track and monitor errors. A well-established and well-documented tool for tracking problems with software or hardware, the error cluster is ideal for systems that always have an operator present. For sophisticated control applications that frequently run headless, you can improve the error cluster with techniques for sharing errors between parallel tasks, a management loop to take action based on the error, and the ability to monitor errors remotely. In LabVIEW 8.6, CompactRIO systems running the NI Scan Engine include a new feature called the NI Fault Engine.

The Fault Engine

Fault Engine

The fault engine is a memory space that records and shares error conditions on the CompactRIO system. By default, it records and reports any errors from the I/O scan. You can enter new faults into this engine and monitor any faults programmatically. Additionally, you can monitor and clear any faults from the NI Distributed System Manager.

To use the fault engine, you need to:

- Record any errors
- Determine appropriate logic for any errors
- Create a fault-handling loop to monitor and respond to any errors

Recording Errors

To enter new errors into the fault engine, use the Set Faults.vi. This takes an error cluster as an input. If an error occurs, the Set Faults.vi automatically logs this as a fault. You can also enter your own user-defined faults by creating and wiring a user-defined error cluster. In the NI Distributed System Manager, the error text appears if you create a custom error file and save it on each machine running the system manager.

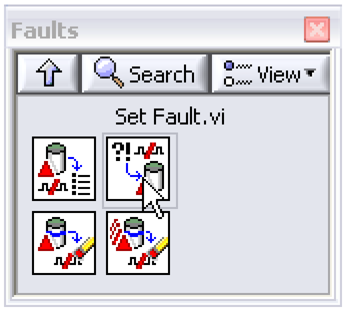


Figure 3.40. The Faults Palette

Error Logic

For each application, you need to determine the appropriate action to take when errors occur. Errors are always an indication that something is wrong with the process, but the action you take may vary based on the severity of the problem. For instance, if the error indicates that you have a broken thermocouple, you may choose to notify maintenance but continue the control process by using other values to estimate the temperature. But if you get an error showing that the motion control is not following the profile, you may need to immediately shut down the machine until it can be serviced. You can read the error code for any faults that are set and determine which action to take. In this simple example, you need to shut down the machine if the same error code is thrown 10 times.

Fault-Handling Loop

You also need to create a loop to monitor any faults and run your fault logic. This loop should run at a high priority so that it always operates and is not "starved" by low-priority tasks.

Example Code with Fault-Handling Loop

LabVIEW Example Code
is provided for this section

In this simple example, you monitor faults from the main task. Notice that you create a user-defined fault if the timed loop finishes late (indicating you are not completing the logic in time). The fault management routing reads all the faults and runs your logic. It can decide to shut down the controller by sending a command. In the initialization routine, you also add code to clear all the faults and log any initialization faults.

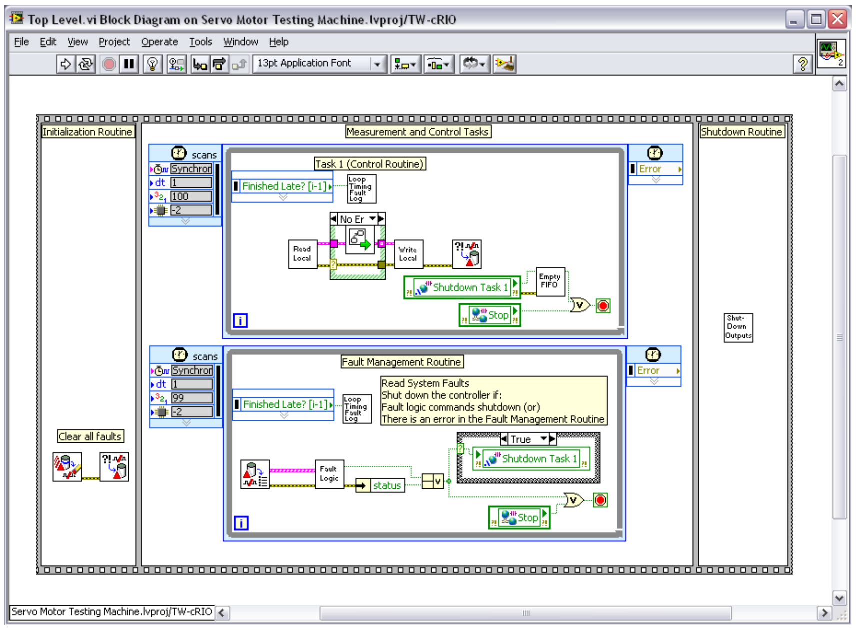


Figure 3.41. A Complete Application with Fault Handling

Real-Time Watchdog

To provide an additional failsafe against programming mistakes such as memory leaks or priority inversion, all NI real-time controllers also feature a built-in hardware timer known as a watchdog timer. You can use a watchdog timer to initiate a recovery procedure if the software becomes unresponsive.

Understanding the Hardware-Software Interface

A watchdog timer is a hardware counter that interfaces with the embedded software application to detect and recover from software failures. During normal operation, the software application initiates the hardware timer to count down from a specific number at a known increment and defines the action to take if the timer reaches zero. After the application starts the watchdog timer, it periodically resets the timer to ensure that the timer never reaches zero.

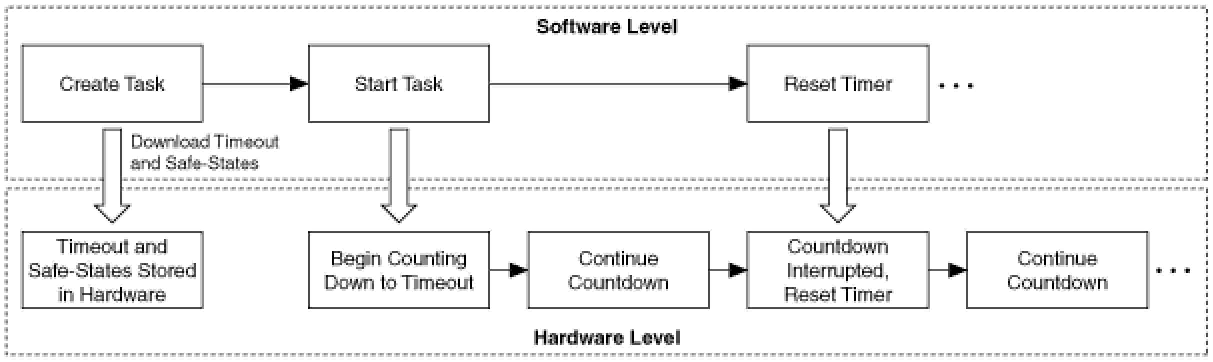

Software Level
Create Task
Start Task
Reset Timer
Download Timeout and Safe-States
Timeout and Safe-States Stored in Hardware
Begin Counting Down to Timeout
Continue Countdown
Countdown Interrupted, Reset Timer
Continue Countdown
Hardware Level

Figure 3.42. A watchdog timer is a hardware feature that can take action if the software becomes unresponsive.

If a software failure prevents the application from resetting the timer, the timeout eventually expires because the hardware counter is independent of the software and thus continues to count down until it reaches zero. When the watchdog timer expires, the hardware triggers the recovery procedure.

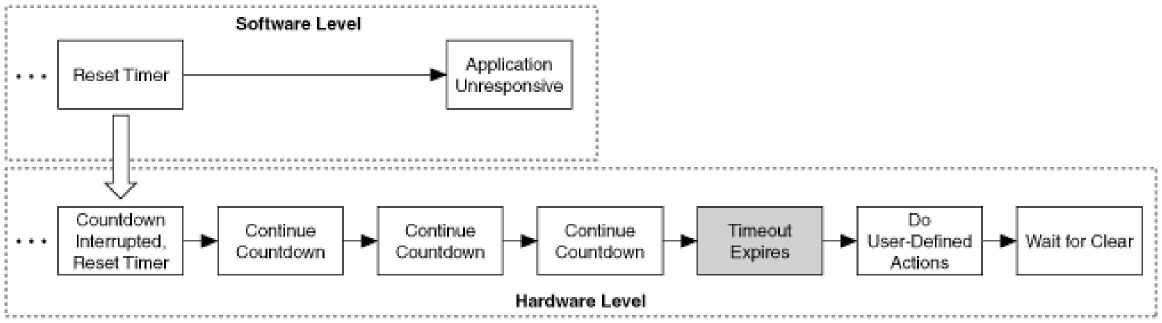


Figure 3.43. If software does not reset the hardware trigger, the hardware can take actions such as rebooting the controller or firing an interrupt.

You can access the built-in watchdog timer hardware using the Real-Time Watchdog VIs. Use the Watchdog Configure VI to set a timeout for the watchdog timer, and to specify the action(s) to perform if the timeout expires. Use the Watchdog Whack VI to periodically reset the counter before the timeout expires.

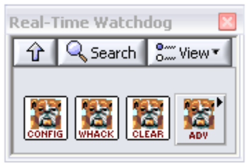


Figure 3.44. The Real-Time Watchdog VI Palette

Choosing an Appropriate Timeout Setting

The appropriate range of timeout values depends on the specific performance characteristics and up-time requirements of the embedded application. You must set the timeout long enough so that it does not expire due to acceptable levels of system jitter. However, you must set the timeout short enough so that the system can recover from failure quickly enough to meet system up-time requirements.

Appropriate Timeout Settings

The appropriate range of timeout values depends on the specific performance characteristics and up-time requirements of the embedded application. You must set the timeout long enough so that it does not expire due to acceptable levels of system jitter. However, you must set the timeout short enough so that the system can recover from failure quickly enough to meet system up-time requirements.

Example Code with Hardware Watchdog Enabled

LabVIEW Example Code
is provided for this section

You modified the code from the fault handler to also enable a hardware watchdog. In the initialization routine configure the hardware watchdog. In this example, you set the timer to 1 second and set the action to reboot the controller. In the fault loop, now "whack the dog" or reset the timer. The timer must be reset every 1 second or the controller reboots. Finally, in the shutdown routine, clear the watchdog once the shutdown outputs VI has completed successfully.

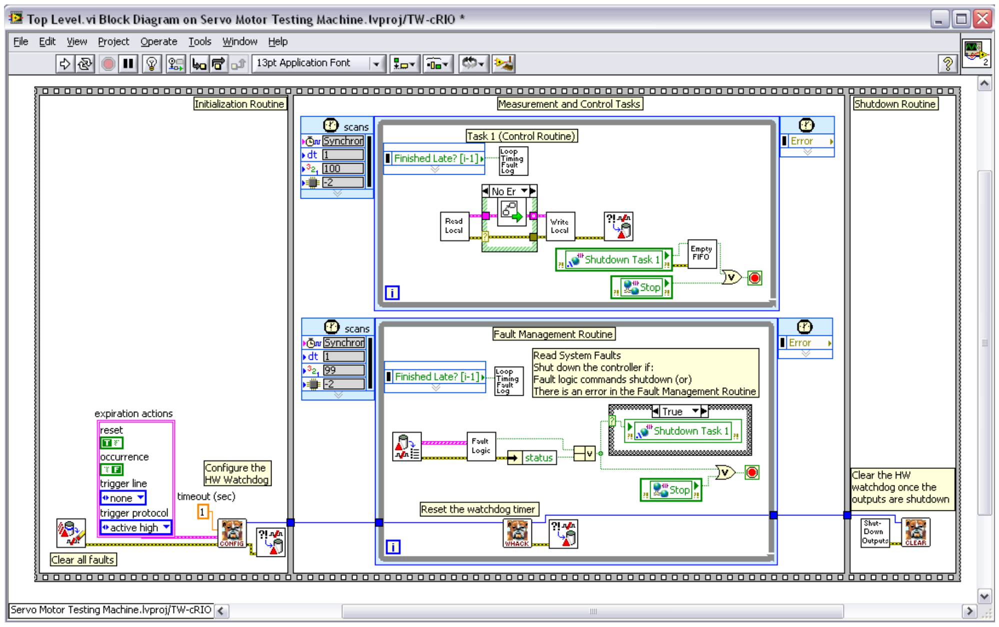


Figure 3.45. A Simple Application Implementing a Hardware Watchdog to Provide Increased Reliability

CHAPTER 4
Communications from the CompactRIO System

Communications Overview

Machine control applications often involve a variety of systems that must exchange information with one another. A central controller may be reading data from peripheral instruments, receiving operator input from an HMI, or sending test results to an enterprise management database at a corporate data center. CompactRIO hardware offers several options for communication including the following:

- Communications directly from the real-time controller
  - Ethernet (TCP, UDP, shared variables, Modbus/TCP, EtherNet/IP, EtherCAT)
  - Serial (RS232, Modbus, Custom Protocols)
- Communications using a plug-in module
  - Serial (RS232/RS422/RS485)
  - CAN
  - PROFIBUS
  - Raw I/O (for custom protocols)

When choosing a communication scheme, consider the nature and type of the information that you are exchanging. In general, machine control applications can contain the following types of communication:

- Message-based communication
- Process data
- Streaming/buffered data

Command or Message-Based Communication

Command or message-based communication is something that happens relatively infrequently and is triggered by some specific event. An example is a user pushing a button on an HMI to stop a conveyor belt. A message must be sent from the HMI to the controller, which then stops the conveyor. In message-based communication, it is important to guarantee delivery of the message in a timely manner. In the previous example, when the operator pushes the stop button, he expects an immediate response (a human's perception of "immediate" is on the order of tenths of a second). The command-based architecture you used in a previous section to trigger parallel loops can be modified for message-based communications across a network.

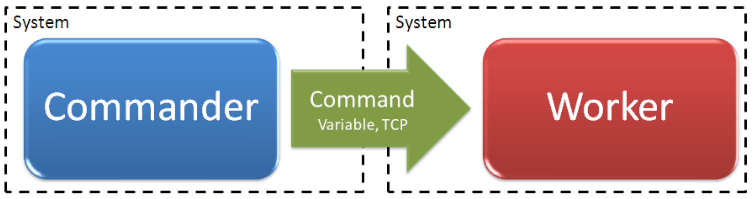


Figure 4.1. You can use multiple technologies to pass commands across Ethernet including shared variables and TCP/IP.

Message-based communication is also appropriate send large sets of fixed size data such as a test result, a thousand measurements from a vibration monitoring application, or an image frame. Most applications where a large amounts of data need to be passed can be broken into periodic tasks where fixed size data sets are passed via a message.

Process Data Communication

Process data communication usually consists of current values being sent periodically between controllers or from controller to HMI. Each data transmission does not need to be guaranteed because the controller or HMI is always interested in the latest value, not in buffered values. An example is data passed to an HMI for displaying the position of a part as it moves along a conveyor belt.

While control applications may require faster data rates, HMI applications normally have relatively slow data update rates. Because only humans see the data, update rates of 1 to 30 Hz are usually sufficient and anything faster is simply wasting bandwidth and processing power. For numeric displays, anything faster than 1 to 2 Hz becomes difficult to see, similar to watching the thousandths digit on a gas pump as you fill your car. For chart or graph displays, 30 Hz provides for smooth updates at the limit of human perception.

Streaming/Buffered Communication

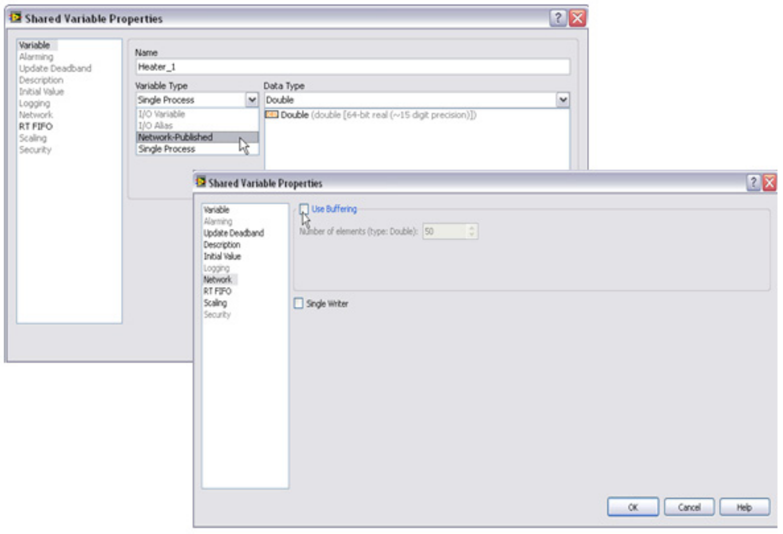
With buffered data communication, information is sent continuously but not necessarily in real time. It is useful for data such as waveforms, for which you need to capture each data point. For example, consider transmitting vibration data from a continuously running MCM application where all the data is stored on a network hard drive for offline analysis. Because the data is not a fixed size, it is difficult to use message-based communications to transmit the data. You need to open a connection and constantly buffer the communication so you do not lose any data. Streaming communication is beyond the basic scope of machine control and is not discussed in detail in this document.

Communication Type	**Characteristics**	**Requirements**
Message-Based	Event-driven, commands	Low latency, guaranteed delivery
Process Data	Single-point, current values	Latest value instead of guaranteed delivery
Streaming/Buffered	Continuous data transfer	High throughput, guaranteed delivery

Table 4.1. Summary of Machine Control Communication Types

Communication Using Network-Published Shared Variables

So far you have examined single-process shared variables used to provide a memory table for deterministically sharing data between loops or sending commands within the one controller. You also can share data across Ethernet by selecting a different type of shared variable, the network-published shared variable. You can use this variable to implement process data communication (sharing data) and message-based communication (sending commands) across Ethernet between controllers, HMIs, and PCs running LabVIEW.



Name: Heater_1
Variable Type: Single Process
Data Type: Double
Double (double [64-bit real (~15 digit precision)])
Use Buffering
Number of elements (type: Double): 50
Single Writer

Figure 4.2. To communicate data across Ethernet, you can use a network-published shared variable.

Network-Published Shared Variable Background

The term *network variable* refers to a software item on the network that can communicate between programs, applications, remote computers, and hardware.

Three important pieces make the network variable work in LabVIEW: network variable nodes, the Shared Variable Engine, and the NI Publish-Subscribe Protocol.

Network Variable Nodes

You can use variable nodes to perform variable reads and writes on the block diagram. Each variable node is considered a reference to a software item on the network (the actual network variable) hosted by the Shared Variable Engine. Figure 4.3 shows a given network variable, its network path, and its respective item in the project tree.

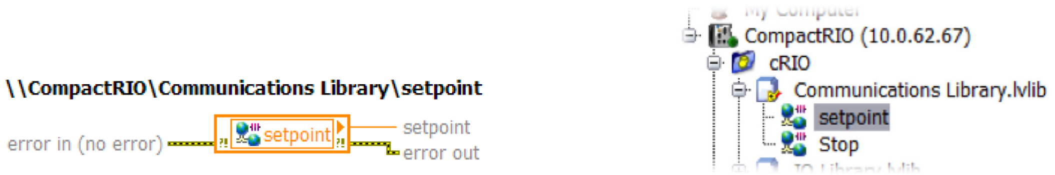


Figure 4.3. Network Variable Node and Its Project Item

Shared Variable Engine

The Shared Variable Engine is a software component that hosts data published over Ethernet. The engine can run on real-time targets or Windows PCs. On Windows, the Shared Variable Engine is a service launched at system startup. On a real-time target, it is an installable startup component that loads when the system boots.

To use network variables, the Shared Variable Engine must be running on at least one of the systems on the network. Any LabVIEW device on the network can read or write to network variables that the Shared Variable Engine publishes. Figure 4.4 shows an example of a distributed system where the Shared Variable Engine runs on a desktop machine and multiple real-time controllers exchange data through a network variable.

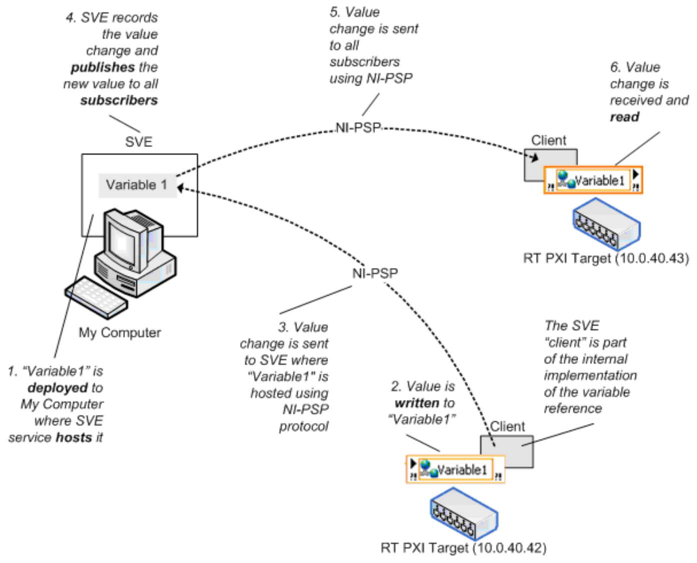

4. SVE records the value change and **publishes** the new value to all **subscribers**
5. Value change is sent to all subscribers using NI-PSP
6. Value change is received and **read**
SVE
Variable 1
NI-PSP
Client
Variable1
RT PXI Target (10.0.40.43)
NI-PSP
My Computer
1. "Variable1" is **deployed** to My Computer where SVE service **hosts** it
3. Value change is sent to SVE where "Variable1" is hosted using NI-PSP protocol
2. Value is **written** to "Variable1"
The SVE "client" is part of the internal implementation of the variable reference
Client
Variable1
RT PXI Target (10.0.40.42)

Figure 4.4. Distributed System Using a Network-Published Shared Variable to Communicate

Publish-Subscribe Protocol (PSP)

The Shared Variable Engine uses the NI Publish-Subscribe Protocol (NI-PSP) to communicate data. The NI-PSP is a networking protocol built using TCP that is optimized for reliable communication of many data elements across an Ethernet network. To minimize Ethernet bandwidth usage each client subscribes to individual data elements. The protocol then implements an event driven communication mechanism where data is only transmitted to subscribed clients on data change. The protocol also combines multiple messages into one packet to minimize Ethernet overhead. In addition, it provides a heartbeat to detect lost connections and features automatic reconnection if devices are added to the network. The NI-PSP networking protocol uses *psp* URLs to transmit data across the network.

Network-Published Shared Variable Features

Buffering

You can use buffering to implement message based communications. Buffering should not be used for Process Data communications.

When you configure a network buffer for a network-published shared variable, you are actually configuring the size of two different buffers, a server side and a client side. The server-side buffer, depicted as the buffer inside the box labeled SVE in Figure 4.5 is automatically created and configured to be the same size as the client-side buffer. The client-side buffer (depicted on the right-hand side of Figure 4.5) is the buffer responsible for maintaining the queue of previous values. Each reader of a network-published shared variable gets its own buffer so readers do not interact with each other.

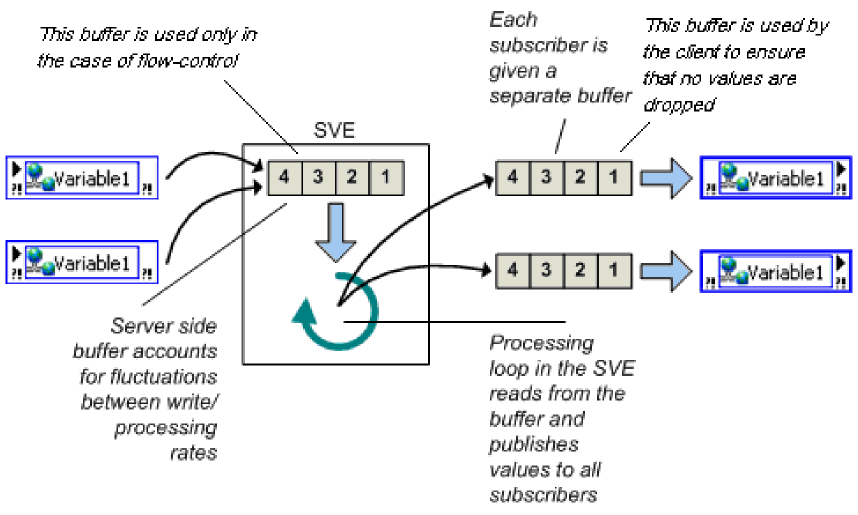

This buffer is used only in the case of flow-control
SVE
Each subscriber is given a separate buffer
This buffer is used by the client to ensure that no values are dropped
Variable1
Variable1
Variable1
Variable1
4 3 2 1
Server side buffer accounts for fluctuations between write/ processing rates
Processing loop in the SVE reads from the buffer and publishes values to all subscribers

Figure 4.5. Buffered Network Variables

Because buffering allocates a buffer for every subscriber, to avoid unnecessary memory usage, use buffering only when necessary and limit the size of the buffers.

Determinism

Network-published shared variables are extremely flexible and configurable. You can create a variable like this that features a real-time FIFO to combine the capabilities of network-published and single-process shared variables. When you do this, LabVIEW automatically runs a background loop to copy the network data into a real-time FIFO.

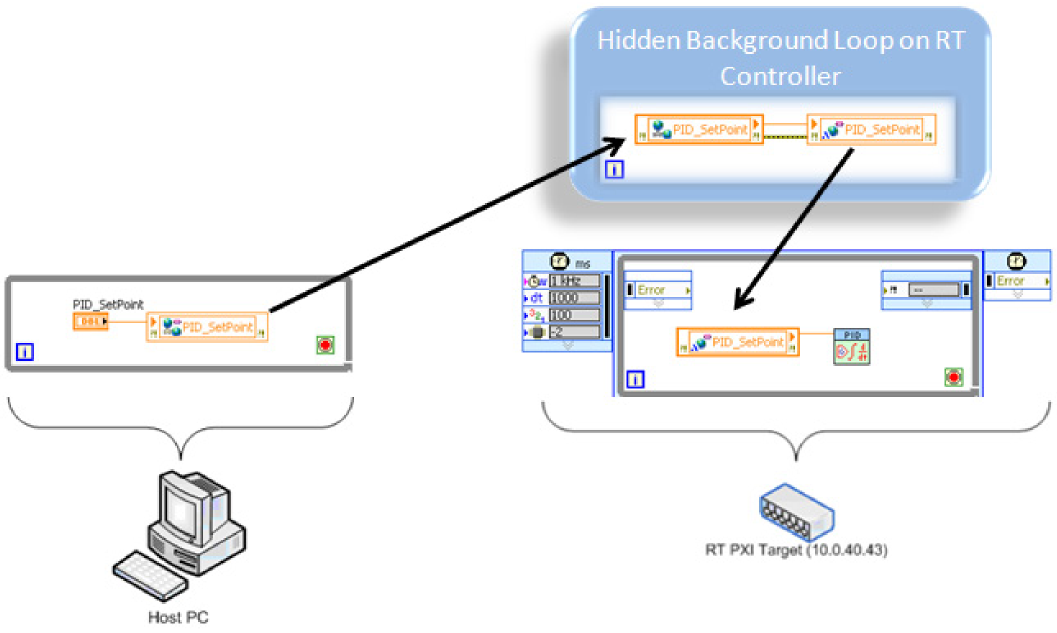


Figure 4.6. When you enable the real-time FIFO for network-published shared variables, a hidden background loop runs on the real-time target to copy the network value into a real-time FIFO.

- While this feature can simplify your program, it has some limitatSome capabilities of network-published variables are not available when you enable the real-time FIFO
- Error management is more difficult because network errors are propagated to the individual nodes throughout the program
- Future modification of the program to use different network communications is more difficult

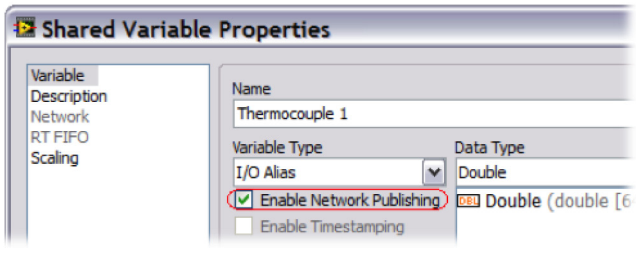
This is an advanced feature and, for the sake of clarity, this document does not examine this capability. Instead you only use network-published shared variables for network communications and continue to use single-process shared variables to provide a memory table with deterministic communications between loops on a single controller. This section instructs you to explicitly create a loop to copy the data from the network-published shared variable into a single-process shared variable.

Lifetime
All shared variables are part of a project library. By default, the Shared Variable Engine deploys and publishes that entire library of shared variables as soon as you run a VI that references any of the contained variables. Stopping a VI does not remove the variable from the network. Additionally, if you reboot a machine that hosts the shared variable, the variable is available on the network again as soon as the machine finishes booting. If you need to remove the shared variable from the network, you must explicitly undeploy the variable or library from the **Project Explorer** window or the NI Distributed System Manager.

SCADA Features
The LabVIEW Datalogging and Supervisory Control (DSC) Module provides a suite of additional SCADA functionality on top of the network-published shared variables including the following:

- Historical logging to NI Citadel database
- Alarms and alarm logging
- Scaling
- User-based security
- Creation of custom I/O servers

Network-Published I/O Variables and Aliases
By default, I/O variables and I/O aliases are published to the network for remote I/O monitoring using the PSP protocol. They are published by a normal priority thread associated with the NI Scan Engine at a rate you specify under the properties of the controller. You can configure whether I/O variables publish their states by accessing the **Shared Variable Properties** dialog.



Name: Thermocouple 1
Variable Type: I/O Alias
Data Type: Double
[✓] Enable Network Publishing
[ ] Enable Timestamping
Double (double [6

Figure 4.7. Enabling Network Publishing for an I/O Variable

Published I/O variables are optimized for I/O monitoring and do not support all the features of network-published shared variables and are not supported by all LabVIEW devices. For maximum flexibility when sharing data between LabVIEW applications, you should use network-published shared variables.

Hosting and Monitoring Network-Published Shared Variables

Hosting

To use network-published shared variables, a Shared Variable Engine must be running on at least one of the nodes in the distributed system. Any node on the network can read or write to shared variables that the Shared Variable Engine publishes. All nodes can reference a variable without having the Shared Variable Engine installed, and, in the case of real-time controllers, a small installable variable client component is required to reference variables hosted on other systems.

You also might have multiple systems running the Shared Variable Engine simultaneously, allowing applications to deploy shared variables to different locations as required.

You must consider the following factors when deciding from which computing device(s) to deploy and host network-published shared variables in a distributed system.

Shared Variable Engine Compatibility

Some of the computing devices in your distributed system may not support hosting the Shared Variable Engine including Macintosh, Linux, and Windows CE systems. Refer to the "NI-PSP Networking Technology" section of the LabVIEW Help for a list of compatible systems and platforms.

Available Resources

Hosting a large number of network variables can take considerable resources away from the system, so for large distributed applications, NI recommends dedicating a system for running the Shared Variable Engine.

Required Features

If the application requires DSC functionality, then those variables must be hosted on a Windows machine running the Shared Variable Engine.

Availability

Some of the hosted process variables may be critical to the functioning of the distributed application, so they benefit from running on a reliable embedded OS such as LabVIEW Real-Time to increase the overall reliability of the system.

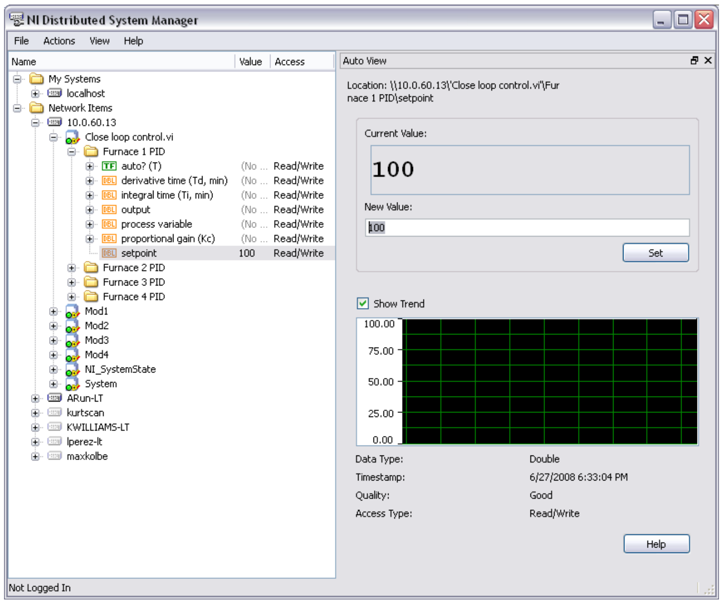
Dynamic Access Variables

A network variable can also be referenced by a path name allowing for you build programs that dynamically select what variable to read or write. The path name is similar to a Windows network share name, such as `\\machine\myprocess\item`. In this example, *machine* is the computer name, IP address, or fully qualified domain name of the server machine that is hosting the variable; *myprocess* contains network variable folders or variables and is referred to as a network variable process; and *item* is the name of the network variable. Below are additional examples of network variable references:

```
\\localhost\my_process\my_variable
\\test_machine\my_process\my_folder\my_variable
\\192.168.1.100\my_process\my_variable
```

Monitoring Variables

The NI Distributed System Manager offers a central location for monitoring systems on the network and managing published data. From the system manager, you can access network-published shared variables and I/O variables without needing the LabVIEW development environment.



Name	Value	Access
My Systems		
localhost		
Network Items		
10.0.60.13		
Close loop control.vi		
Furnace 1 PID		
auto? (T)	(No ...	Read/Write
derivative time (Td, min)	(No ...	Read/Write
integral time (Ti, min)	(No ...	Read/Write
output	(No ...	Read/Write
process variable	(No ...	Read/Write
proportional gain (Kc)	(No ...	Read/Write
setpoint	100	Read/Write
Furnace 2 PID		
Furnace 3 PID		
Furnace 4 PID		
Mod1		
Mod2		
Mod3		
Mod4		
NI_SystemState		
System		
ARun-LT		
kurtscan		
KWILLIAMS-LT		
lperez-lt		
maxkolbe		

Location: \\10.0.60.13\'Close loop control.vi'\Furnace 1 PID\setpoint

Current Value: 100

New Value: 100

Data Type:	Double
Timestamp:	6/27/2008 6:33:04 PM
Quality:	Good
Access Type:	Read/Write

Not Logged In

Figure 4.8. NI Distributed System Manager

With the NI Distributed System Manager, you can write to network-published variables so you can remotely tune and adjust process settings without the explicit need for a dedicated HMI. You can also use the NI Distributed System Manager to monitor and manage controller faults and system resources on real-time targets. From LabVIEW, select **Tools»Distributed System Manager** to launch the system manager.

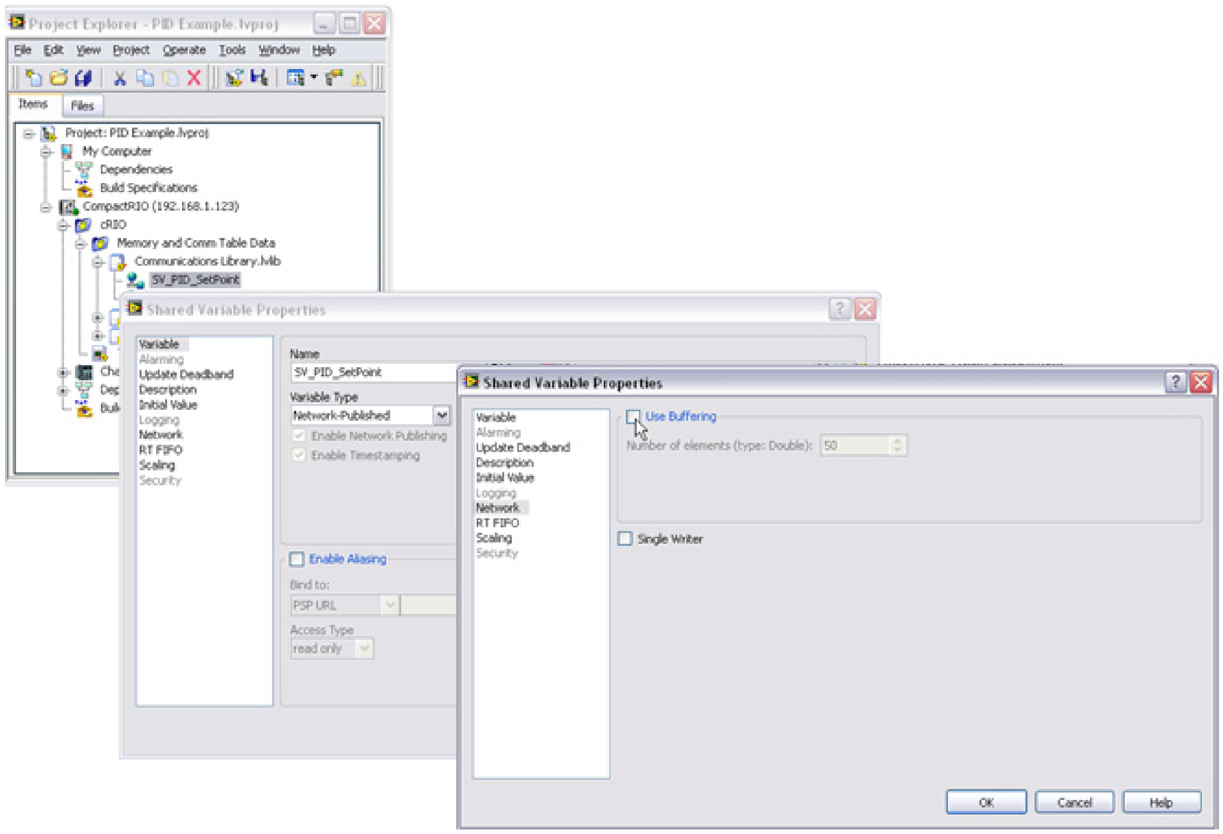
Using Network-Published Variables to Share Process Data

Example

LabVIEW Example Code
is provided for this section

You can easily use network variables to publish process variables. In the PID example from section 2 in this document, you hard-coded the PID setpoint. Now modify the example so you can constantly update the setpoint from another LabVIEW device on the network.

First create a network-published shared variable called SV_PID_SetPoint. Go to the Network tab. By default, the buffering is enabled, so turn **OFF** the buffering. Save the variable in a communications library. When you deploy the code to the CompactRIO system, you automatically deploy this library and the variable is available on the network.


Project Explorer - PID Example.lvproj

- Project: PID Example.lvproj
  - My Computer
    - Dependencies
    - Build Specifications
  - CompactRIO (192.168.1.123)
    - cRIO
      - Memory and Comm Table Data
        - Communications Library.lvlib
          - SV_PID_SetPoint

Shared Variable Properties

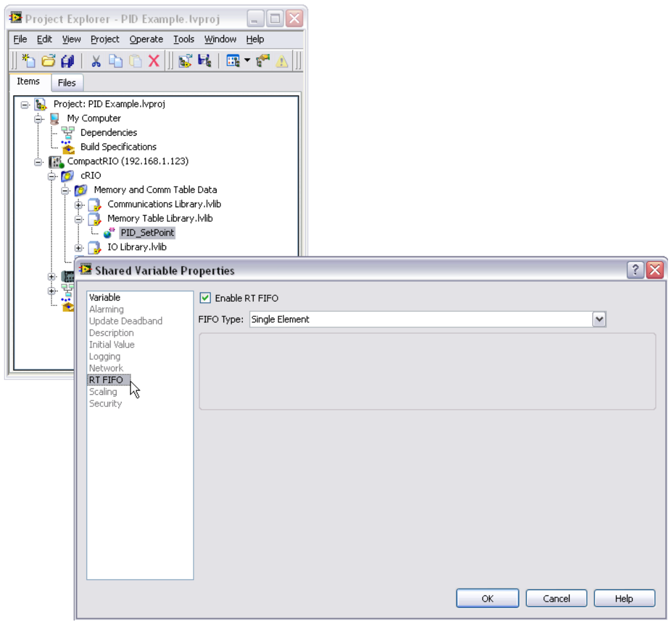
Variable — Name: SV_PID_SetPoint; Variable Type: Network-Published; Enable Network Publishing; Enable Timestamping; Enable Aliasing; Bind to: PSP URL; Access Type: read only

Shared Variable Properties

Network — Use Buffering; Number of elements (type: Double): 50; Single Writer

Figure 4.9. Network-published shared variables allow communication over Ethernet between LabVIEW nodes. Be sure to turn OFF the network buffering.

Because network communications over Ethernet is not deterministic, read SV_PID_SetPoint in a parallel loop to your control loop. Also add an element in your data table to transmit data from your communications loop to your control loop. Create a single-process shared variable with a real-time FIFO called PID_SetPoint.


Project Explorer - PID Example.lvproj

- Project: PID Example.lvproj
  - My Computer
    - Dependencies
    - Build Specifications
  - CompactRIO (192.168.1.123)
    - cRIO
      - Memory and Comm Table Data
        - Communications Library.lvlib
        - Memory Table Library.lvlib
          - PID_SetPoint
        - IO Library.lvlib

Shared Variable Properties

RT FIFO — Enable RT FIFO; FIFO Type: Single Element

Figure 4.10. You can use a single-process shared variable with a real-time FIFO to communicate data deterministically between loops.

Now modify the block diagram to add a normal priority loop that reads SV_PID_SetPoint, check for any errors or warnings, and write the value to PID_SetPoint. Run this loop at 100 Hz.

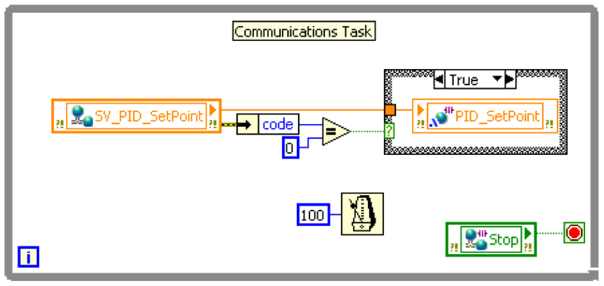


Figure 4.10. A communications task passes data from the network to the local memory table if there is no error or warning.

Finally set the PID block so it reads from the PID_SetPoint and, in the initialization routine, set a default value for the PID_SetPoint.

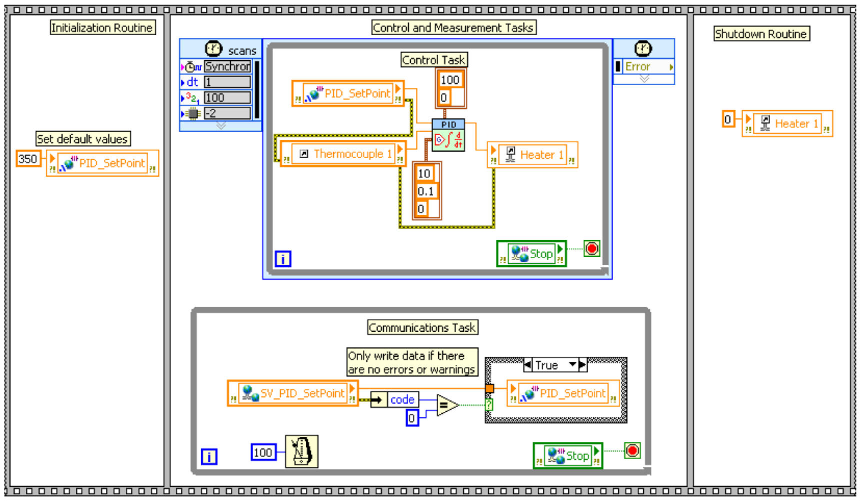


Figure 4.11. A Final Application with Network Communications

You can now continually update the setpoint from any LabVIEW node on the network.

Using Network-Published Variables to Send Commands

An earlier section described how to build a command-based architecture for actions such as triggering tasks. You can use this same concept to generate commands from other devices on the network. Networking technologies allow distributed systems to publish their statuses and coordinate their actions. In the case of the command architecture, you can use those networking technologies to communicate between systems.

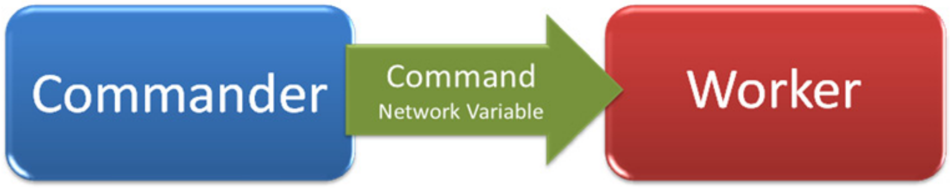


Figure 4.12. Simple Command Architecture Using Network Variables

The publish-subscribe model of network variables also makes it easy to implement multiple commanders.

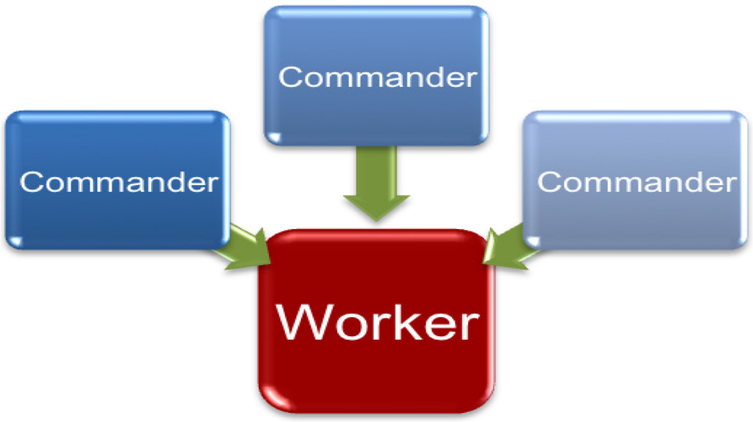


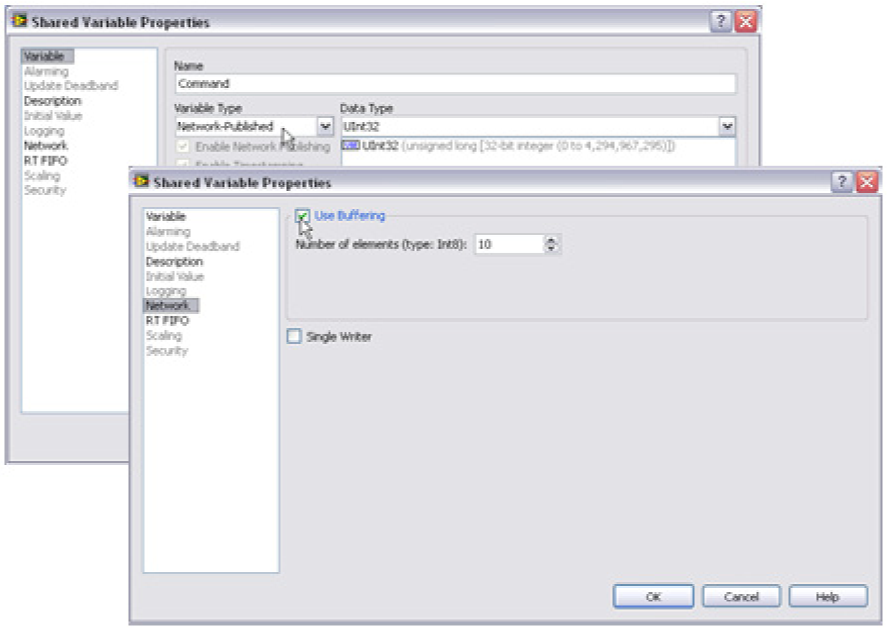
Figure 4.13. Multiple Commanders and One Worker System

The network-published shared variable offers a practical way to create a communication channel for the command messages being sent over the network. To build a scalable application you should design your worker system so that it has a single command parser task that you can use to interpret and redistribute the command as needed. This assures that ongoing critical tasks are not disturbed by the arrival of the command and makes it easy to modify the code to handle additional commands.

To send data, use one network-published shared variable with buffering enabled. Without buffering, consecutive commands overwrite each other, leading to the loss of commands. Buffering is also critical when supporting multiple commanders because it allows for the orderly servicing of commands.

The network-published shared variable offers buffering mechanisms at each of its implementation layers. The Shared Variable Engine buffers values and delivers them to all subscribers, where each instance of a variable node gets its own buffer.

To use one variable for multiple commands, create a variable with the data type of a U32.



Name: Command
Variable Type: Network-Published
Data Type: UInt32
UInt32 (unsigned long [32-bit integer (0 to 4,294,967,295)])
Enable Network Publishing

Shared Variable Properties
[✓] Use Buffering
Number of elements (type: Int8): 10
[ ] Single Writer

Figure 4.14. For network commands, create a network-published shared variable with buffering enabled.

To ensure scalability, you need to be able to add new commands to the system without requiring a significant overhaul of the application or harming the run-time performance. To make it easier to maintain the code, create a type def enum (U32) with an entry for each command. An enumerated type definition, in the form of LabVIEW enum, is an effective way to define commands that can be directly passed to the network variable.

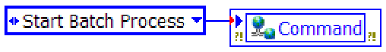


Figure 4.15. Issuing a Command through a Network Variable

The enum type definition provides an ordered list of the available commands, and making it a type definition allows changes to be automatically propagated through the code.

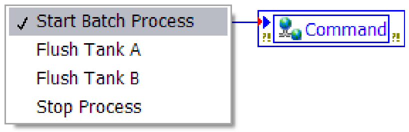


Figure 4.16. Enumerated Command List

Commander Architecture

The commander is any source of commands that the worker must react to. A common commander is a UI event handler that is part of an HMI, taking UI events and translating them into commands for the machine controller. In the case of a networked architecture, the commander takes care of distributing these commands as well as taking local actions, such as temporarily disabling UI items and reporting status.

You can easily implement the commander architecture for events that originate on a user interface by using the standard LabVIEW UI-handling templates. Just translate the UI event into the appropriate command by building the command message and writing it to the network variable.

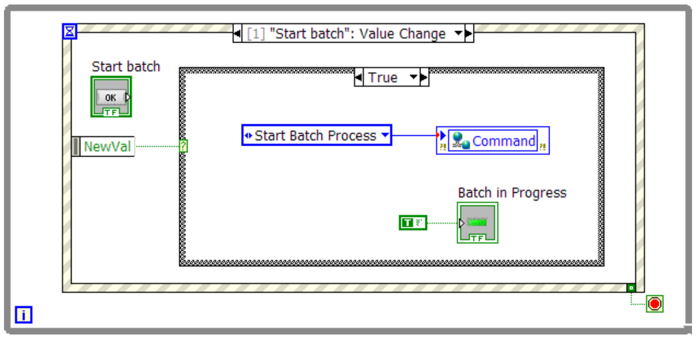


Figure 4.17. Simple UI-Driven Commander VI

Refer to the HMIs section for more information on proper architectures for building an HMI.

Worker Architecture

On the worker system, in a command parser task you can read the network-published shared variable to check for new values. This situation requires the incoming command to be interpreted and applied without affecting the determinism of the control loop. Additionally, the servicing of the commands does not have to occur at the same rate as the servicing of the control algorithm. You therefore use a dedicated task in charge of parsing incoming commands and distributing the necessary information and events across the whole application. Call this the *Command Parser* task.

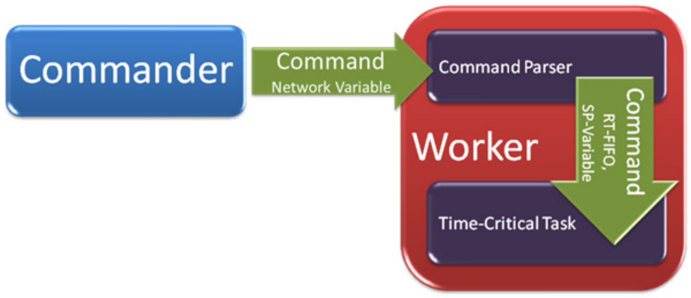


Figure 4.18. Centralized Command Parser Architecture

To implement the *Command Parser* task, simply add a parallel process where new commands are received and handled. A separate command parser loop provides scalability because it:

- Allows the application to limit access to certain process variables based on command origin, parameter value range, and current application state.
- Allows for additional processing and reinterpretation of incoming commands before they are forwarded to the appropriate tasks.
- Allows complex commands to package parameters, forming a coherent set of data to be applied as part of the command.
- Simplifies modifications of the program to handle different network types such as raw TCP.
- Allows an acknowledged command architecture where the command and acknowledgement handling is performed within the same task.

Building the Command Parser

Timeout Reads

To reduce CPU usage you can use a blocking read on the network-published shared variable by specifying a timeout value (in milliseconds). You can enable the blocking behavior on a per-node basis by selecting the **Show Timeout** option from the context menu of the variable node.

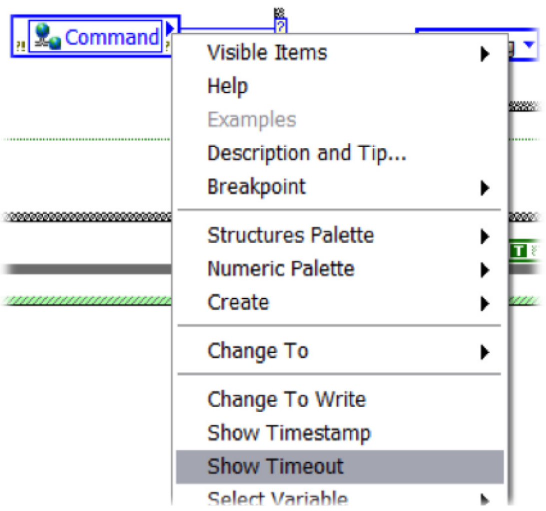


Figure 4.19. Enabling Blocking-Read Functionality for a Network Variable Node

You can then wire a timeout value to the timeout input of the variable to specify how long the variable node should wait for a new value. If no newer values exist in the buffer or arrive before the timeout expires, the variable node returns the last-known value for the variable and sets the timeout Boolean terminal to TRUE. You can therefore handle the timeout case accordingly.

Error Handling

At this point, you need to consider application behavior under error conditions. The network variable node can return warnings or errors to signal the status of its critical underlying layers, which are the Shared Variable Engine, the respective variable-hosting process, and the network state, to name a few. Here are the most important factors to consider when error proofing the receiving side of the command architecture:

A. Upon returning an error, the network variable also returns the default value for its data type. It returns FALSE in the case of a Boolean variable, returns 0 for a numeric type, or returns the first element from the enum list in the case of your enumerated command type.

   - You want the application to override the return value in case of error.

B. Once the connection is established, a buffer for each variable node is created and the current value of the variable is passed in as the first item in the buffer. This presents a problem for your command-based approach because you really cannot tell if this current command value is something the worker should act upon or if it is just there from the last time the application was executed.

   - All readers must flush the variable buffer to guarantee a known initial state by reading the variable until it signals its buffer to be empty by returning a timeout.

These requirements are specific to the command-based architecture when implemented with a network variable; you can fulfill them by wrapping the variable node with a customizable layer that takes care of the above considerations while hiding some of the implementation details of the communication channel. You can use the following code snippet, extracted from the Command Reader Template, to implement your own command-based architecture.

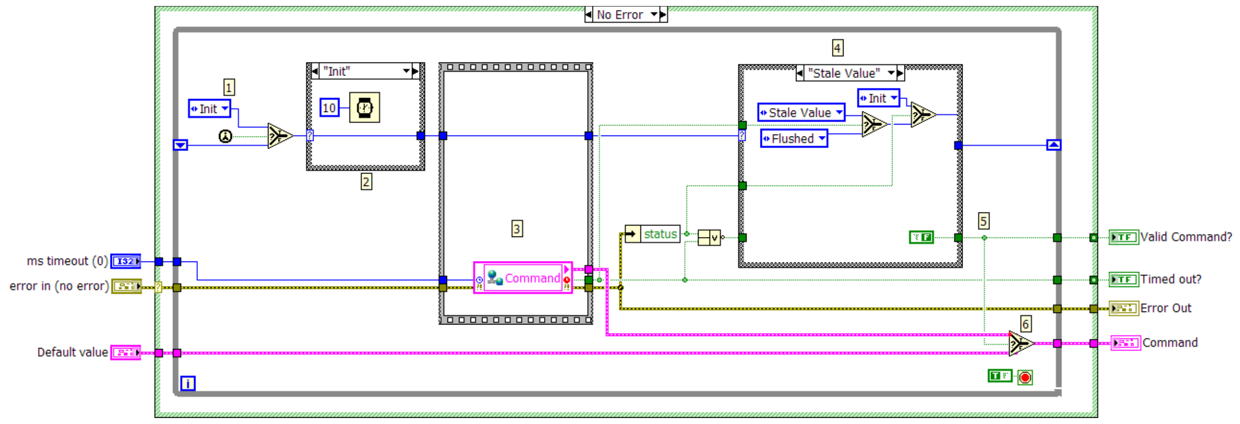


Figure 4.20. Command Reader Template VI

- The code takes care of point A by returning the specified default value when the variable node errors.
- The code takes care of point B, flushing the variable buffer, by reading it until it is empty and then waiting for a command.

Example of Command-Based Architecture Using Network-Published Shared Variables

LabVIEW Example Code
is provided for this section

Now modify the PID with a communications example to implement a network-based command. You want to modify the example so you have a command parser to stop each loop.

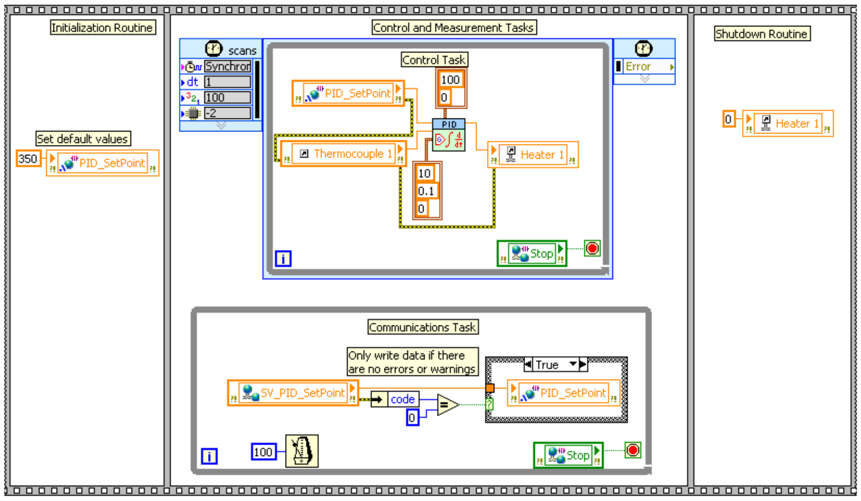


Figure 4.21. In this example, modify this application to add a command parser task.

First create the type def enum for the command. In this very simple example, you have only one command: "Stop."

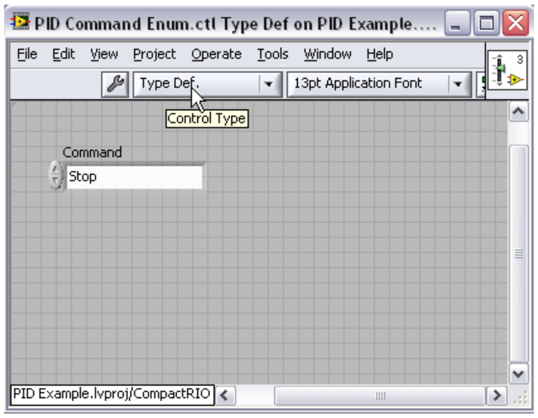
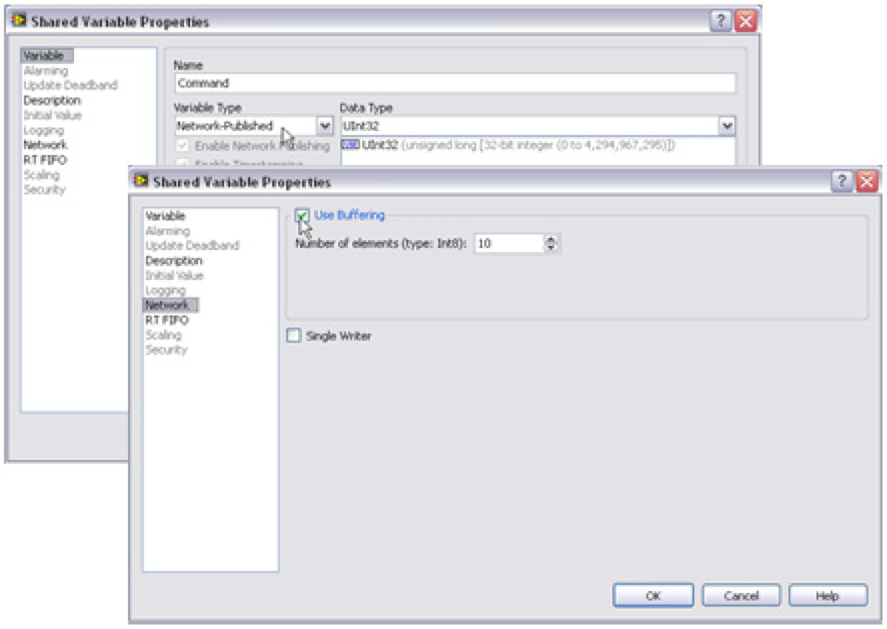


Figure 4.22. A type def enum provides scalability.

Next create a network-published shared variable called "Command." The shared variable buffering is enabled.



Name: Command

Variable Type: Network-Published

Data Type: UInt32 — U32 UInt32 (unsigned long [32-bit integer (0 to 4,294,967,295)])

☑ Enable Network Publishing

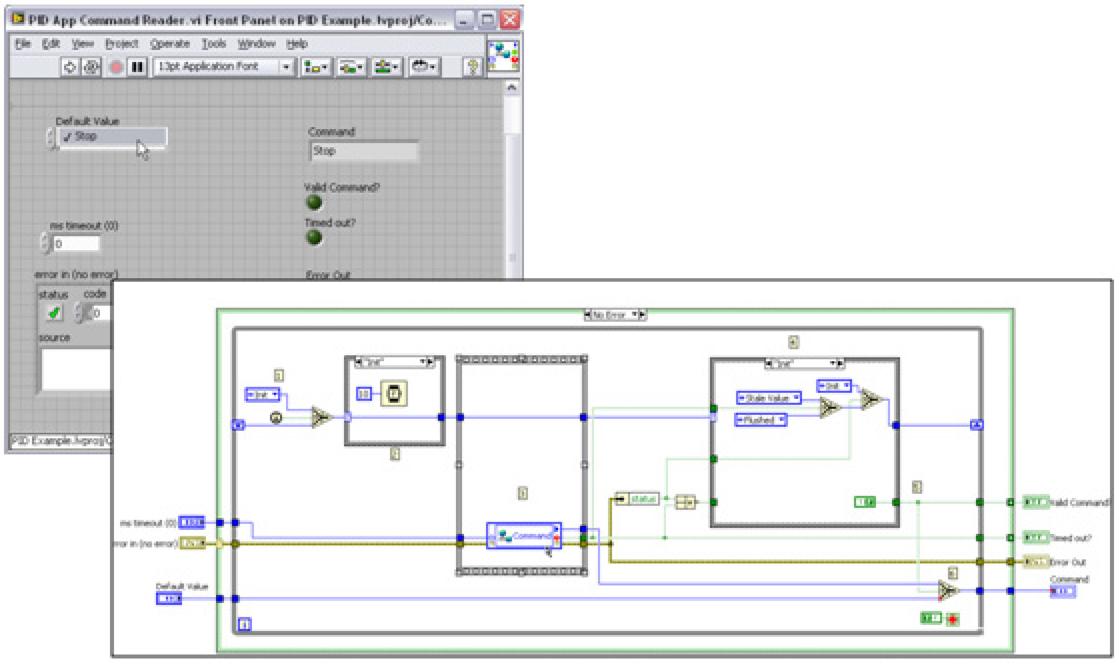
☑ Use Buffering

Number of elements (type: Int8): 10

☐ Single Writer

Figure 4.23. For network commands, create a network-published shared variable with buffering enabled.

Next modify the Command Reader VI by replacing the "Default Value" input and "Command" output with the type def enum. Also replace the shared variable read so it now reads the "Command" shared variable.



Default Value: ✓ Stop

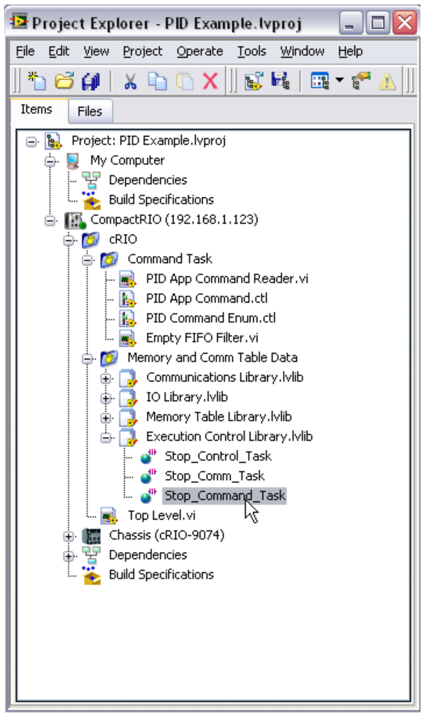
Command: Stop

ms timeout (0): 0

error in (no error): status ✓, code 0, source

Figure 4.24. The Command Reader VI empties the network queue on first call.

Now create single-process shared variables to serve as the commands that shut down each of the loops.



```
Project: PID Example.lvproj
  My Computer
    Dependencies
    Build Specifications
  CompactRIO (192.168.1.123)
    cRIO
      Command Task
        PID App Command Reader.vi
        PID App Command.ctl
        PID Command Enum.ctl
        Empty FIFO Filter.vi
      Memory and Comm Table Data
        Communications Library.lvlib
        IO Library.lvlib
        Memory Table Library.lvlib
        Execution Control Library.lvlib
          Stop_Control_Task
          Stop_Comm_Task
          Stop_Command_Task
      Top Level.vi
    Chassis (cRIO-9074)
    Dependencies
    Build Specifications
```

Figure 4.25. Commands are relayed within the controller using single-process shared variables with multi-element, real-time FIFOs as covered in an earlier section.

Finally add another loop to handle the command parsing. This loop wires to a case structure that executes logic if it was a valid command. If a valid command is received, a second case structure runs the logic for each command. In this simple example, you have only one command: Stop. This command triggers each of the three loops to shut down using the architecture to handle internal commands (covered in section 3).

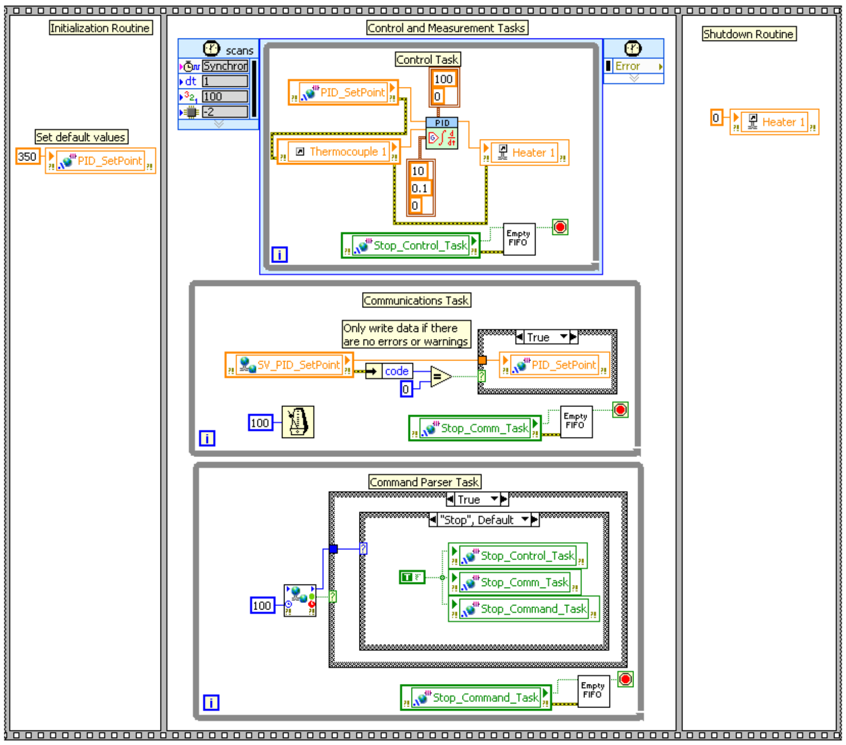


Figure 4.26. Final Block Diagram with a Command Parser Task and Commands to Stop Each Loop

LabVIEW Example Code
is provided for this section

Command-Based Architecture – Incorporating Data and Commands
You can extend the command to include generic parameters so that additional information can be transmitted along with the command. This helps implement more complex commands and ensure data coherency.

You can specify the data type of the network-published shared variable helps you easily extend the contents of the command message to include generic values that you can interpret as the parameters for specific commands.

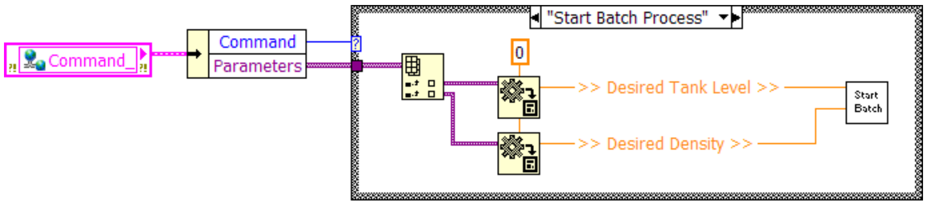


Figure 4.27. Handling a Command with Extended Parameters

This bundling of the command along with its parameters provides parameter coherency to the command and helps avoid race conditions.

To store the information, use a cluster with the command enum and an array for data parameters. The array can be a numeric type, but if you make it a variant type, you can use the flatten and unflatten functions to pass any data type.

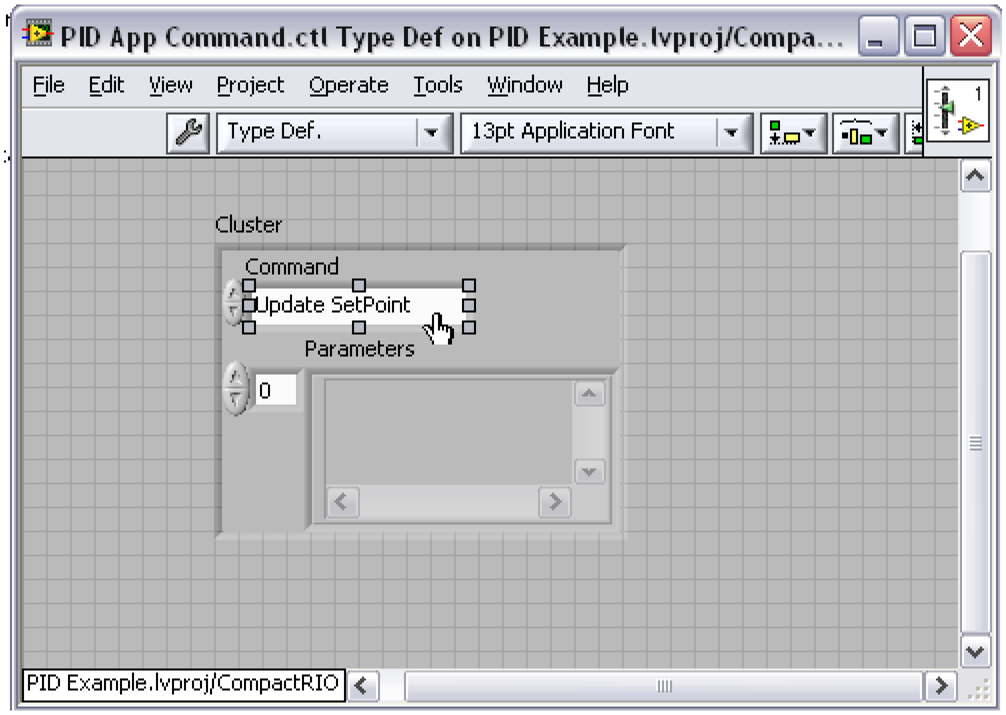
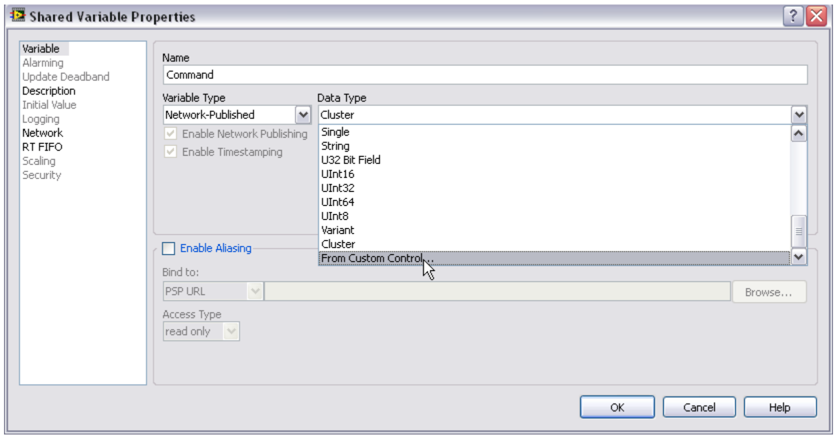

PID App Command.ctl Type Def on PID Example.lvproj/Compa...
File Edit View Project Operate Tools Window Help
Type Def.
13pt Application Font
Cluster
Command
Update SetPoint
Parameters
0
PID Example.lvproj/CompactRIO

Figure 4.28. A cluster can store both the command and associated command data.

You now need to update the network-published shared variable type to make it a cluster. You can do this by changing the type to "From Custom Control" and select the custom control you created.


Shared Variable Properties

Variable
Alarming
Update Deadband
Description
Initial Value
Logging
Network
RT FIFO
Scaling
Security

Name: Command

Variable Type: Network-Published

Data Type: Cluster

- Single
- String
- U32 Bit Field
- UInt16
- UInt32
- UInt64
- UInt8
- Variant
- Cluster
- From Custom Control...

Enable Network Publishing
Enable Timestamping
Enable Aliasing
Bind to: PSP URL
Access Type: read only

OK | Cancel | Help

Figure 4.29. Network-published shared variables can be custom types to match the command cluster.

To remain independent from custom types when deployed, shared variables do not create a permanent link to the custom control used to define their types. This means that if you make changes to the command typedef, for example, to add more command types, you need to relink the shared variable to the custom control for the shared variable to update its type. To eliminate this problem you may choose to make a different cluster for the shared variable where you use a U32 instead of an enum. You still use the type def enum in all applications where you read or write data to the shared variable.

Now modify the command parser task to read the command and wire it to the case structure. Inside each case, unflatten the parameters. In this example, you removed the communications loop because you no longer constantly update the setpoint from the network – instead, you update it based on an input command.

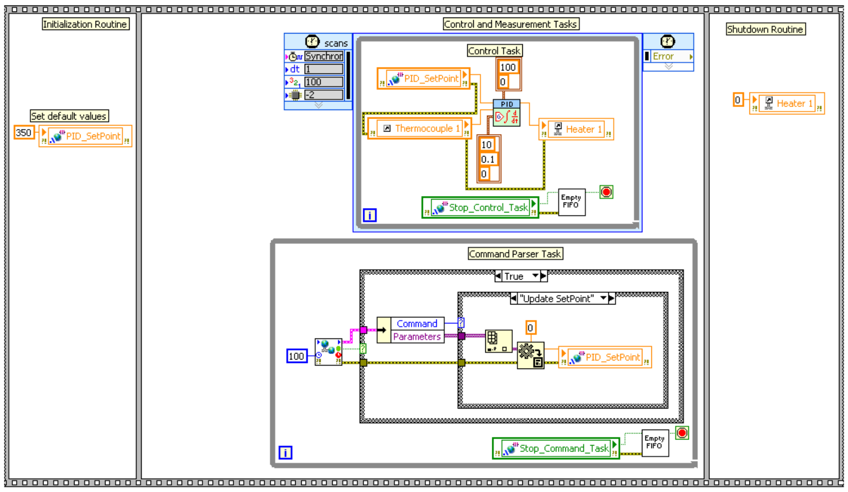


Figure 4.30. Final Block Diagram with a Command Parser Task That Also Has Associated Data

Acknowledged Command-Based Architecture

The PSP protocol provides built-in connection management for all subscribers. The variable node returns information on connection and quality status through errors and warnings on its error output terminal. However, in some applications it is beneficial to implement a more sophisticated communications mechanism to get acknowledgement that a controller not only got the command but was in a state to execute the command. To add acknowledgement, use a second variable for the acknowledgement communication channel.

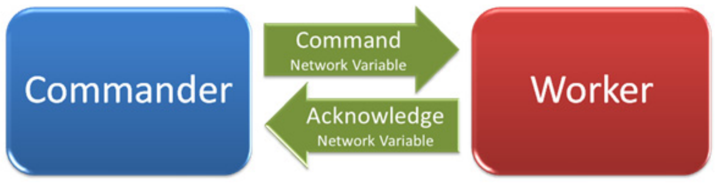


Figure 4.31. Simplified Acknowledged Command Architecture Using Network Variables

Acknowledgement Type

Different applications define message acknowledgement in different ways. While one application may not care if the message is actually delivered (noncritical information), others may want to receive confirmation that the message was indeed received, and yet others may want further confirmation that the message (or command) has been accepted and acted upon. Therefore, use the following acknowledgment type definitions for the context of your command architecture:

- Upon Delivery: Worker loop has read the command from the network. The command is considered received.
- Upon Acceptance: Worker has evaluated the command and determined whether it accepts or refuses to execute it.
- Upon Completion: Worker responds once the command has been executed or informs the commander in case of an error while executing the command.

For any given command, the commander may choose to wait for any of these levels of acknowledgment.

A simple way to implement this is to use a typedef enum to enumerate the different message acknowledgement types.

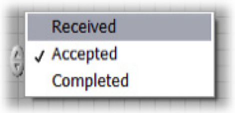


Figure 4.32. Acknowledgment Type Enum

Aside from the acknowledgment type, you must consider the following items when implementing the acknowledged command architecture:

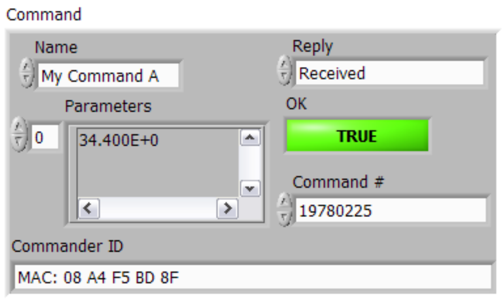
Command ID

When acknowledging multiple instances of the same command, you must attach a series number or sequence identifier to the command so that individual commands can be recognized and acknowledged.

Commander ID

If multiple commanders are present, the command must specify its origin through a unique identifier, so that the acknowledge message can be directed back at the appropriate commander.

Accounting for all of the above, you can devise a format for your command message such as the one represented below.



Command

Name	My Command A
Parameters	0 / 34.400E+0
Reply	Received
OK	TRUE
Command #	19780225
Commander ID	MAC: 08 A4 F5 BD 8F

Figure 4.33. Sample Data Cluster for Acknowledgeable Commands

Using this cluster as the data type for your command network variable allows the worker to receive the information it needs and to uniquely identify the command and its origin. Therefore, the commander is augmented to provide the command sequence number and commander ID along with the relevant command information.

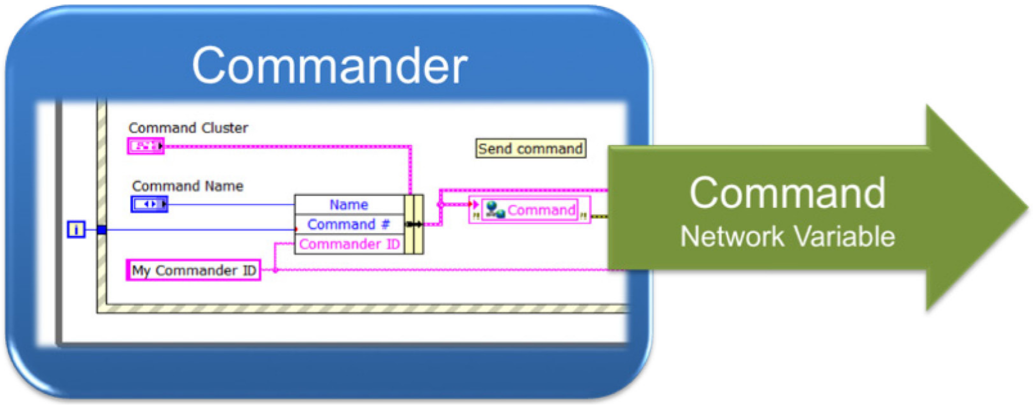


Figure 4.34. Sending Acknowledgeable Commands

The worker receives the command using the same techniques discussed in previous section and proceeds to send the acknowledgement. Conveniently, the cluster type definition you have defined can be reused for the reply message. You implement the acknowledgement by setting the *reply* enum value and the *OK* status to signal that the command was successfully received. The message is then written to a network variable dedicated to command acknowledgement, the *reply* network variable, which is identical to the *command* variable.

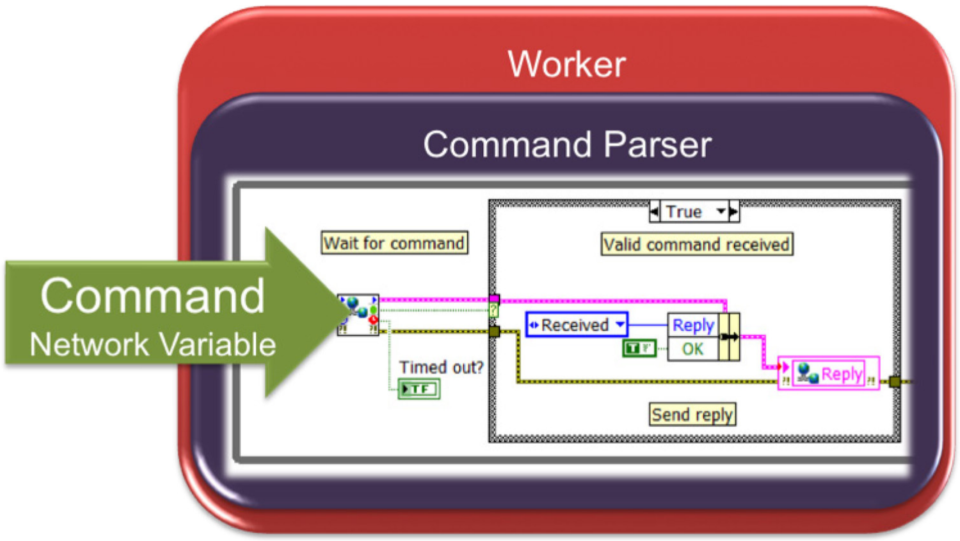


Figure 4.35. Receiving and Acknowledging Commands

Waiting on acknowledgement is a more complicated subject because the different expected acknowledgements may arrive out of order (some commands may get acknowledged at different stages). The accompanying examples provide a template for a VI that can wait on multiple acknowledgements simultaneously. Using the provided template, you can build a simple version of an event-driven commander.

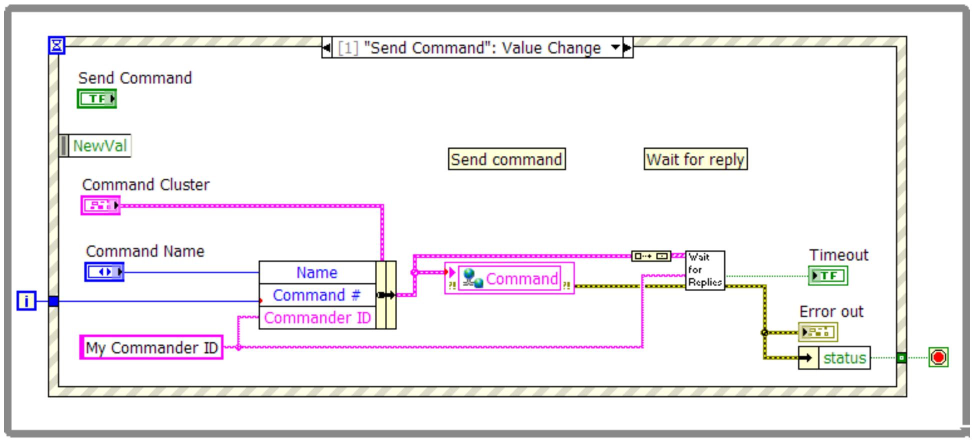


Figure 4.36. Sample Acknowledged Commander VI

Important Considerations When Using Network Variables for Commands

Variable Type Linkage

To remain independent from custom types when deployed, shared variables do not create a permanent link to the custom control used to define their types. This means that if you make changes to the command typedef, for example, to add more command types, you need to relink the shared variable to the custom control for the shared variable to update its type.

Buffering Terminology

Because some of the implementations of the command variable shown in previous sections use custom controls as their type, it is important to point out that for those types, the buffer size unit shown in the **Network** properties page is bytes and not elements. You need to adjust this size based on application expectations, taking into account the maximum number of outstanding commands the buffer likely holds, the number of commanders, and the average size of a command, including the possible variable-size parameters in the command that can be encapsulated in the command.

Limited Command Buffer

The buffered network variable guarantees that successive values are buffered and sent to all subscribers. This guarantee holds as long as none of the individual buffers for each of the subscribers overflows. Buffer overflows usually trigger warnings at the variable node level, but in a publish-subscribe model, it is difficult to recover from these warnings.

Command History

One of the side effects of having to flush the communication channel to guarantee that no stale commands are reprocessed is that any outstanding commands, issued before the worker process starts, are not processed. An acknowledged command-based architecture helps alleviate this situation by providing feedback to all commanders regarding command delivery in case the worker process is temporarily offline.

Raw Ethernet (TCP/UDP)

While Network-Published Shared Variables are the recommended method for passing data between LabVIEW nodes, an alternative to using shared variables for network communication is raw TCP or UDP communication. TCP and UDP are the low level building blocks of all Ethernet standards. Tools for raw TCP and UDP are natively supported in virtually every programming environment including NI LabVIEW. They offer lower-level communication functions that are more flexible but less user-friendly. Functions such as establishing connections and packaging data must be handled at the application level.

TCP or UDP are good options if you need very low level control of the communications protocol. They may also be options if you need to communicate to third-party devices that do not support Shared Variable communications, however other standard protocols such as Modbus TCP are simpler to use.

TCP provides point-to-point communications with error handling for guaranteed packet delivery. UDP can broadcast messages where multiple devices can receive the same information. UDP broadcast messages may be filtered by network switches and do not provide guaranteed packet delivery.

TCP communication follows a client/server scheme where the server listens on a particular port to which a client opens a connection. Once you establish the connection, you can freely exchange data using basic write and read functions. With TCP functions in LabVIEW, all data is transmitted as strings. This means you must flatten Boolean or numeric data to string data to be written and unflattened after being read. Because messages can vary in length, it is up to the programmer to know how much data is contained within a given message and to read the appropriate number of bytes. Refer to the LabVIEW examples *Data Server.vi* and *Data Client.vi* for a basic overview of client/server communication in LabVIEW.

Building Your Own Communications Protocol

If you are going to use raw TCP or UDP to send and receive data you need to create your own protocol that defines the method of data packing and unpacking. For scalability a communication protocol should have the following characteristics:

- Easily packages and parses data
- Abstracts the transport layer (TCP/IP, UDP, and so on) implementation details
- Minimizes network traffic by sending data only when it is needed
- Minimizes impact in the overall overhead and throughput
- Lends itself to communication with environments other than LabVIEW (C, C++, and so on)

In every messaging protocol there is some overhead (metadata) associated with parsing the data stream on the receiving end. Sending a complete set of meta data with every package adds significant overhead. Because you are likely focusing on high-performance applications, you want to minimize the communication overhead by sending a reduced set of meta information with each packet.

Message and Process Communication

TCP is inherently message-based so your protocol must perform additional logic to also transmit process data. This logic may be as simple as sending all the data periodically or it may implement change detection to reduce the amount of data transmitted over the network.

Example of a Custom Communications Protocol

LabVIEW Example Code
is provided for this section

Developers who create a custom communications protocol are normally trying to optimize code for their specific application. As such, this guide cannot offer step-by-step instructions on creating a custom protocol. However, it provides an example of a custom protocol written in LabVIEW that shows proper implementation. This example is also designed so you can use it directly if you need to communicate to targets outside LabVIEW or it can serve as a starting place for development of your own custom protocol.
The Simple Messaging Protocol for LabVIEW (STM), abstracts some of the complexity of raw TCP communication. Delivery of messages is guaranteed as a result of buffering and acknowledgement within the protocol layer and it uses a name-based mechanism for sending messages. It provides a defined header for each communication message that includes the size of the data packet as well as an index that correlates the message with metadata that has been predefined to describe the contents of the message.
The protocol is implemented in such a way that the client and server exchange metadata when you first establish the connection. This prevents the transmission of redundant information and keeps the overhead of each message at a relatively small 6 bytes. Just as the writer is responsible for converting the data to a string, the reader is responsible for knowing how to interpret the data for a given message. When you read a message, you can parse or convert it to an appropriate data type based on the information contained in the metadata.

Simple Messaging (STM) Protocol – A LabVIEW Implementation

The STM is a generic Ethernet protocol for LabVIEW. You can find an installer for the Simple Messaging Reference Library (STM) component, which includes an application programming interface (API) and example code, on **ni.com**. Some of the examples emphasize LabVIEW Real-Time applications, but most run on any LabVIEW platform.

Metadata

The metadata is implemented as an array of clusters. Each array element contains the data properties necessary to package and decode one variable value. You have defined the only the Name property, but you can use a cluster to customize the STM by adding metaproperties (such as data type) according to your application requirements. The metadata cluster is a typedef, so adding properties should not break your code.

Figure 4.37 shows an example of the metadata cluster configured for two variables: Iteration and RandomData.

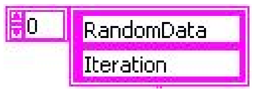


Figure 4.37. Metadata Array of Strings

Before each data variable is transmitted, a packet is created that includes fields for the Data Size, the Metadata ID and the data itself. Figure 4.38 shows the packet format.

Data Size (32 bits)	Metadata ID (16 bits)	Data

Figure 4.38. Packet Format

The Metadata ID field is populated with the index of the meta data array element corresponding to the data variable. The receiving host uses the Metadata ID to index the metadata array to get the properties of the message data.

Simple TCP/IP Messaging API

The STM API is very simple. For basic operation it consists of a Read VI and a Write VI. There are also two supplemental VIs to help with the transmission of the metadata, but their use is not mandatory. Each of the main VIs are polymorphic, which allows for the use with different transport layers. The API for each layer is very similar. The following is a brief description of the main API VIs (only the TCP/IP versions are shown). Additional utility VIs are installed as well and are documented in the VI Help.

STM Write Message

Use this VI to send any type of data to a remote host. This VI creates a packet based on the data, the data name and metadata information. When this VI is called, it retrieves the index of the variable specified by Name in the metadata array. It then assembles the message packet and sends it to the remote host via TCP/IP using the connection ID.

The Connection Info consists of a transport layer reference and the metadata array. This is an output from both the STM Read Metadata and STM Write Metadata VIs.

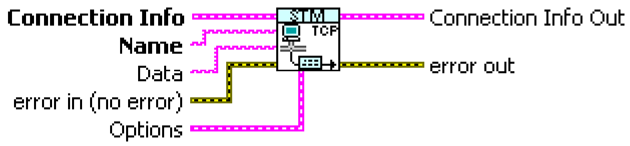


Figure 4.39. STM Write Message (TCP).vi

STM Read Message

Use this VI to receive any type of data from a remote host. This VI reads and unpacks the meta data index and flattened string data. It looks up the meta element and returns it along with the data string. The application can then convert the flattened data to the message data type using the name or other meta properties as a guide.

This VI is usually used inside a loop. Because there is no guarantee that the data arrives at a given time, a "timeout" parameter allows the loop to run periodically.

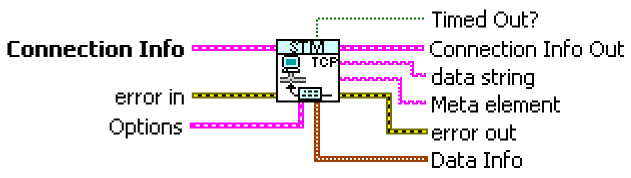


Figure 4.40. STM Read Message (TCP).vi

STM Write Metadata

Use this VI to send meta data information to a remote host. For correct message interpretation, the meta data must be consistent on both receiving and sending side. Instead of maintaining copies of the data on each host, you can maintain the meta data on the server and use this VI to send it to clients as they connect.

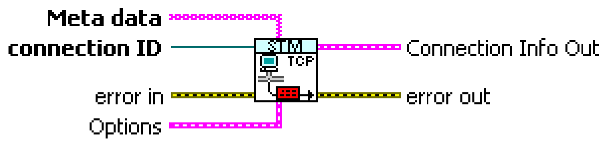


Figure 4.41. STM Write Metadata (TCP).vi

STM Read Metadata

Use this VI to receive meta data information from a remote computer. This VI reads and unpacks the meta data array, which can be passed to the read and write VIs.

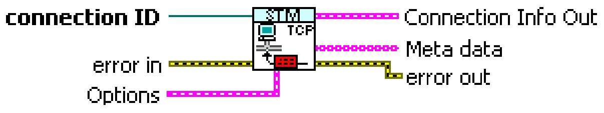


Figure 4.42. STM Read Metadata (TCP).vi

Using the STM API to Send Data

Figure 4.43 shows an example of a data server using the STM API. Notice that this program sends the metadata to a remote host as soon as a connection is established. The example writes two values: the iteration counter and an array of doubles. The metadata contains the description for these two variables.

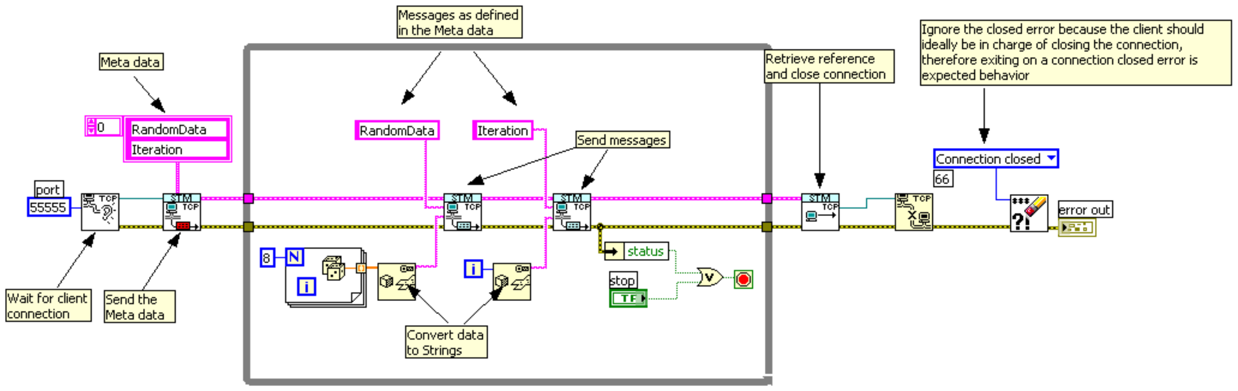

Messages as defined in the Meta data
Meta data
RandomData
Iteration
0
port
55555
Wait for client connection
Send the Meta data
RandomData
Iteration
Send messages
8
N
i
i
Convert data to Strings
status
stop
Retrieve reference and close connection
Ignore the closed error because the client should ideally be in charge of closing the connection, therefore exiting on a connection closed error is expected behavior
Connection closed
66
error out

Figure 4.43. Basic STM Server Example

Using the STM API to Receive Data

Receiving data is also very simple. The design pattern shown waits for the meta data when the connection is established with the server. It then uses the STM Read Message VI to wait for incoming messages. When a message is received, it converts the data and assigns it to a local value according to the meta data name.

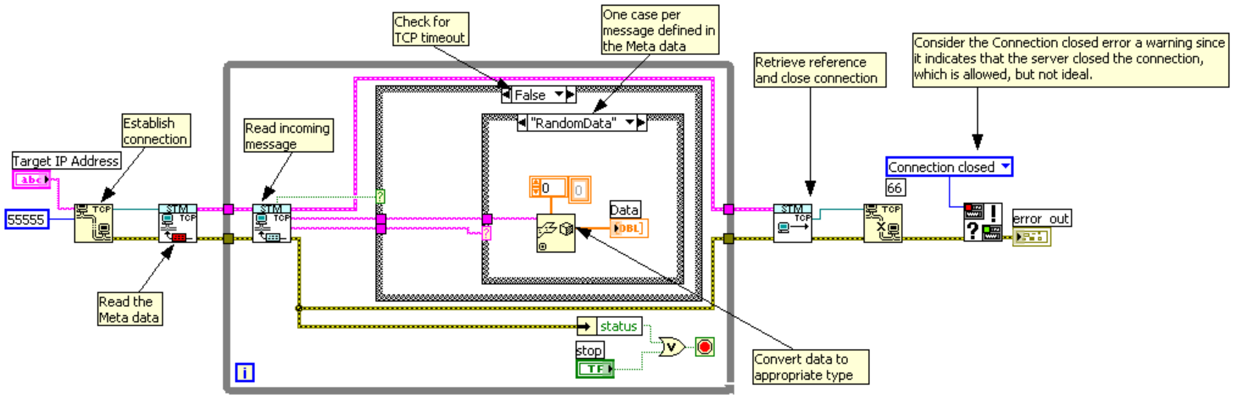

Check for TCP timeout
One case per message defined in the Meta data
Retrieve reference and close connection
Consider the Connection closed error a warning since it indicates that the server closed the connection, which is allowed, but not ideal.
Establish connection
Target IP Address
55555
Read incoming message
Read the Meta data
False
"RandomData"
0
Data
DBL
status
stop
i
Connection closed
66
error out
Convert data to appropriate type

Figure 4.44. Basic STM Client Example

For most LabVIEW to LabVIEW communications, National Instruments recommends network-published shared variables, and for communications to third-party systems, NI recommends higher-level protocols such as Modbus TCP. However, LabVIEW also provides the flexibility to create your own communication protocols such as the STM protocol.

Serial Communication from CompactRIO

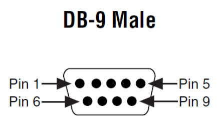
A standard for more than 40 years, RS232 (also known as EIA232 or TIA232) ports can be found on all CompactRIO systems and a wide variety of industrial controllers, programmable logic controllers (PLCs), and devices. RS232 remains a de facto standard for low-cost communications because it requires little complexity, low processing and overhead, and modest bandwidth. While it is fading from the PC industry in favor of newer buses such as USB, IEEE 1394, and PCI Express, RS232 continues to be popular in the industrial sector because of its low complexity, low cost to implement, and wide install base of existing RS232 ports.

The RS232 specification covers the implementation of hardware, bit-level timing, and byte delivery, but it does not define fixed messaging specifications. While this allows for easy implementation of basic byte reading and writing, more complex functionality varies between devices. There are a number of protocols that use RS232-based byte messaging such as Modbus RTU/ASCII, DNP3, and IEC-60870-5.

This section covers how to program byte-level communications for the serial port built into NI CompactRIO real-time controllers using the NI-VISA API.

RS232 Technical Introduction

RS232 is a single-drop communications standard, meaning only two devices can be connected to each other. The standard defines many signal lines used in varying degrees by different applications. At a minimum, all serial devices use the Transmit (TXD), Receive (RXD), and ground lines. The other lines include flow control lines, Request to Send, and Clear to Send, which are sometimes used to improve synchronization between devices and prevent data loss when a receiver cannot keep up with its sender. The remaining lines are typically used with modem-to-host applications and are not commonly implemented in industrial applications with the exception of radio modems.



Pin	232 DTE	232 DCE	422/485
1	DCD[2]	DCD	GND
2	RXD	TXD	CTS+ (HSI+)
3	TXD	RXD	RTS+ (HSO+)
4	DTR[2]	DSR	RXD+
5	GND	GND	RXD–
6	DSR[2]	DTR	CTS– (HSI–)
7	RTS	CTS	RTS– (HSO–)
8	CTS	RTS	TXD+
9	RI[2]	RI	TXD–

Figure 4.45. D-Sub 9-Pin Connector Pinout

As a single-ended bus, RS232 is ideal for shorter-distance communications (under 10 m). Shielded cables reduce noise over longer runs. For longer distances and higher speeds, RS485 is preferred because it uses differential signals to reduce noise. Unlike buses such as USB and IEEE 1394, RS232 is not designed to power end devices.

CompactRIO controllers have an RS232 port that features the standard D-Sub 9-pin male connector (also known as a DE-9 or DB-9). You can use this port to monitor diagnostic messages from a CompactRIO controller when the console out switch is engaged or to communicate with low-cost RS232 devices in your application.

RS232 Wiring and Cabling

There are two standard pinouts for RS232 ports. You can classify RS232 devices as data terminal equipment (DTE), which is like a master or host, and data communications equipment (DCE), which is similar to slaves. DCE ports have inverted wiring so that DCE input pins connect to DTE output pins and vice versa. The serial port on CompactRIO controllers is a DTE port, and the serial port on a device such as a GPS, bar code scanner, modem, or printer is a DCE port.

When connecting a CompactRIO DTE RS232 port to a typical end device with a DCE RS232 port, you use a regular straight-through cable. When connecting two DTE or DCE devices together, such as a CompactRIO controller to a PC, you need to use a null modem cable, which swaps the transmit and receive signals in the cabling.

Device 1	Device 2	Cable Type
DTE	DTE	Null modem cable
DTE	DCE	Straight-through cable
DCE	DTE	Straight-through cable
DCE	DCE	Null modem cable

Table 4.1. Selecting a Cable to Run between Different RS232 Ports

Loopback Testing

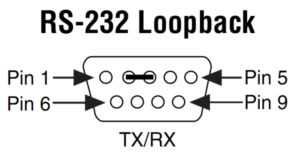


Figure 4.46. RS232 Loopback Wiring

A common technique for troubleshooting and verifying serial port functionality is loopback testing. At a basic level, all you need to do is connect the TXD to the RXD pin. Use a standard RS232 straight-through or loopback cable and a small bent paper clip to short pins 2 and 3. With the loopback cable installed, bytes sent from the read function can be read by the read function. You can verify that software, serial port settings, and drivers are working correctly with this technique.

Serial Communications from LabVIEW

The native CompactRIO RS232 serial port is attached directly to the CompactRIO real-time processor, so you should place code that accesses the serial port in the LabVIEW Real-Time VI. To send and receive bytes to the serial port, use NI-VISA functions.

NI-VISA is a driver that provides a single interface for communicating with byte-level interfaces such as RS232, RS485, GPIB, and so on. Code written with the NI-VISA functions is usable on any machine with a serial port and NI-VISA installed. This means you can write and test a Serial VI on a Windows machine with LabVIEW and then reuse the same code in LabVIEW Real-Time on CompactRIO. You can then directly use thousands of prewritten and verified instrument drivers located at **ni.com/idnet**.

To get started with the serial port, locate the Virtual Instrument Software Architecture (VISA) functions in the palettes, under **Data Communication»Protocols»Serial**.

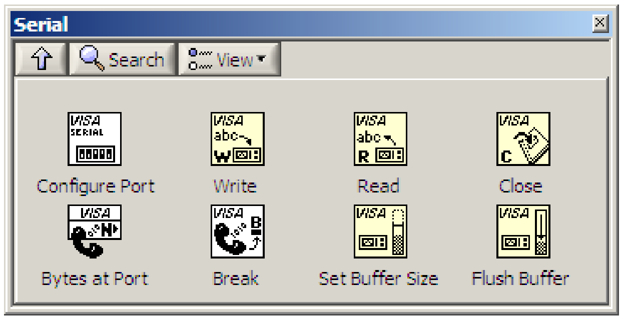


Figure 4.47. VISA Functions

For most simple instrument applications, you need only two VISA functions, VISA Write and VISA Read. Refer to the Basic Serial Write and Read VI in the labview\examples\instr\smplserl.llb for an example of how to use VISA functions.

Most devices require you to send information in the form of a command or query before you can read information back from the device. Therefore, the VISA Write function is usually followed by a VISA Read function. Because serial communication requires you to configure extra parameters such as baud rate, you must start the serial port communication with the VISA Configure Serial Port VI. The VISA Configure Serial Port VI initializes the port identified by the VISA resource name.

It is important to note that the VISA resource name refers to resources on the machine for which the VI is targeted.

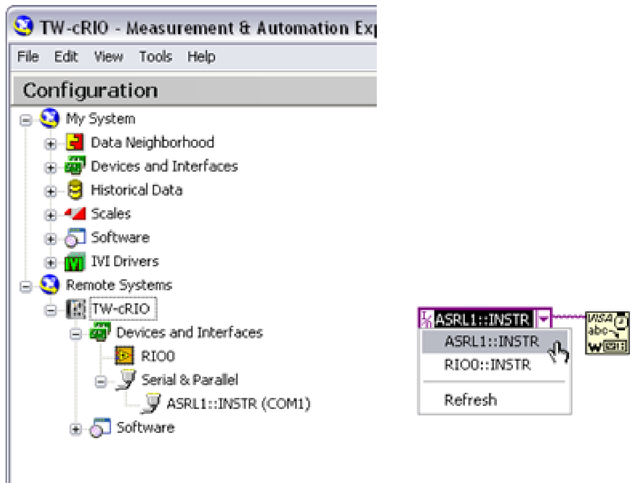


Figure 1.48. You can directly browse to the VISA resource name from the VISA resource control in LabVIEW or from Measurement & Automation Explorer

Check the bottom left corner of the VI to determine which target the VI is linked to. For CompactRIO, use COM1 to use the built-in port. COM 1 on a CompactRIO controller is ASRL1::INSTR. If you are connected to the CompactRIO you can directly browse to the VISA resource name from the VISA resource control in LabVIEW or from Measurement and Automation Explorer (downloadable utility used to configure all NI devices).

Timeout sets the timeout value for the serial communication. Baud rate, data bits, parity, and flow control specify those particular serial port parameters. The error in and error out clusters maintain the error conditions for this VI.

While the ports work at the byte level, the interfaces to the read and write functions are strings. In memory, a string is simply a collection of bytes with a length attached to it, which makes working with many bytes easier to program. The string functions in LabVIEW provide tools to break up, manipulate, and combine data received from a serial port. You can also convert string data to an array of bytes for using LabVIEW array functions to work with low-level serial data.

Because serial operations deal with varying length strings, these tasks are non-deterministic. Additionally, serial interfaces by nature are asynchronous and do not have any mechanisms to guarantee timely delivery of data. To maintain the determinism of your control loops, when programming serial applications keep any communications code in a separate loop from the control code.

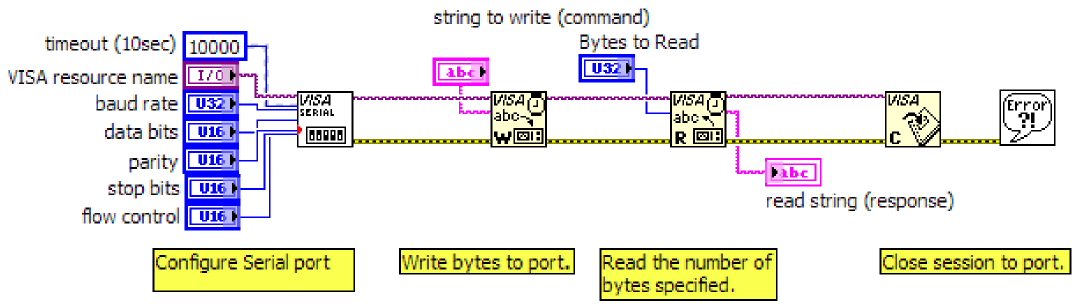


Figure 4.49. Simple Serial Communication Example

The VISA Read function waits until the number of bytes requested arrives at the serial port or until it times out. For applications where this is known, this behavior is acceptable, but some applications have different lengths of data arriving. In this case, you should read the number of bytes available to be read, and read only those bytes. You can accomplish this with the Bytes at Serial port property accessible from the Serial Palette.

You can access serial port parameters, variables, and states using property nodes. If you open the VISA Configure Serial Port VI and examine the code, you see that this VI simply makes a call to many property nodes to configure the port. You can use these nodes to access advanced serial port features, including the auxiliary data lines, bytes waiting to be read/written, and memory buffer sizes. Figure 4.49 shows an example of using a property node to check the number of bytes available at the port before reading.

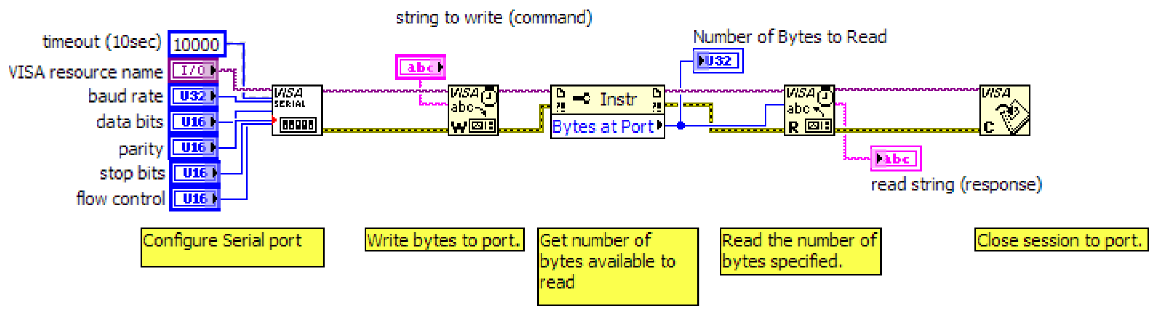


Figure 4.50. Using a VISA Property Node to Read the Number of Bytes

In general, working with devices can range from simply reading periodically broadcasted bytes such as GPS data to working with complex command structures. For some devices and instruments, you can find preprogrammed instrument drivers on **ni.com/idnet**, which can simplify development. Because these drivers use NI-VISA functions, they work with the onboard CompactRIO serial port.

Instrument Driver Network

To help speed development National Instruments has worked with major measurement and control vendors to provide a library of ready to run instrument drivers for more than 7,000 devices. An instrument driver is a set of software routines that control a programmable instrument. Each routine corresponds to a programmatic operation such as configuring, reading from, writing to, and triggering the instrument. Instrument drivers simplify instrument control and reduce program development time by eliminating the need to learn the programming protocol for each device.

Where to Find Instrument Drivers and How to Download Them

You can find and download instrument drivers two different ways. If you are using LabVIEW 8.0 or later, the easiest way is to use the NI Instrument Driver Finder. If you have an older version of LabVIEW, then you can use the Instrument Driver Network (**ni.com/idnet**).

Use the NI Instrument Driver Finder to find, download, and install LabVIEW Plug and Play drivers for an instrument. Select Tools»Instrumentation»Find Instrument Drivers to launch the Instrument Driver Finder. This tool searches IDNet to find the specified instrument driver. Refer to Figure 4.51 to see how to launch the NI Instrument Driver Finder from LabVIEW.

You can use an instrument driver for a particular instrument as is. However, LabVIEW Plug and Play instrument drivers are distributed with their block diagram source code, so you can customize them for a specific application. You can create instrument control applications and systems by programmatically linking instrument driver VIs on the block diagram.

Example of Serial Communication from LabVIEW

LabVIEW Example Code
is provided for this section

Now modify the original PID example so that instead of getting the temperature value from a built-in thermocouple module, you retrieve it from a serial instrument. In this case, read the value from a Lake Shore Cryotronics 211.

On IDNet, you can search for the instrument and download a driver. You can also directly search for a driver from LabVIEW.

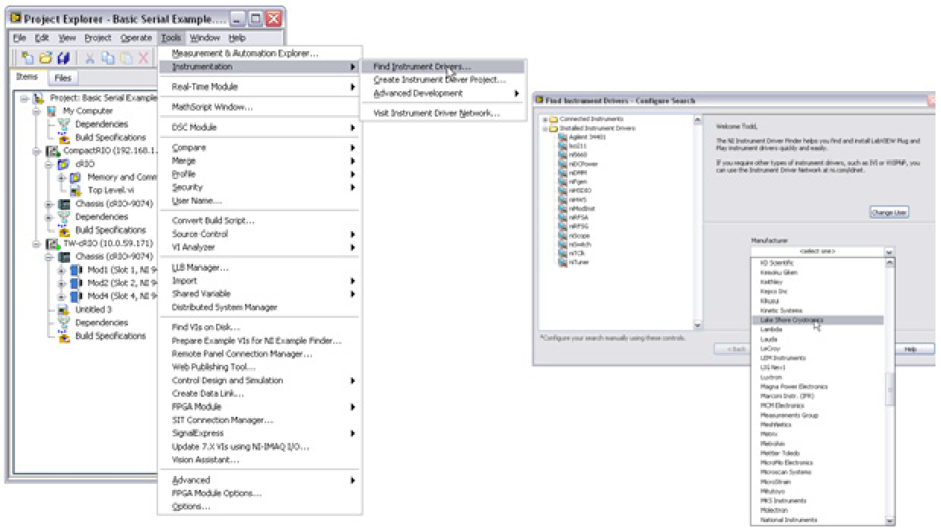

Project Explorer - Basic Serial Example...
File Edit View Project Operate Tools Window Help
Measurement & Automation Explorer...
Instrumentation
Find Instrument Drivers...
Create Instrument Driver Project...
Advanced Development
Visit Instrument Driver Network...
Find Instrument Drivers - Configure Search
Manufacturer
Lake Shore Cryotronics

Figure 4.51. With LabVIEW, you can directly search and install communications drivers for more than 7,000 devices.

The utility automatically downloads and saves the drivers in the instr.lib file under the LabVIEW directory. Drivers saved in this folder automatically show up on the palette when LabVIEW is launched, so, in LabVIEW, you see a new entry in the Instrument Drivers palette for the LSCI 211 temperature monitor.

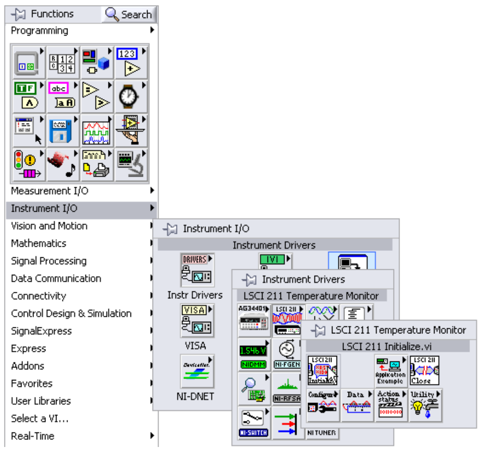


Figure 4.52. You can browse instrument drivers from the Instrument I/O palette.

Next modify the PID application so that Thermocouple 1 is a single-process shared variable with a real-time FIFO (instead of an I/O alias). Set a default value for Thermocouple 1 in the initialization routine.

Finally add a second loop for the serial communications task. Before entering the loop initialize serial communications to the device, in the loop read the temperature and write it to the memory table (the single-process shared variable). When you stop the loop close the serial communications.

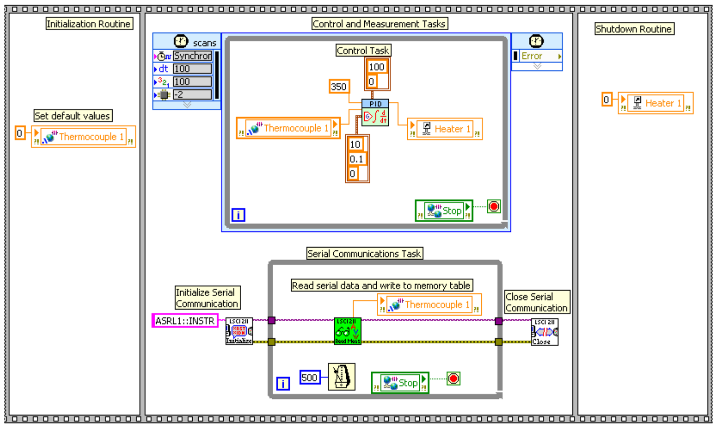

Initialization Routine
Set default values
Thermocouple 1
Control and Measurement Tasks
Control Task
Thermocouple 1
Heater 1
PID
stop
Serial Communications Task
Initialize Serial Communication
ASRL1::INSTR
Read serial data and write to memory table
Thermocouple 1
Close Serial Communication
Stop
Shutdown Routine
Heater 1

Figure 4.53. A Complete Application Reading Temperature Data from a Serial Device and Passing the Data to a High-Priority Control Loop

RS232 and RS422/RS485 Four-Port NI C Series Modules for CompactRIO

If your application requires higher performance or additional RS232 ports and/or RS485/RS422 ports, you can use the NI 9870 and NI 9871 serial interfaces for CompactRIO to add more serial ports. These boards are accessed directly from the FPGA using LabVIEW FPGA. LabVIEW FPGA programming is covered in detail in a later section.

Communicating with PLCs and Other Industrial Networked Devices

Often, applications call for integrating NI PACs such as CompactRIO into an existing industrial system or with other industrial devices. These systems can consist of traditional PLCS, PACs, or a wide variety of specialty devices such as motor controllers, sensors, and human machine interfaces. These devices come with a wide arrange of communication options ranging from simple RS232 to specialized high-speed industrial networks.

There are three common methods to connect CompactRIO to an industrial system or PLC. You can directly wire the CompactRIO to the PLC using standard analog or digital IO. This method is simple but scales for only very small systems. For larger systems, you can use an industrial communications protocol to communicate between the CompactRIO and the PLC. For large SCADA applications OPC may be a good tool. OPC scales to very high channels but uses Windows technology and, therefore, requires a Windows PC to perform the communications.

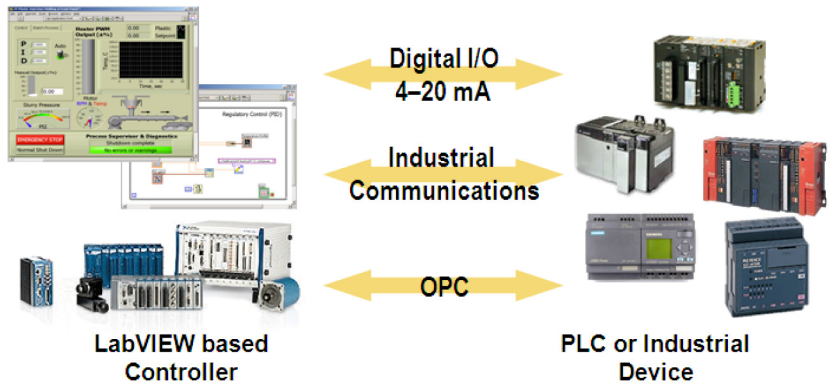


Figure 4.54. Three Methods for Connecting a CompactRIO Controller to an Industrial Device

Industrial Communications Protocols

Beyond using standard I/O, the most common way to connect a CompactRIO to other industrial devices is using industrial communications protocols. Most protocols use standard physical layers such as RS232, RS485, CAN, or Ethernet. As discussed earlier, buses like RS232 and Ethernet provide the physical layer for simple communications, but lack any predefined methods of higher-level communication, providing just enough for a user to send and receive individual bytes and messages. When working with large amounts of industrial data, handling those bytes and converting them into relevant control data quickly becomes tedious. Industrial communications protocols provide a framework for how data is communicated. They are fundamentally similar to instrument drivers discussed earlier although they typically run a more sophisticated stack.

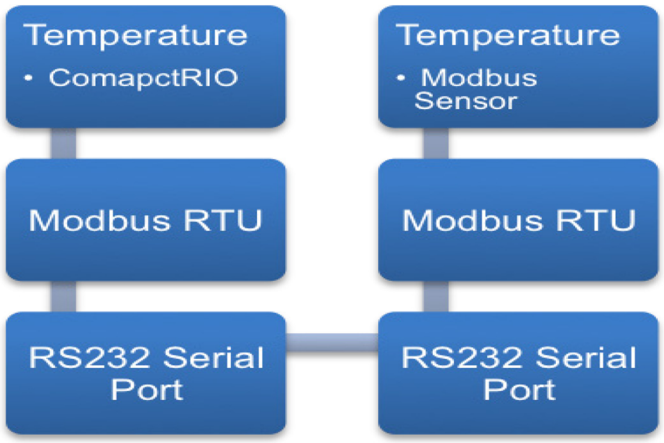


Figure 4.55. Typical Industrial Protocol implementation

Industrial protocols take a low level bus like RS232 or TCP/IP and add predefined techniques for communication on top. This is very similar to how HTTP, the protocol for web content delivery, sits on top of TCP/IP. It is possible to write your own HTTP client like Internet Explorer or Firefox, but it takes a large effort and may not accurately connect to all servers without hours of testing and verification. Similarly, devices and controllers that support industrial networking standards are much easier to integrate at the program level and abstract the low-level communication details from the end user, allowing engineers to focus on the application.

Because CompactRIO has built-in serial and Ethernet ports it can easily support many protocols including Modbus TCP, Modbus Serial, Ethernet-IP and others. There are also plug-in modules available for communication with most common industrial protocols such as PROFIBUS, CANopen and more.

When native communication is not available another option is to use gateways. Gateways are protocol translators and can convert between different industrial protocols. For example, if you wanted to connect to a CC-Link network you can do this by using a gateway. The gateway can talk Modbus TCP on the PAC end and translate that to CC-Link on the other end.

Modbus Communications

Modbus is the most common industrial protocol in use today. It was developed in 1979 and supports both serial and Ethernet physical layers. Modbus is an application layer messaging protocol for client/server communication between devices connected on a bus or network. Modbus can be implemented using asynchronous serial transmission to communicate with touch screens, PLCs, and gateways to support other types of industrial buses. Modbus can be used with a CompactRIO and its on-board serial or Ethernet boards. Note that the on-board serial port of CompactRIO uses RS232 while some Modbus devices may use the RS485 electrical layer. In this case, a common RS485 to RS232 adapter can be used.

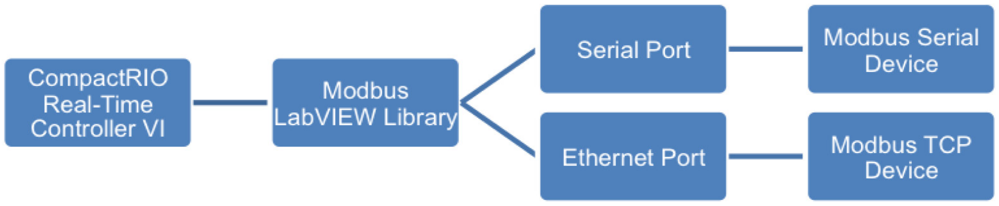


Figure 4.56. Software Architecture for Communicating with Modbus Devices

The Modbus Serial protocol is based on a master/slave architecture. An address, ranging from 1 to 247, is assigned to each slave device. Only one master is connected to the bus at any given time. Slave devices do not transmit information unless a request is made by the master device, and slave devices cannot communicate to other slave devices.

Information is passed between master and slave devices by reading and writing to registers located on the slave device. The Modbus specification distinguishes the use of four register tables, each capable of 65,536 items and differentiated by register type and read-write access. Register address tables do not overlap in this implementation of LabVIEW.

Tables	Object Type	Type of Access	Comments
Discrete Inputs	Single-Bit	Read Only	Master only can read. Only the slave device can change its register values.
Coils	Single-Bit	Read-Write	Both master and slave can read and write to these registers.
Input Registers	16-Bit Word	Read Only	Master only can read. Only the slave device can change its register values.
Holding Registers	16-Bit Word	Read-Write	Both master and slave can read and write to these registers.

Table 4.2. Information is passed between master and slave devices by reading and writing to registers located on the slave device.

For a comprehensive introduction to Modbus, visit **ni.com**. You can also download the free Modbus LabVIEW Library. Once installed, using the Modbus Library is similar to using the NI-VISA functions for serial communications. The palette installs to **Functions»User Libraries»NI Modbus**. Examples are included in the third and fourth columns of the palette.

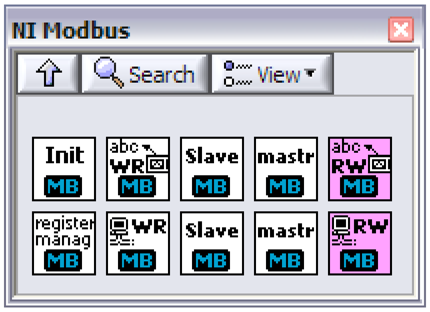


Figure 4.57. LabVIEW offers a Modbus Library for master and slave communication.

Before starting with Modbus programming, you must determine several important parameters of your Modbus device by consulting its documentation:

1. Master or slave? If your Modbus device is a slave, configure your CompactRIO as a master. This is the most common configuration. Likewise, connecting to a Modbus master (such as another PLC) requires configuring the CompactRIO as a slave.

2. Serial or Ethernet? Modbus Ethernet devices are sometimes called "Modbus TCP" devices.

3. For Modbus serial:

   - RS232 or RS485? Many Modbus devices are RS232, but some use the higher-power RS485 bus for longer distances. RS232 devices can plug directly into a CompactRIO system, but an RS485 device needs an RS232-RS485 converter to function properly.

   - RTU or ASCII mode? RTU/ASCII refers to the way data is represented on the serial bus. RTU data is in raw binary, whereas ASCII data is human-readable on the raw bus. This parameter is required by the Modbus VIs.

4. What are the register addresses? Every Modbus device has a mapping of its I/O to registers, coils, and discrete inputs for reading and writing, represented by a numerical address. Per Modbus convention, a register address is always one less than the register name. This is similar to the first element of an array being element 0. The Modbus LabVIEW Library requires register addresses, not register names.

Modbus Example

The example shows using CompactRIO in its most common Modbus implementation as a Modbus RTU Master on an RS232 serial port. The configuration values are not wired for simplicity and therefore run as their defaults:

- Baud rate: 9600, no parity, no flow control, 10 s timeout
- RTU data format
- Slave address 0

If your device uses different parameters, make sure to wire the inputs on all of the affected VIs.

While this example shows using Modbus on a serial port, it can be easily adapted to using Modbus over TCP/Ethernet by replacing the VIs with their Ethernet equivalents in the palettes.

When run, this example polls the most recent temperature value from a Modbus device, scale the data, and place it in the main memory table. It then reads the most recent output value, scales the data back to integer values, and updates the slave with the new value. To integrate this example into your control architecture, run this loop as another parallel "driver" task. Inputs and outputs from the Modbus device are stored locally in a memory table using single-process shared variables with real-time FIFOs.

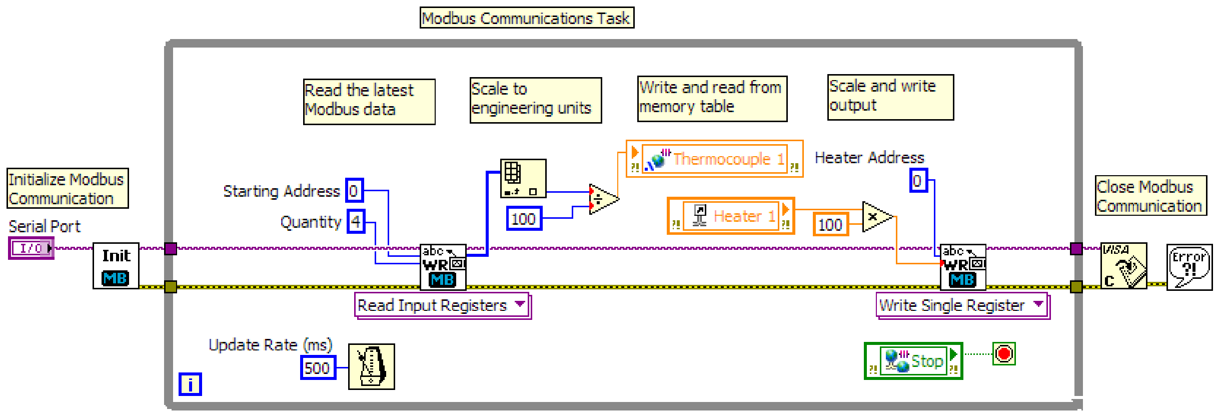


Figure 4.58. A LabVIEW Example of Modus Communications

First, you must select the serial port. For CompactRIO, this is ASRL1::INSTR. If the VI is targeted to a connected CompactRIO, this field should populate automatically. Then, the main Modbus loop begins. This example shows a simple input register read/write. Registers are used for multivalue data and are 16-bit integers. Typically, input registers are read in a batch, so the starting address is specified and the number of sequential addresses to be read is specified. In this way, you can read many inputs with one call to the Read VI.

For the example, the first value returned is indexed from the array, scaled to engineering units, and placed in the memory table. Dependent on the sensor, calibration, and other factors, the scaling is determined by the end user. Typically users place this scaling function in a subVI.

To write the most recent output data to the Modbus network, a Modbus Write is called. Once again, data must be scaled back to the integer form of data used by Modbus, and then published to the appropriate host address. The example shows a single register write; however there are functions for updating many values at once.

This example shows a loop rate update of 500 ms, however this rate can be adjusted to achieve the update rate desired to the end device. Keep in mind that the maximum loop rate is dependent on the number of registers/coils being read/written, the baud rates of the Modbus connection, and the capabilities of the end device. In general, Modbus/TCP devices are able to be updated more quickly than Modbus serial devices. Also consider that excessive loop rates under 10 ms can use excessive CPU time on the CompactRIO and leave less processing time for other tasks.

EtherNet/IP

Another very common industrial protocol is EtherNet/IP. EtherNet/IP is a protocol commonly found on modern Rockwell (Allen-Bradley) PLCs and uses standard Ethernet cabling for communication with industrial I/O. NI has a library available on **ni.com/labs** that allows CompactRIO to communicate via Explicit Messaging to directly read and write tags from the PLC or to be a full EtherNet/IP Class 1 Adapter (slave). While the library is a labs product, it is built from a verified EtherNet/IP protocol stack and has been thoroughly tested. Install the driver by running the setup program.

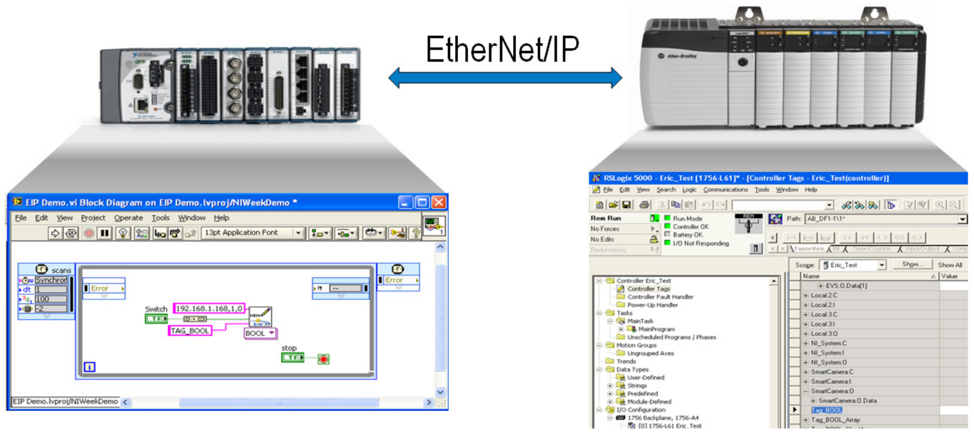


Figure 4.59. With EtherNet/IP, you can directly read and write to tags on a Rockwell PLC from CompactRIO.

The EtherNet/IP driver integrates into the control architecture using the same techniques as the Modbus library. Create another parallel loop that works as a "driver" task. Pass data from the EtherNet/IP network to the memory table using single-process shared variables with a real-time FIFO.

OPC

OPC, or OLE for Process Control, is a common protocol used in many applications with high channel counts and low update rates, which are common in process industries such as oil and gas and pharmaceutical manufacturing. OPC is designed for connecting HMI and SCADA systems to controllers – it is not generally used for communication between controllers. A common use for OPC is to integrate a CompactRIO controller into a third-party system for HMI and SCADA applications.

The OPC specification is a large collection of standards that span many applications. This discussion focuses on OPC Data Access, the core specification that defines how to universally move data around using common Windows technologies. The network shared variable in LabVIEW provides a gateway to OPC.

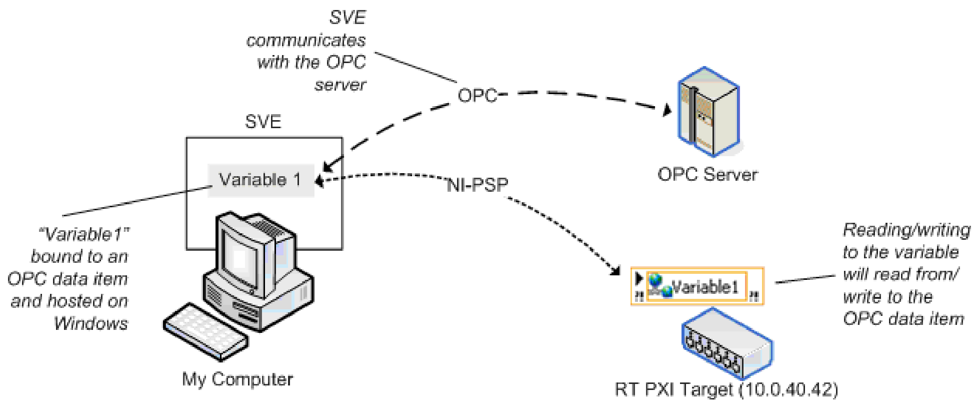

SVE communicates with the OPC server
OPC
SVE
Variable 1
NI-PSP
OPC Server
"Variable1" bound to an OPC data item and hosted on Windows
My Computer
Variable1
Reading/writing to the variable will read from/ write to the OPC data item
RT PXI Target (10.0.40.42)

Figure 4.60. The network-published shared variable provides a gateway to OPC.

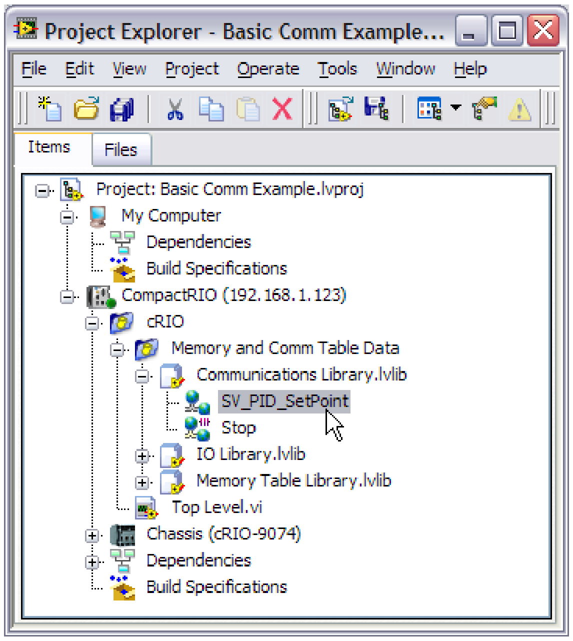
OPC is a Windows-only technology and CompactRIO cannot directly communicate using the OPC protocol. Instead a Windows PC can communicate to the CompactRIO using the PSP protocol and can translate and republish the information via OPC. When running on a Windows machine, the NI Shared Variable Engine functions as an OPC

Server and performs this translation and republishing automatically. This means other OPC clients such as third-party SCADA packages or process control software can access and retrieve data from the CompactRIO device.

Publishing Data from a CompactRIO System over OPC

The following example describes how to use the windows Shared Variable Engine to make values available as an OPC server. The example assumes you are already publishing data onto the network via the PSP protocol (network-published shared variables) hosted by the CompactRIO system. In this example the network shared variable is named "**SV_PID_SetPoint.**"

1. The Windows PC also hosts network-published shared variables. Any Windows based network shared variable by default is also published as an OPC item.
2. The Windows variables are "bound" to the data from the CompactRIO system causing the Windows variable engine to subscribe to and receive data from the CompactRIO system and update the OPC items.
3. Other OPC clients connect to the Windows PC and use the data for display, HMI, etc.
4. CompactRIO publishes data onto the network using networked-published shared variables hosted on the CompactRIO controller.


Project Explorer - Basic Comm Example...
File Edit View Project Operate Tools Window Help
Items Files

Project: Basic Comm Example.lvproj
My Computer
Dependencies
Build Specifications
CompactRIO (192.168.1.123)
cRIO
Memory and Comm Table Data
Communications Library.lvlib
SV_PID_SetPoint
Stop
IO Library.lvlib
Memory Table Library.lvlib
Top Level.vi
Chassis (cRIO-9074)
Dependencies
Build Specifications

Figure 4.61. Network-Published Shared Variables Hosted on the CompactRIO System

5. The Windows PC also hosts network-published shared variables. Any Windows-based network shared variable by default is also published as an OPC item. In the LabVIEW Project, Windows shared variables are located under My Computer. Create a shared variable right-clicking My Computer and selecting New»Variable.

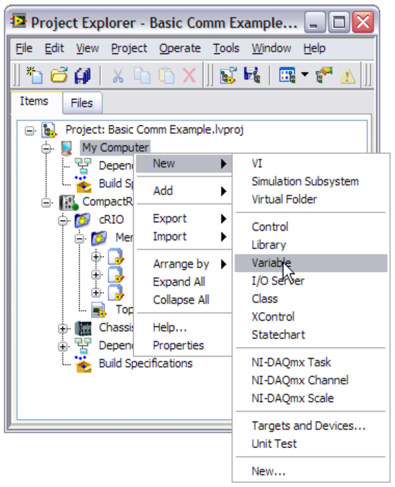


Figure 4.62. Network-published shared variables hosted on Windows are automatically published via OPC.

6. The Windows variables are "aliased" to the data from the CompactRIO causing the Windows variable engine to subscribe and receive data from the CompactRIO and update the OPC items. In the Shared Variable Properties window, give the variable a useful name such as "SV_PID_SetPoint_OPC." Check the Enable Aliasing box and click the Browse button to browse to the SV_PID_SetPoint variable on the CompactRIO.

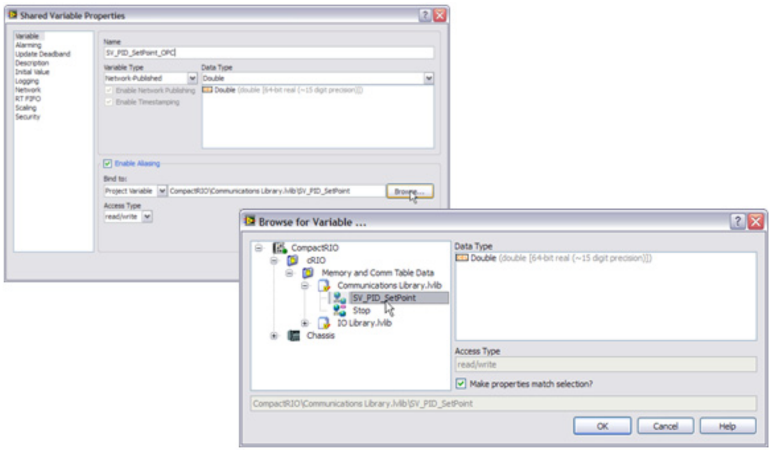
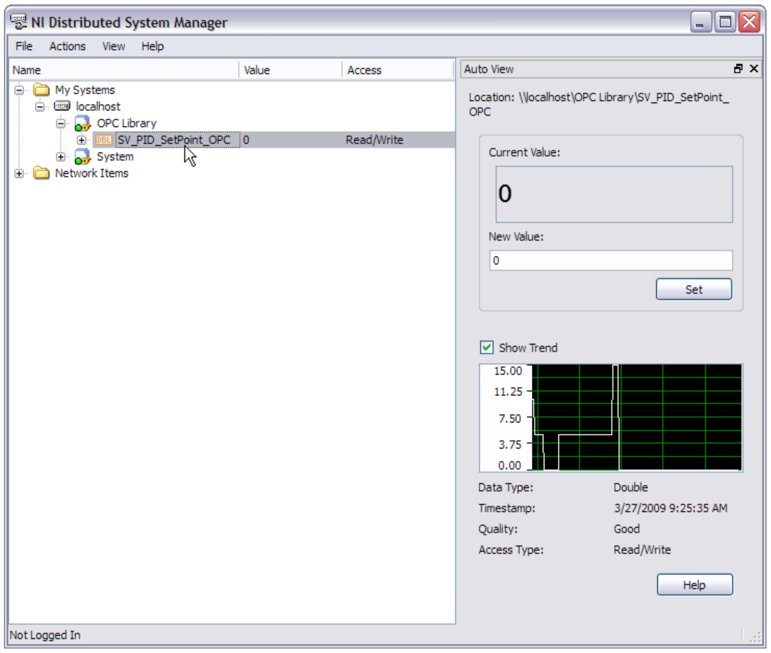


Figure 4.63. An aliased network-published shared variable provides a gateway between CompactRIO and OPC.

7. Save the new library that is created with a useful name, such as "OPC Library.lvlib."

8. Deploy the variables to the shared variable engine by right clicking the library and selecting Deploy.

9. The shared variable is now published an OPC item. You can verify the variable is working correctly by launching an OPC client such as the NI Distributed System Manager (**Start»Programs»National Instruments»NI Distributed System Manager**).



Name | Value | Access
My Systems
localhost
OPC Library
SV_PID_SetPoint_OPC | 0 | Read/Write
System
Network Items

Auto View

Location: \\localhost\OPC Library\SV_PID_SetPoint_OPC

Current Value: 0

New Value: 0

Data Type: Double
Timestamp: 3/27/2009 9:25:35 AM
Quality: Good
Access Type: Read/Write

Not Logged In

Figure 4.64. The OPC clients, such as the NI Distributed System Manager, can read and write to the OPC item.

10. If you have an OPC client available, such as the LabVIEW DSC Module, you can verify the variable is working from that client as well. NI shared variable OPC items are published under the National Instruments.Variable Engine.1 server name.

11. Once variables are deployed in the Shared Variable Engine, they stay deployed with subsequent reboots of the OPC server machine.

Find more information on the Shared Variable Engine and OPC on **ni.com**.

CHAPTER 5
Adding I/O to the CompactRIO System

Adding I/O to CompactRIO

A typical CompactRIO system can hold up to eight NI C Series modules in its backplane, but some control applications require more I/O channels or I/O that is distributed from the main controller. You can use two methods to easily expand your hardware system:

- Ethernet I/O
- Deterministic Ethernet I/O

Ethernet I/O

Expanding a CompactRIO system using Ethernet I/O involves the addition of one or more CompactRIO systems to the same network. The main controller is responsible for running the real-time control loop using I/O from two different systems. The expansion controller, which has either no logic running on its processor or simple logic to implement a network watchdog, provides expansion or distributed I/O for the main controller. Programmers can use existing network infrastructure such as switches and routers. Although full duplex switch networks eliminate packet collisions, switches introduce jitter and general Ethernet should only be used in applications where deterministic communication is not required. If synchronization between the local I/O and expansion I/O is required, see the "Deterministic Ethernet I/O" section for more information.

Using another CompactRIO system as Ethernet I/O lends itself to several different architectures such as the following:

1. The host PC runs the LabVIEW code and the two CompactRIO systems provide I/O and/or additional processing
2. The main controller and the expansion controller both run embedded real-time/FPGA logic and share information using I/O aliases or shared variables

Step 1. Configure the Expansion System

The simplest method of adding another CompactRIO system as Ethernet I/O involves connecting two CompactRIO systems to the same network. Detailed instructions on how to configure each the controller and its IP address are covered later in this document in a Getting Started with CompactRIO section.

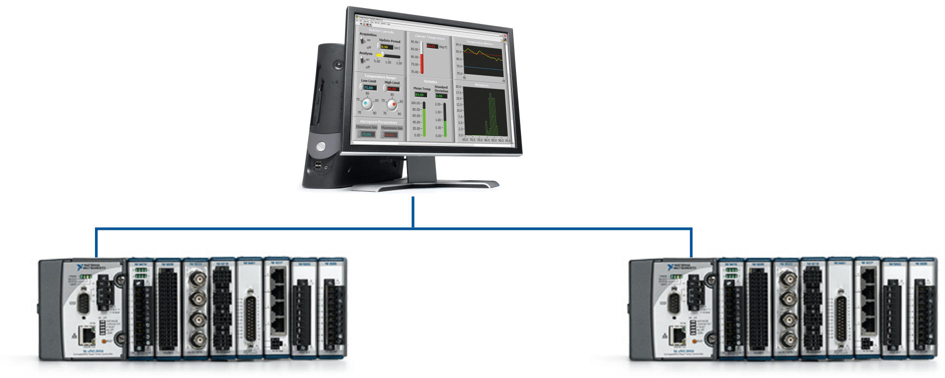
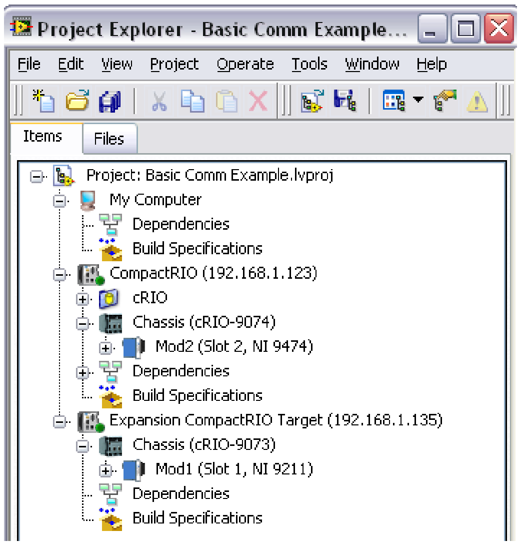


Figure 5.1. The simplest method of adding another CompactRIO system as Ethernet I/O involves connecting two CompactRIO systems to the same network.

Once both controllers have IP addresses, add both controllers into the LabVIEW Project. Select Scan Interface as the programming mode for both. The LabVIEW Project Explorer window automatically adds all the modules and I/O channels in the main and expansion systems.

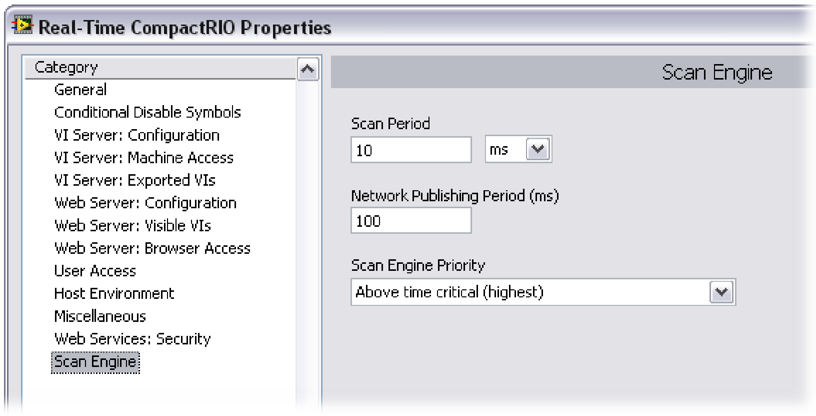

Project Explorer - Basic Comm Example...
File Edit View Project Operate Tools Window Help
Items
Files

Project: Basic Comm Example.lvproj
My Computer
Dependencies
Build Specifications
CompactRIO (192.168.1.123)
cRIO
Chassis (cRIO-9074)
Mod2 (Slot 2, NI 9474)
Dependencies
Build Specifications
Expansion CompactRIO Target (192.168.1.135)
Chassis (cRIO-9073)
Mod1 (Slot 1, NI 9211)
Dependencies
Build Specifications

Figure 5.2. The LabVIEW Project Explorer window displays two CompactRIO systems connected over TCP/IP.

The I/O from the expansion controller is automatically published via the PSP protocol. You can directly drag the I/O onto the block diagram of the main controller to read and write to the expansion controller.

You can adjust how often the expansion chassis sends updated data on the network by changing the Network Publishing Period. Right-click on the controller and select **Properties**. In the Real-Time CompactRIO Properties window, select **Scan Engine** on the left-hand menu to access these periods.



Scan Engine

Scan Period: 10 ms

Network Publishing Period (ms): 100

Scan Engine Priority: Above time critical (highest)

Figure 5.3. The scan period and network publishing period can be adjusted in the properties of the CompactRIO controller.

Considerations for Ethernet Disconnections

Because the expansion controller is designed to run an embedded real-time program, there is no network watchdog. This means that if the connection between the main controller and the expansion controller is broken, the outputs remain in their last state. If this is not acceptable, you should write a small real-time application for the expansion controller that monitors a heartbeat from the main controller and sets outputs to safe states if disconnected.

Step 2. Add the I/O into the Scan of the Main Controller

In the program running on the main controller read the I/O and add it to the I/O scan. An earlier section goes into detail of adding data to the I/O scan from sources such as Ethernet. However here is a brief review.

Asynchronous Nondeterministic I/O

Because you are reading data over standard Ethernet this I/O is nondeterministic. For this use case, you should create a regular while loop or a timed loop with a lower priority. This allows the control task to run deterministically and reliably because it is not affected by the possibly high-jitter I/O device in the custom I/O scan task. Set the task timing based on the required update rate.

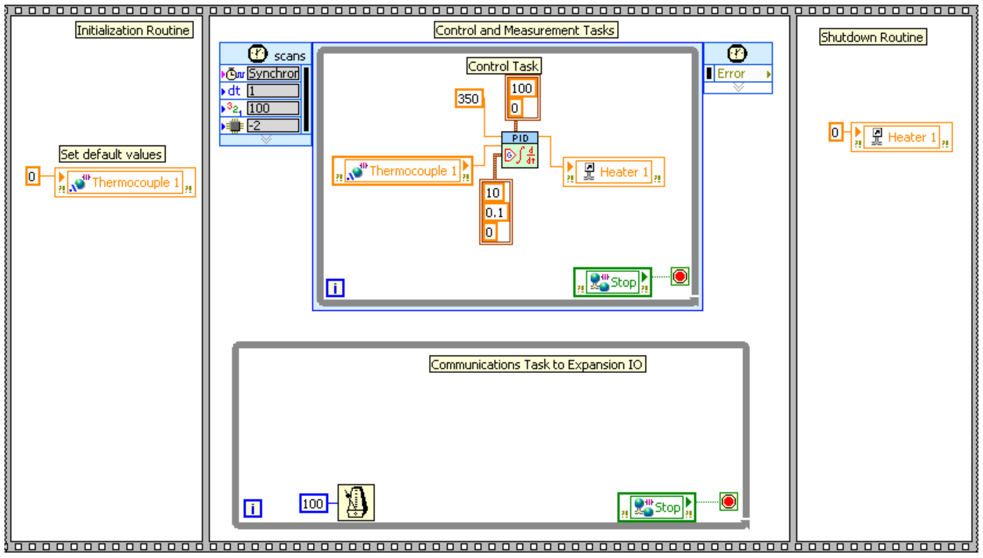
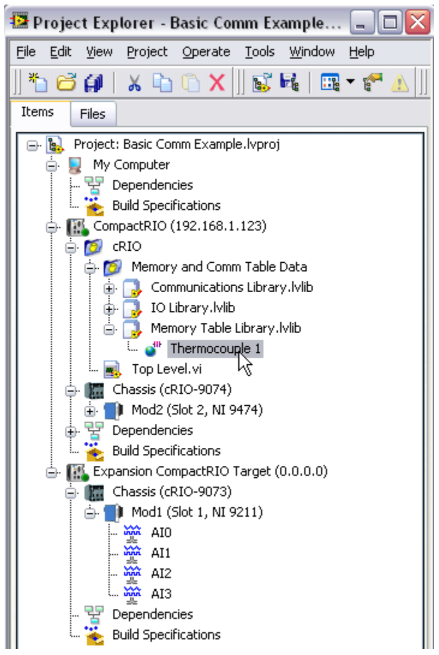


Figure 5.4. Add a loop or timed loop to handle the I/O expansion scan task.

Step 3. Copy the Data into the Memory Table

To access the data read and written in the custom I/O scan task throughout the application, create single-process shared variables with real-time FIFOs, as discussed in the section "Passing Data between Tasks."


Project Explorer - Basic Comm Example...
File Edit View Project Operate Tools Window Help
Items
Files

```
Project: Basic Comm Example.lvproj
    My Computer
        Dependencies
        Build Specifications
    CompactRIO (192.168.1.123)
        cRIO
            Memory and Comm Table Data
                Communications Library.lvlib
                IO Library.lvlib
                Memory Table Library.lvlib
                    Thermocouple 1
            Top Level.vi
        Chassis (cRIO-9074)
            Mod2 (Slot 2, NI 9474)
        Dependencies
        Build Specifications
    Expansion CompactRIO Target (0.0.0.0)
        Chassis (cRIO-9073)
            Mod1 (Slot 1, NI 9211)
                AI0
                AI1
                AI2
                AI3
        Dependencies
        Build Specifications
```

Figure 5.5. Use single-process shared variables to share the custom I/O data with other tasks.

Tip: For high-channel-count systems, you can streamline the creation of multiple I/O aliases by exporting and importing .csv spreadsheet files using the Multiple Variable Editor.

In the loop read the variables off the network, check for warnings or errors, and write the data into the single-process shared variables.

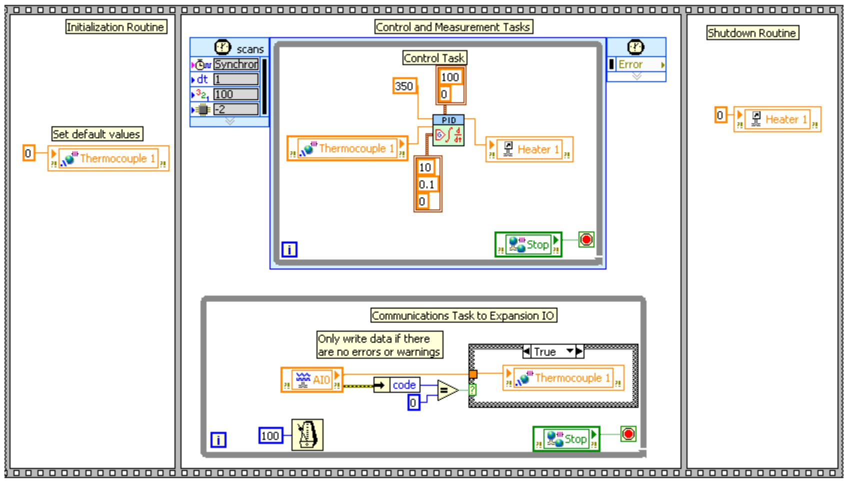


Figure 5.6. In a communications task read the data from the expansion chassis and write it into the memory table using single-process shared variables.

Deterministic Ethernet I/O

LabVIEW Example Code
is provided for this section

In certain applications, the main I/O and expansion I/O systems require tight synchronization, such that all the inputs and outputs are updated at the same time. Using a deterministic bus allows the main controller to know not only when the expansion I/O is updated but also exactly how long the data takes to arrive. You can easily distribute CompactRIO with deterministic Ethernet technology using the NI 9144 expansion chassis.

Overview of the NI 9144 – Deterministic Ethernet Chassis

The NI 9144 is a rugged expansion chassis for expanding CompactRIO systems using a deterministic Ethernet protocol called EtherCAT. With a master-slave architecture, you can use any CompactRIO controller with two Ethernet ports as the master device and the NI 9144 as the slave device. The NI 9144 also has two ports that permit daisy chaining from the controller to expand time-critical applications.

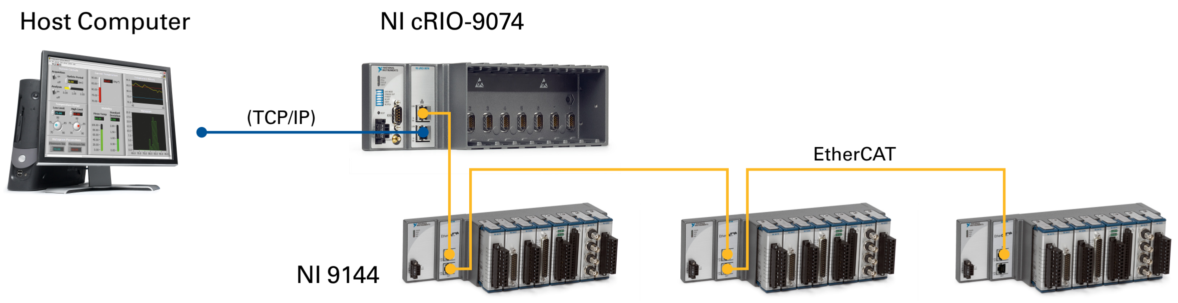


Figure 5.7 The CompactRIO hardware architecture expands time-critical systems using the NI 9144 deterministic Ethernet chassis.

This eight-slot chassis supports more than 30 analog and digital C Series modules with single-point measurements. Unlike the CompactRIO controller, the NI 9144 is primarily designed to provide distributed I/O because it does not contain a processor that can be targeted by the LabVIEW Real-Time Module. However, it incorporates a network watchdog that sets outputs to a default (0V or off) state if it is disconnected from the master and is directly integrated into the NI Scan Engine with I/O variables.

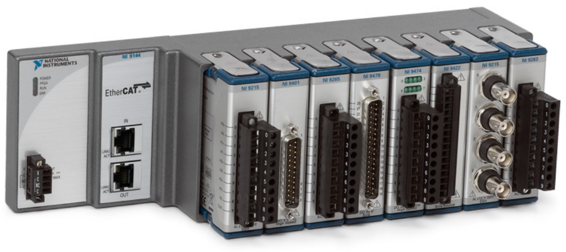


Figure 5.8. NI 9144 Expansion Chassis

Step 1. Configure the Deterministic Expansion Chassis

To enable EtherCAT support on the CompactRIO controller (an NI cRIO-9074 in this example), you must install the NI-Industrial Communications for EtherCAT driver on the host computer. The NI-Industrial Communications for EtherCAT driver is available on the CD included with the NI 9144 chassis and on **ni.com** for free download.

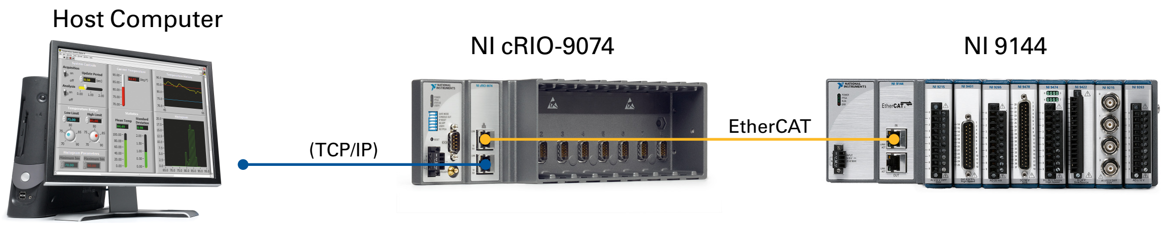


Figure 5.9. To configure the deterministic Ethernet system, connect Port 1 of the cRIO-9074 to the host computer and connect Port 2 to the NI 9144.

Configure the Real-Time CompactRIO to Be the Deterministic Bus Master

1. Launch Measurement & Automation Explorer (MAX) and connect to the CompactRIO controller.

2. In MAX, expand the controller under **Remote Systems** in the **Configuration** pane. Right-click **Software** and select **Add/Remove Software**.

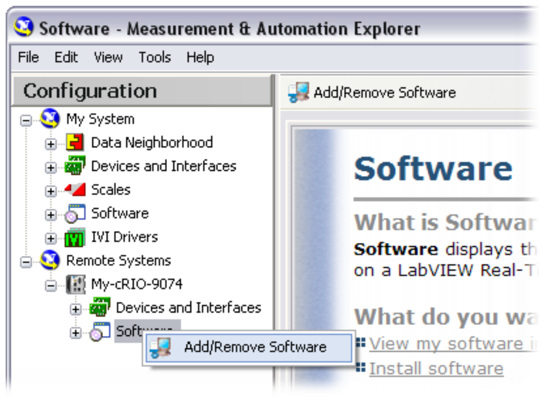
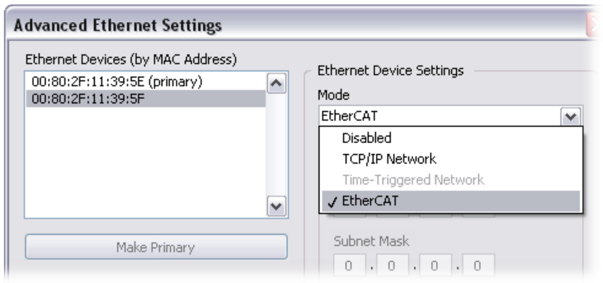


Figure 5.10. Install the appropriate software in Measurement & Automation Explorer.

3. If the controller already has LabVIEW Real-Time and NI-RIO installed on it, select **Custom software installation** and click **Next**. Click **Yes** if a warning dialog box appears. Click the box next to **IndCom for EtherCAT Scan Engine Support**. The required dependences are automatically checked. Click **Next** to continue installing software on the controller.

4. Once the software installation is finished, select the controller under Remote Systems in the Configuration pane. Click the **Advanced Ethernet Settings** button. In the Ethernet Devices window, select the secondary MAC address (the one that does not say primary). Under Ethernet Device Settings, select **EtherCAT** in the drop-down box under **Mode** and click **OK**.


Advanced Ethernet Settings

Ethernet Devices (by MAC Address)
00:80:2F:11:39:5E (primary)
00:80:2F:11:39:5F

Ethernet Device Settings
Mode: EtherCAT
Disabled
TCP/IP Network
Time-Triggered Network
✓ EtherCAT

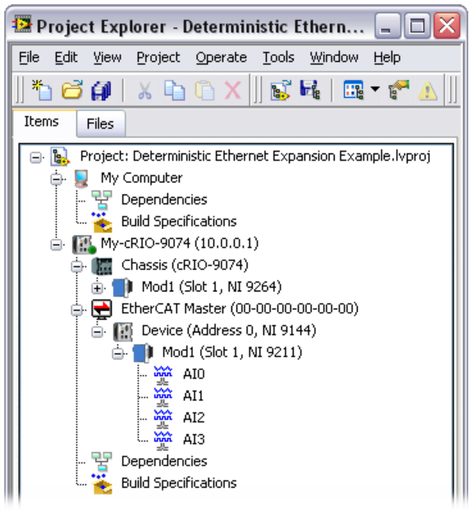
Figure 5.11. Select "EtherCAT" as the mode for the second Ethernet port of the CompactRIO controller.

Step 2. Add Deterministic I/O to the IO Scan

1. Connect the host computer and Port 1 of the CompactRIO controller to the same Ethernet network. Use a standard Ethernet cable to directly connect Port 2 of the CompactRIO controller to the IN port of the NI 9144 expansion chassis. To add more NI 9144 chassis, use the Ethernet cable to connect the OUT port of the previous NI 9144 to the IN port of the next NI 9144.

2. In the LabVIEW Project Explorer window, right-click on the CompactRIO controller and select **New»Targets and Devices.**

3. In the Add Targets and Devices dialog window, expand the category **EtherCAT Master Device** to autodiscover the EtherCAT port on the master controller. In the Scan Slaves dialog window that appears, choose the first option to autodiscover any slave devices connected to the controller. Click **OK**. The LabVIEW Project now lists the master controller, the NI 9144 chassis, their I/O modules, and the physical I/O on each module.


Project Explorer - Deterministic Ethern...

```
Project: Deterministic Ethernet Expansion Example.lvproj
  My Computer
    Dependencies
    Build Specifications
  My-cRIO-9074 (10.0.0.1)
    Chassis (cRIO-9074)
      Mod1 (Slot 1, NI 9264)
    EtherCAT Master (00-00-00-00-00-00)
      Device (Address 0, NI 9144)
        Mod1 (Slot 1, NI 9211)
          AI0
          AI1
          AI2
          AI3
    Dependencies
    Build Specifications
```

Figure 5.12. The LabVIEW Project lists the master controller, NI 9144 chassis, and the I/O modules.

4. Finally create I/O aliases that correspond to the physical channels and use these to access the expansion IO. The scan engine automatically manages I/O synchronization such that all modules are read and updated at the same time once each scan cycle.

Tip: For high-channel-count systems, you can streamline the creation of multiple I/O aliases by exporting and importing .csv spreadsheet files using the Multiple Variable Editor.

Machine Vision/Inspection

Machine vision is a combination of image acquisition from one or more industrial cameras and the processing of the images acquired. These images are usually processed using a library of image processing functions that range from simply detecting the edge of an object to reading various types of text or complex codes.

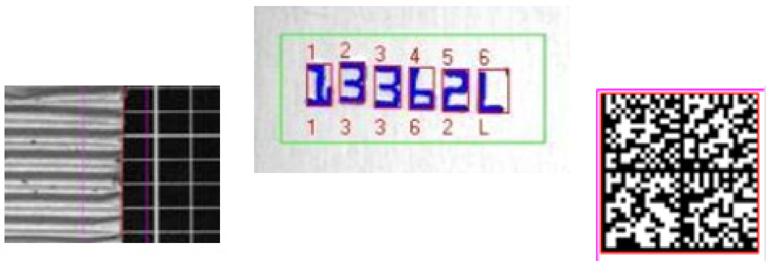


Figure 5.13. Edge detection, optical character recognition, and 2D code reading are common machine vision tasks.

Oftentimes, more than one of these measurements is made on one vision system from one or more images. You can use this for many applications including verifying that the contents of a container match the text on the front of the bottle or ensuring that a code has printed in the right place on a sticker.

The information from these processed images is fed into the control system for data logging, defect detection, motion guidance, process control, and so on.

For information on the algorithms available in NI vision tools, see the Vision Concepts Manual .

Machine Vision System Architecture

Typical machine vision systems consist of an industrial camera that connects to a real-time vision system, usually via a standardized camera bus such as IEEE 1394, gigabit Ethernet, or Camera Link. The real-time system processes the images and has I/O available for communicating to the control system.

A few companies have combined the camera with the vision system, creating something called a smart camera. Smart cameras are industrial cameras that have onboard image processing and typically include some basic I/O.

National Instruments offers both types of embedded machine vision systems. The NI Compact Vision System (Figure 5.14) is a real-time embedded vision system that features direct connectivity to up to three IEEE 1394 cameras as well as 29 general-use I/Os for synchronization and triggering. This system features Ethernet connectivity to your industrial network, allowing communication back to NI CompactRIO hardware.

NI Smart Cameras (Figure 5.14) are industrial image sensors combined with programmable processors to create rugged, all-in-one solutions for machine vision applications. These cameras have a VGA (640x480 pixels) or SXGA (1280x1024 pixels) resolution as well as an option to include a digital signal processor (DSP) for added performance for specific algorithms. These cameras feature dual gigabit Ethernet ports, digital inputs and outputs, and a built-in lighting controller.

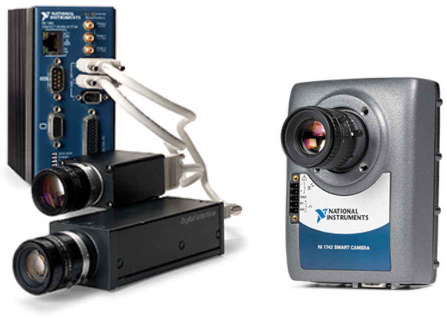


Figure 5.14. NI Compact Vision System and NI Smart Camera

National Instruments also offers plug-in image acquisition boards called frame grabbers that provide connectivity between industrial cameras and PCI, PCI Express, PXI, and PXI Express slots. These boards are commonly used for scientific and automated test applications, but you also can use them to prototype a vision application on a PC before you purchase any industrial vision systems or smart cameras.

All NI image acquisition hardware uses the same driver software called NI Vision Acquisition software. With this software, you can design and prototype your application on the hardware platform of your choice and then deploy to the industrial platform that best suits your application requirements with minimal code changes.

Lighting and Optics

This primer does not cover lighting and optics in depth, but they are crucial to the success of your application. These documents cover the majority of the basic and some more advanced machine vision lighting concepts:

- A Practical Guide to Machine Vision Lighting – Part I
- A Practical Guide to Machine Vision Lighting – Part II
- A Practical Guide to Machine Vision Lighting – Part III

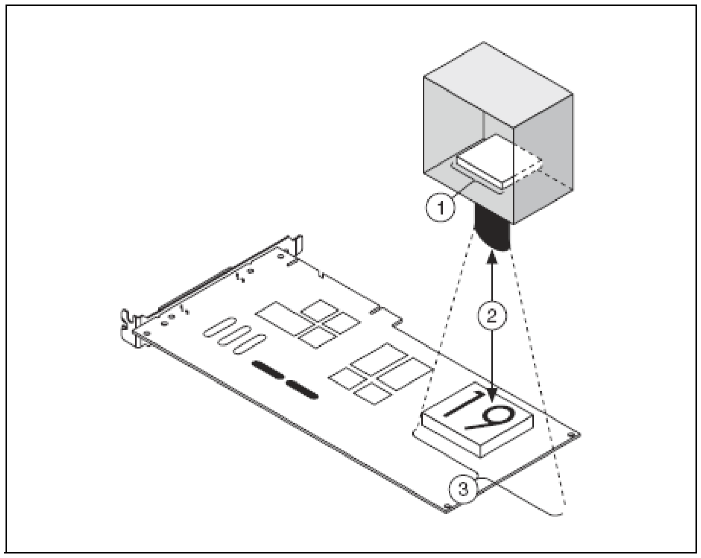
The lens used in a machine vision application changes the field of view. The field of view is the area under inspection that is imaged by the camera. It is critical to ensure that the field of view of your system includes the object you want to inspect. To calculate the horizontal and vertical field of view (FOV) of your imaging system, use Equation 1 and the specifications for the image sensor of your camera.

$$FOV = \frac{Pixel\ Pitch \times Active\ Pixels \times Working\ Distance}{Focal\ Length}$$

Equation 1, Field of View Calculation

Where:

- *FOV is the field of view in either the horizontal or vertical direction,*
- *Pixel Pitch measures the distance between the centers of adjacent pixels in either the horizontal or vertical direction,*
- *Active Pixels is the number of pixels in either the horizontal or vertical direction,*
- *Working Distance is the distance from the front element (external glass) of the lens to the object under inspection,*
- *Focal Length measures how strongly a lens converges (focuses) or diverges (diffuses) light.*



1 Horizontal Imaging Width	3 Horizontal Field of View
2 Working Distance	

Figure 5.15. Lens selection determines the field of view.

For example, if the working distance of your imaging setup is 100 mm, and the focal length of the lens is 8 mm, then the field of view in the horizontal direction of an NI Smart Camera using the VGA sensor in full scan mode is:

$$FOV_{horizontal} = \frac{0.0074\text{ mm} \times 640 \times 100\text{ mm}}{8\text{ mm}} = 59.2\text{ mm}$$

Similarly, the field of view in the vertical direction is:

$$FOV_{vertical} = \frac{0.0074\text{ mm} \times 480 \times 100\text{ mm}}{8\text{ mm}} = 44.4\text{ mm}$$

Based on the result, you can see that you might need to adjust the various parameters in the FOV equation until you achieve the right combination of components that match your inspection needs. This might include increasing your working distance, choosing a lens with a shorter focal length, or changing to a high resolution camera.

Software Options

Once you have chosen the hardware platform for your machine vision project, you need to select the software platform you want to use. National Instruments offers two application development environments (ADEs) for machine vision. Both NI Compact Vision Systems and NI Smart Cameras are LabVIEW Real-Time targets, so you can develop your machine vision application using the LabVIEW Real-Time Module and the NI Vision Development Module.

The NI Vision Development Module is a library of machine vision functions that range from basic filtering to pattern matching and optical character recognition. This library also includes the NI Vision Assistant and the Vision Assistant Express VI. The Vision Assistant is a rapid prototyping tool for machine vision applications. With this tool, you can use click-and-drag, configurable menus to set up most of your application. With the Vision Assistant Express VI, you can use this same prototyping tool directly within LabVIEW Real-Time.

Another software platform option is NI Vision Builder for Automated Inspection (AI). Vision Builder AI is a configurable machine vision ADE based on a state diagram model, so looping and decision making are extremely simple. Vision Builder AI features many of the high-level tools found in the Vision Development Module. Both hardware targets work with Vision Builder AI as well, giving you the flexibility to choose the software you are most comfortable with and the hardware that best suits your application.

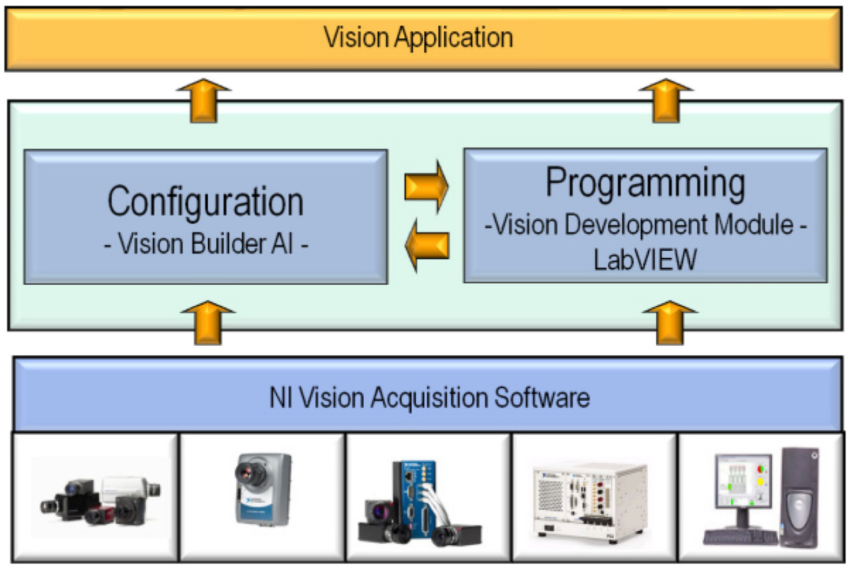


Figure 5.16. National Instruments offers both configuration software and a full programming environment for machine vision application development.

Machine Vision/Control System Interface

Most smart camera or embedded vision system applications provide real-time inline processing and give outputs that you can use as another input into the control system. The control system usually has control over when this image acquisition and processing starts via sending a trigger to the vision system. The trigger can also come from hardware sensors such as proximity sensors or quadrature encoders.

The image is processed into a set of usable results such as the position of a box on a conveyor or the value and quality of a 2D code printed on an automotive part. These results are reported back to the control system and/or sent across the industrial network for logging. You can choose from several methods to report these results, from a simple digital I/O to shared variables or direct TCP/IP communication, as discussed previously in this document.

Machine Vision Using LabVIEW Real-Time

The following example demonstrates the development of a machine vision application for an NI Smart Camera using LabVIEW Real-Time and the Vision Development Module.

Step 1. Add an NI Smart Camera to the LabVIEW Project

You can add the NI Smart Camera to the same LabVIEW Project as the CompactRIO system. If you wish to prototype without the smart camera connected, you can also simulate a camera.

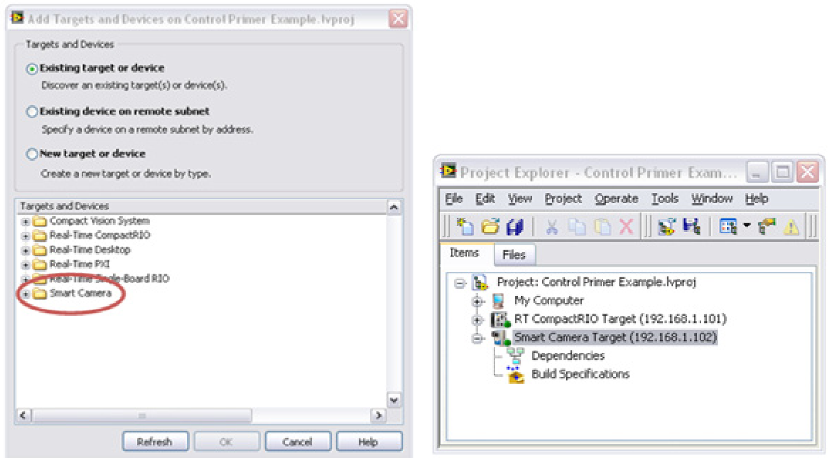

Add Targets and Devices on Control Primer Example.lvproj
Targets and Devices
Existing target or device
Discover an existing target(s) or device(s).
Existing device on remote subnet
Specify a device on a remote subnet by address.
New target or device
Create a new target or device by type.
Targets and Devices
Compact Vision System
Real-Time CompactRIO
Real-Time Desktop
Real-Time PXI
Real-Time Single-Board RIO
Smart Camera
Refresh
OK
Cancel
Help
Project Explorer - Control Primer Exam...
File Edit View Project Operate Tools Window Help
Items
Files
Project: Control Primer Example.lvproj
My Computer
RT CompactRIO Target (192.168.1.101)
Smart Camera Target (192.168.1.102)
Dependencies
Build Specifications

Figure 5.17. You can add NI Smart Camera systems to the same LabVIEW Project as CompactRIO systems.

Step 2. Use LabVIEW to Program the NI Smart Camera

Creating an application for the smart camera is almost identical to creating an application for a CompactRIO real-time controller. The main difference is the use of the NI Vision Acquisition drivers to acquire your images and algorithms in the Vision Development Module in order to process them.

You can create a new VI and targeted to the smart camera just as you have created VIs for CompactRIO.

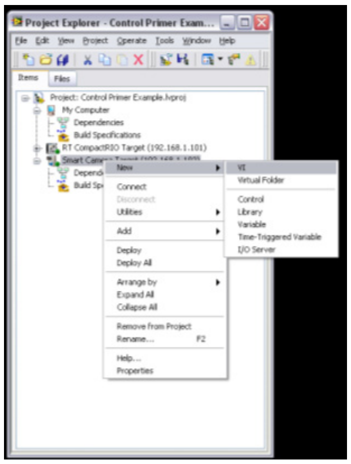


Figure 5.18. Adding a VI on Your NI Smart Camera to Your Project in LabVIEW Real-Time

Upon deployment, this VI resides on the storage on the smart camera and runs on the smart camera during run time.

To simplify the process of acquiring and processing images, National Instruments includes Express VIs in the Vision Palette. Use these Express VIs in this example to acquire images from the smart camera (or vision system, if that's what you're using) as well as process the images. To access these Express VIs, right-click on the block diagram and choose **Vision»Vision Express.**

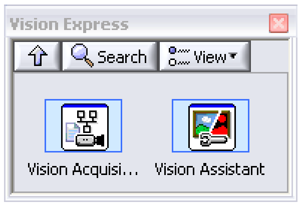
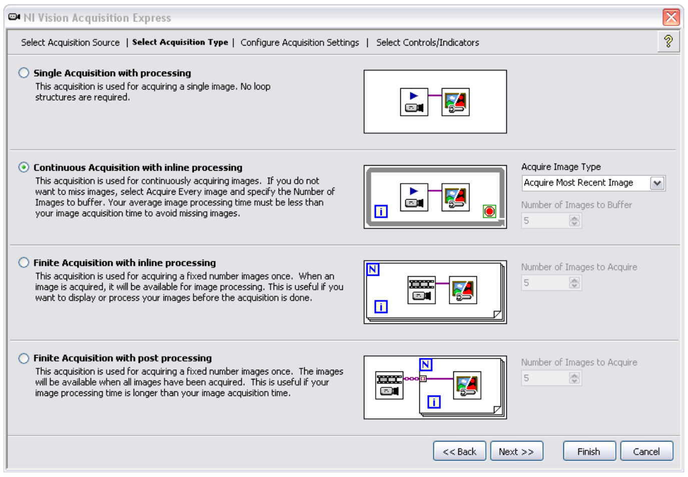


Figure 5.19. The Vision Express Palette

The first step is to set up an acquisition from your camera. Or, if your camera is not available or not fixtured correctly yet, you also can set up a simulated acquisition by opening images stored on your hard drive.

Start by dropping the Vision Acquisition Express VI onto the block diagram. This menu-driven interface is designed so that you can quickly acquire your first image. If you have any recognized image acquisition hardware connected to your computer, it shows up as an option. If not, you have the option on the first menu to open images from disk.


NI Vision Acquisition Express

Select Acquisition Source | **Select Acquisition Type** | Configure Acquisition Settings | Select Controls/Indicators

( ) **Single Acquisition with processing**
This acquisition is used for acquiring a single image. No loop structures are required.

(✓) **Continuous Acquisition with inline processing**
This acquisition is used for continuously acquiring images. If you do not want to miss images, select Acquire Every image and specify the Number of Images to buffer. Your average image processing time must be less than your image acquisition time to avoid missing images.

Acquire Image Type: Acquire Most Recent Image
Number of Images to Buffer: 5

( ) **Finite Acquisition with inline processing**
This acquisition is used for acquiring a fixed number images once. When an image is acquired, it will be available for image processing. This is useful if you want to display or process your images before the acquisition is done.

Number of Images to Acquire: 5

( ) **Finite Acquisition with post processing**
This acquisition is used for acquiring a fixed number images once. The images will be available when all images have been acquired. This is useful if your image processing time is longer than your image acquisition time.

Number of Images to Acquire: 5

Figure 5.20. The Vision Acquisition Express VI guides you though creating a vision application in LabVIEW.

Next, choose which type of acquisition you are implementing. For this example, select **Continuous Acquisition with inline processing**. This allows you to sit in a loop and process images. Next, test your input source and verify that it looks right. If it does, then click the finish button at the bottom.

Once this Express VI generates the LabVIEW code behind the scenes, the block diagram appears again. Now drop the Vision Assistant Express VI just to the right of the Vision Assistant Express VI.

With the Vision Assistant, you can prototype your vision processing quickly. You can deploy this same tool to real-time systems, although traditional LabVIEW VIs are usually implemented to provide greater efficiency in the real-time systems. For an overview of the tools available in this assistant, view the NI Vision Assistant Tutorial.

Step 3. Communicate with the CompactRIO System

Once you have set up the machine vision you plan to conduct with the Vision Assistant Express VI, the last thing to do is communicate the data with the CompactRIO. You can use network-published shared variables to pass data between the two systems. Details on network communication between LabVIEW systems are covered in depth in an earlier section in this document. In this example, you are examining a battery clamp, and you want to return the condition of the holes (if they are drilled correctly) and the current gap shown in the clamp.

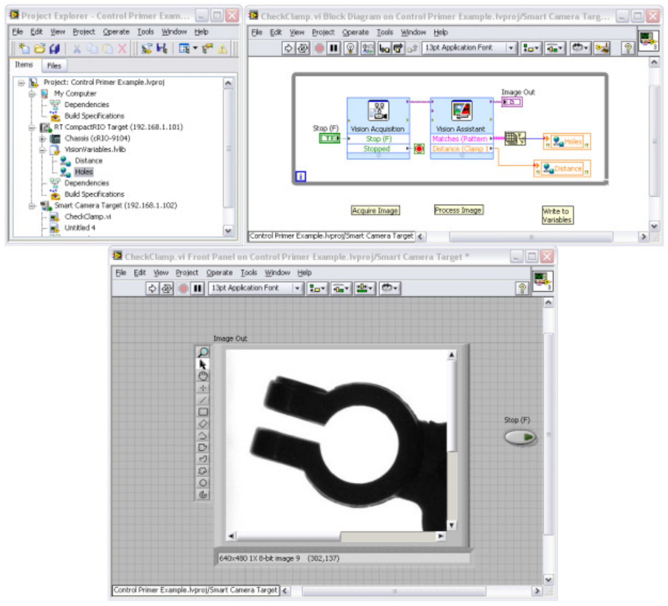

Project Explorer - Control Primer Example
CheckClamp.vi Block Diagram on Control Primer Example.lvproj/Smart Camera Target
Vision Acquisition
Vision Assistant
Image Out
Stop (F)
Acquire Image
Process Image
Write to Variables
CheckClamp.vi Front Panel on Control Primer Example.lvproj/Smart Camera Target

Figure 5.21. A complete inspection in LabVIEW.

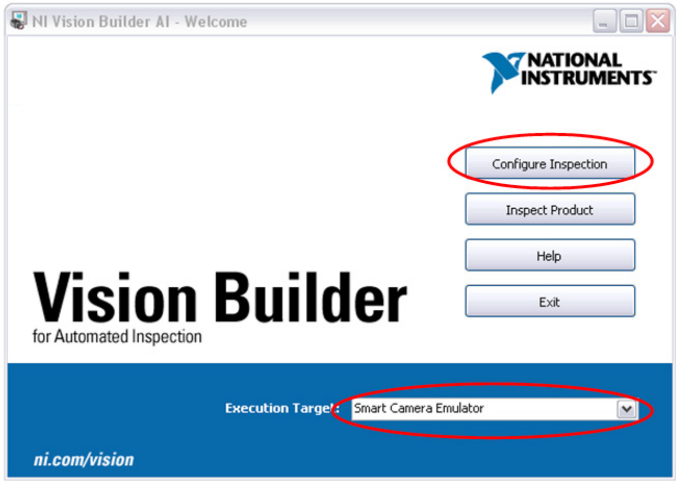
As you can see in Figure 5.21, the results of the inspection are passed as current values to the CompactRIO via shared variables hosted on the CompactRIO. You can also pass the data as commands to the CompactRIO.

Machine Vision Using Vision Builder AI

As mentioned above, Vision Builder AI is a configurable environment for machine vision. You implement all of the image acquisition, image processing, and data handling through configurable menus.

Step 1. Configure an NI Smart Camera with Vision Builder AI

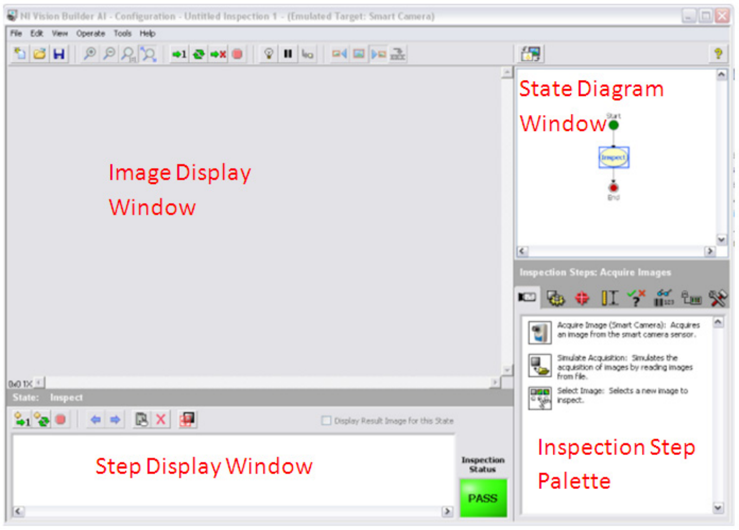
When you open the environment, you see a splash screen where you can choose your execution target. If you do not have a smart camera connected, you can simulate a smart camera. Choose this option for this example.



Execution Target: Smart Camera Emulator

Figure 5.22. Choose your execution target from the Vision Builder AI splash screen.

With this emulator, you can set up lighting and trigger options, arrange I/O to communicate to the CompactRIO hardware, and complete many of the other actions required to configure the smart camera without having one available for your development system. Once you have selected your execution target, click on **Configure Inspection**. This takes you into the development environment, which features four main windows. Use the largest window, the **Image Display Window**, to see the image you have acquired as well as any overlays you have placed on the image. Use the window to the upper right, the **State Diagram Window**, to show the states of your inspection (with your current state highlighted). The bottom right window, the **Inspection Step Palette**, displays all the inspection steps for your application. Lastly, the thin bar at the bottom is the **Step Display Window**, where you see all of the steps that are in the current state.


NI Vision Builder AI - Configuration - Untitled Inspection 1 - (Emulated Target: Smart Camera)
File Edit View Operate Tools Help
State Diagram Window
Start
Inspect
End
Image Display Window
Inspection Steps: Acquire Images
State: Inspect
Display Result Image for this State
Step Display Window
Inspection Status
PASS
Inspection Step Palette

Acquire Image (Smart Camera): Acquires an image from the smart camera sensor.

Simulate Acquisition: Simulates the acquisition of images by reading images from file.

Select Image: Selects a new image to inspect.

Figure 5.23. The Vision Builder AI Development Environment

This example implements the same inspection as the LabVIEW example, which inspected battery clamps and returned the number of holes and the gap of the clamp back to CompactRIO via the two shared variables hosted on the CompactRIO hardware.

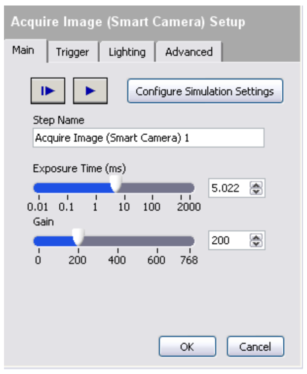
Step 2. Configure the Inspection

The first step of the inspection, just like in LabVIEW, is to acquire the image. Implement this with the **Acquire Image (Smart Camera)** step. Select this step and configure the emulation by clicking on the **Configure Simulation Settings** button. For example, set the image to grab the images from the following path:

C:\Program Files\National Instruments\Vision Builder AI 3.6\DemoImg\Battery\BAT0000.PNG

This library of images should install with every copy of Vision Builder AI (with differing version numbers). After selecting the file above, also make sure to check the box for **Cycle through folder images**.

You also see that with this window, you can configure exposure times, gains, and other settings for the camera. These settings do not make any difference in the emulator, but in the real world, they go hand-in-hand with lighting and optics choices.



Step Name
Acquire Image (Smart Camera) 1

Exposure Time (ms)
5.022

Gain
200

Figure 5.24. The Image Acquisition Setup Step

From here, set up pattern matching, edge detection, object detection, code reading, or any other algorithms you need. For example, implement a pattern match to set the overall rotation of the object, a detect objects to see if both of the holes were in the clamp, and a caliper tool to detect the distance between the clamp prongs. This generates a few pass/fail results that you can use to set the overall inspection status, which you can display on the overlay to the user.

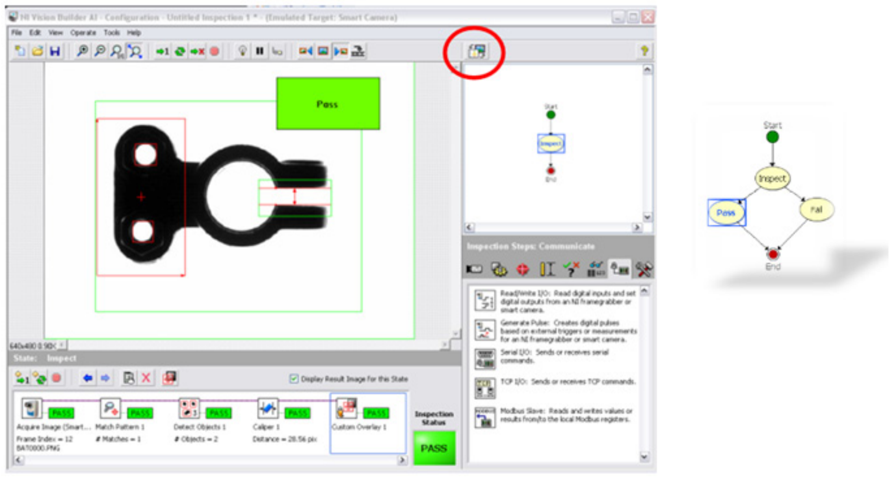


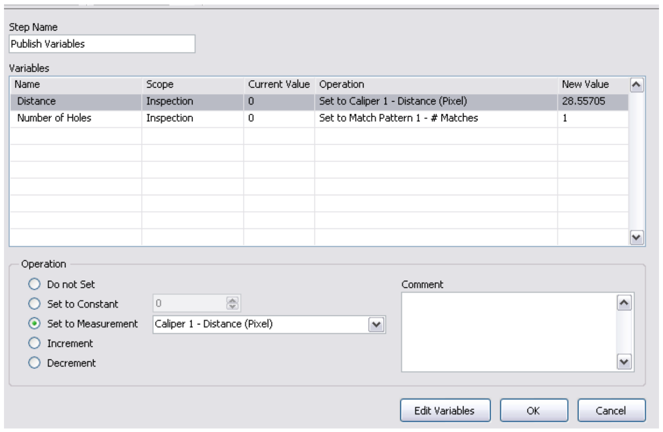
Figure 5.25. A Completed Inspection in Vision Builder AI

Now create two new states by clicking the toggle main window view button (circled in red). Create a pass state and a fail state. In both states, report the values back to CompactRIO but, in the fail state, also reject the part using some digital I/O.

Step 3. Communicate with the CompactRIO System

Vision Builder AI provides access to many of the I/O types that have been discussed previously, including network-published shared vVariables, RS232, Modbus, Modbus/TCP, and raw TCP/IP. In this example, use the variable manager to access the shared variables hosted on CompactRIO. Access the variable manager by navigating to **Tools»Variable Manager**.

Here you can discover the variables on CompactRIO by navigating to the **Network Variables** tab. From there, select and add the variables and bind them to variables within the inspection.



Step Name
Publish Variables

Variables

Name	Scope	Current Value	Operation	New Value
Distance	Inspection	0	Set to Caliper 1 - Distance (Pixel)	28.55705
Number of Holes	Inspection	0	Set to Match Pattern 1 - # Matches	1

Operation
- Do not Set
- Set to Constant 0
- [✓ Set to Measurement] Caliper 1 - Distance (Pixel)
- Increment
- Decrement

Comment

Edit Variables | OK | Cancel

Figure 5.26. Setting Up Variable Communication in Vision Builder AI

Now these results are sent back to CompactRIO, and the inspection waits on the next camera trigger.

As you can see, both methods (programmable and configurable) offer the user a way to acquire images, process them, and then use the pertinent information to report results back to a CompactRIO control system or to directly control I/O from a real-time vision system.

CHAPTER 6
Customizing the Hardware through LabVIEW FPGA

Extending CompactRIO through LabVIEW FPGA

So far, the control architecture discussed in this document has focused on using the onboard FPGA within CompactRIO with a fixed personality running the CompactRIO Scan Mode. This section demonstrates when, why, and how to extend the capabilities of the FPGA. The scan mode can support most NI C Series I/O modules for CompactRIO and multiple higher-level functions, such as quadrature-encoder measurements and PWM generation, but the FPGA is capable of much more when customized with LabVIEW FPGA.

With LabVIEW FPGA you can:

- Collect analog waveform at rates of hundreds of kilohertz
- Create custom digital pulse trains at up to 40 MHz
- Implement custom digital communication protocols
- Run control loops at rates in the hundreds of kilohertz
- Use modules not supported by the scan mode including CAN and PROFIBUS communication
- Implement custom timing, triggering, and filtering

Because the CompactRIO FPGA is completely customizable, you can run both the scan mode and custom LabVIEW FPGA code side by side at the same time.

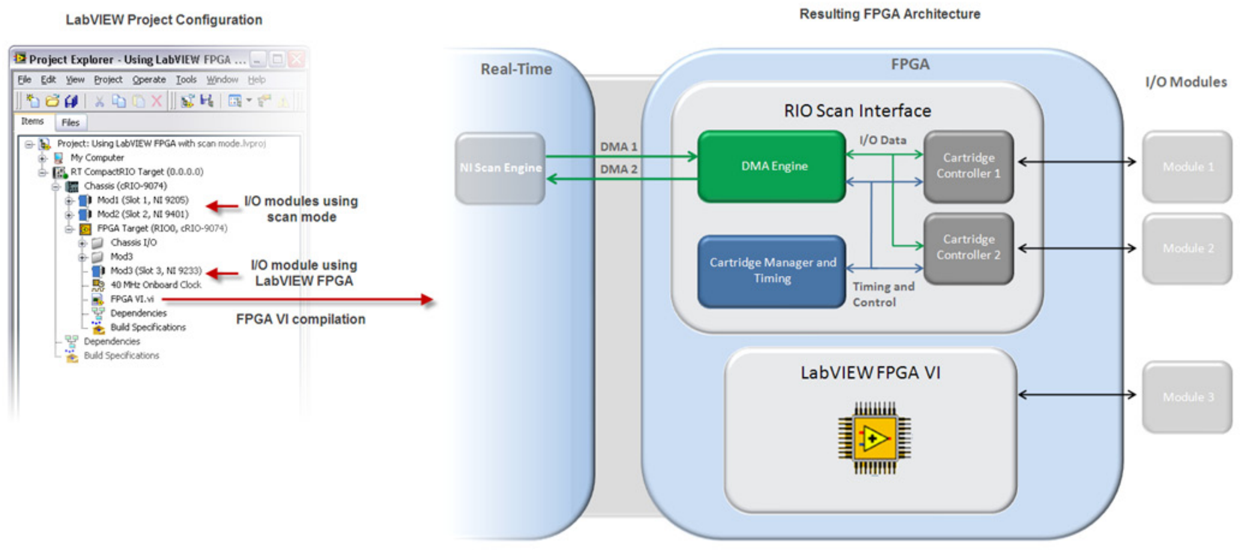


Figure 6.1. You can supplement the capabilities of the scan mode with custom LabVIEW FPGA code.

When to Use LabVIEW FPGA

You should supplement the scan mode using the LabVIEW FPGA Module for several reasons. With the combination of a real-time processor and programmable FPGA on CompactRIO, you can create a system to take advantage of the strengths of each computing platform. The real-time processor excels at floating-point math and analysis and peripheral communication such as network-published shared variables and Web services. The FPGA excels at smaller tasks that require very high-speed logic and precise timing. Below is a list of scenarios for which you program the FPGA directly.

High-Speed Waveform Acquisition /Generation (greater than 1 kHz)

The CompactRIO Scan Mode is optimized for control loops running at less than 1 kHz, but many C Series I/O modules are capable of acquiring and generating at much higher rates. If you need to take full advantage of these module features and acquire or generate at speeds higher than 1 kHz, you can use LabVIEW FPGA to acquire at a user-defined rate tailored to your application.

Custom Triggering/Timing/Synchronization

With the reconfigurable FPGA, you can create simple, advanced, or otherwise custom implementations of triggers, timing schemes, and I/O or chassis synchronization. These can be as elaborate as triggering a custom CAN message based on the rise of an analog acquisition exceeding a threshold or as simple as acquiring input values on the rising edge of an external clock source.

Hardware-Based Analysis/Generation and Coprocessing

Many sensors output more data than can be reasonably processed on the real-time processor alone. You can use the FPGA as a valuable coprocessor to analyze or generate complex signals while freeing the processor for other critical threads. This type of FPGA based coprocessing is commonly used in applications such as:

- Encoding/decoding sensors
  – Tachometers
  – Standard and/or custom digital protocols
- Signal processing and analysis
  – Spectral analysis (fast Fourier transforms and windowing)
  – Filtering, averaging, and so on
  – Data reduction
  – Third-party IP integration
- Sensor simulation
  – Cam and crank
  – Linear-voltage differential transformers (LVDTs)
- Hardware-in-the-loop simulation

Highest-Performance Control

Not only can you use the FPGA for high-speed acquisition and generation, but you also can implement many control algorithms on the FPGA. You can use single-point I/O with multichannel, tunable PID or other control algorithms to implement deterministic control with loop rates up to hundreds of kilohertz.

Unsupported Modules

Several C Series modules do not feature scan mode support. For these modules, you need to use LabVIEW FPGA to build an interface between the I/O and your real-time application. For a list of modules that feature scan mode support, see C Series Modules Supported by CompactRIO Scan Mode.

Unsupported Targets
CompactRIO targets with 1M gate FPGAs cannot fully support the scan mode. You can implement some scan mode features on unsupported targets, but you must use LabVIEW FPGA. The KnowledgeBase article "Using CompactRIO Scan Mode with Unsupported Backplanes" describes how to use LabVIEW FPGA to build a custom scan mode interface for an unsupported FPGA target.

FPGA Overview

A field-programmable gate array, or FPGA, is a programmable chip composed of three basic components: logic blocks, programmable interconnects, and I/O blocks.

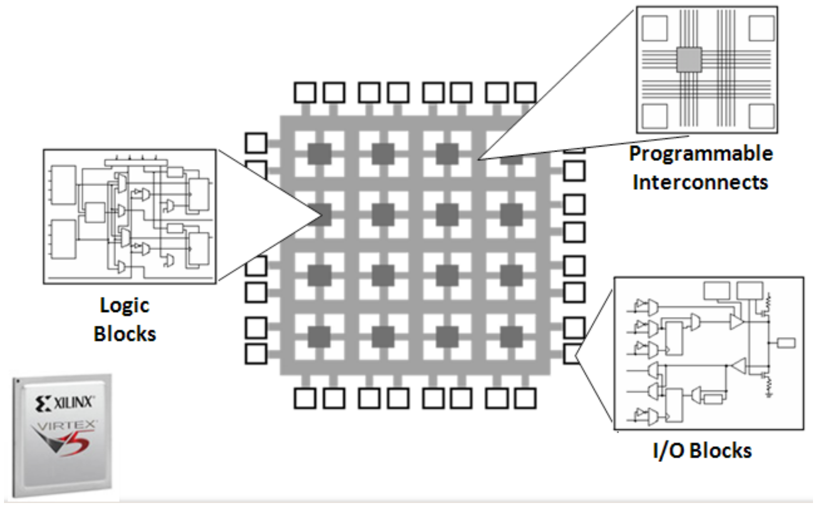
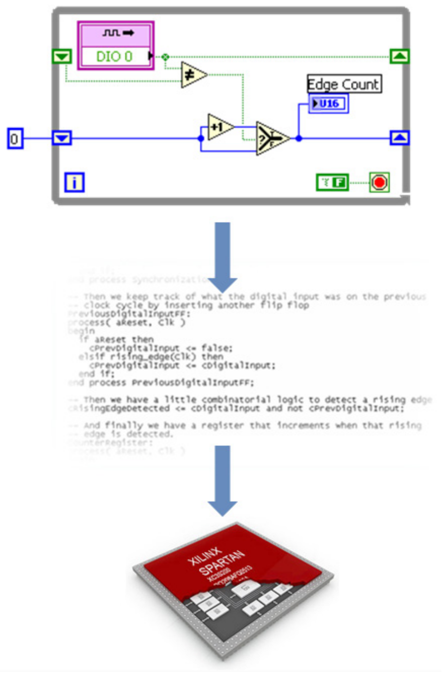


Figure 6.2. An FPGA is composed of configurable logic and I/O blocks tied together with programmable interconnects.

The logic blocks are a collection of digital components such as lookup tables, multipliers, and multiplexers where bits are crunched and processed to produce programmatic results. These logic blocks are connected with programmable interconnects that serve as a sort of micro-breadboard to route signals from one logic block to the next. The programmable interconnect can also route signals to the I/O blocks that are connected to the pins on the chip for two-way communication to surrounding circuitry. FPGAs are really just a blank silicon canvas that you can program to be any type of custom digital hardware. Traditionally, programming these FPGA chips has been difficult and, therefore, only possible by experienced digital designers and hardware engineers. National Instruments has abstracted programming these devices through graphical system design with LabVIEW FPGA so that nearly anyone can take advantage of these powerful reconfigurable chips.

Because LabVIEW FPGA VIs are synthesized down to physical hardware gates connected by programmable interconnects, the FPGA compile process is different from the compile process for a traditional LabVIEW for Windows or LabVIEW Real-Time application. When writing code for the FPGA, you write the same LabVIEW code as you do for any other target, but when you select the Run button, LabVIEW internally goes through a different process. First, LabVIEW FPGA generates VHDL code and passes it to the Xilinx compiler. Then the Xilinx compiler synthesizes the VHDL and places and routes all synthesized components into a bitfile. Finally, the bitfile is downloaded to the FPGA and the FPGA assumes the personality you have programmed. This process, which is more complicated than other LabVIEW compiles, can take five to 10 minutes to complete or up to several hours for complicated designs. Because of the relatively long compile times, you should spend more time debugging and planning your LabVIEW FPGA code before attempting to compile.



```
  end if;
end process Synchronization

-- Then we keep track of what the digital input was on the previous
-- clock cycle by inserting another flip flop
PreviousDigitalInputFF:
process( aReset, Clk )
begin
  if aReset then
    cPrevDigitalInput <= false;
  elsif rising_edge(Clk) then
    cPrevDigitalInput <= cDigitalInput;
  end if;
end process PreviousDigitalInputFF;

-- Then we have a little combinatorial logic to detect a rising edge
cRisingEdgeDetected <= cDigitalInput and not cPrevDigitalInput;

-- And finally we have a register that increments when that rising
-- edge is detected.
CounterRegister:
process( aReset, Clk )
```

Figure 6.3. Under the hood, the LabVIEW FPGA compiler translates LabVIEW code into VHDL and invokes the Xilinx compile tools to generate a bitfile, which runs directly on the FPGA.

Benefits of FPGAs

Because your LabVIEW FPGA code runs directly on hardware, you can take advantage of the following FPGA-based design benefits:

High Reliability

LabVIEW FPGA code running on a FPGA is highly reliable because the logic is compiled into a physical hardware design. Once you program the FPGA, it becomes a hardware chip with all of the associated reliability.

High Determinism

Processor-based systems often involve several abstraction layers to help schedule tasks and share resources among multiple processes. The driver layer controls hardware resources and the operating system manages memory and processor bandwidth. For any given processor core, only one instruction can execute at a time. Real-time operating systems reduce jitter to a finite maximum when programmed with good priority hierarchy. FPGAs do not use any operating systems. This minimizes reliability concerns with true parallel execution and deterministic hardware dedicated to every task. For NI FPGA-based hardware, you can achieve 25 ns timing accuracy of critical components within your design.

True Parallelism

Multithreaded applications break down into multiple parallel sections of code which are executed in a round-robin fashion, giving the appearance of parallel execution. Multicore processors expand on that idea by allowing multithreaded applications to truly execute multiple parallel code at one time. The number of parallel pieces of code executing concurrently is limited to the number of cores available in the specific processor. Because an FPGA implements parallel code as parallel circuits in hardware, you are not limited by processor cores; therefore, every piece of parallel code in an entire FPGA application can execute concurrently. Even traditionally serial operations can improve throughput on FPGAs by implementing pipelining.

Reconfigurability
Being reconfigurable, FPGA chips are able to keep up with any future modifications you might need. As a product or system matures, you can make functional enhancements without spending time redesigning hardware or modifying a board layout. This is especially applicable to industrial communication protocols. As communication protocols evolve and improve over time, you can modify the implementation of that protocol within an FPGA to support the latest technology features and changes.

Instant Boot Up
Because LabVIEW FPGA code runs directly on the FPGA without an overarching operating system, the code you download to the FPGA flash memory begins running within milliseconds of powering on your CompactRIO chassis. This code can begin executing a control loop or setting startup output values.

Programming with LabVIEW FPGA

As you explore the need to extend the control architecture on NI CompactRIO hardware to include custom FPGA programming, consider how to include this in your system. In reality, CompactRIO hardware has three programming modes: scan mode, pure FPGA interface mode, and hybrid mode. The vast majority of this control architecture document has focused on using the CompactRIO Scan Mode to retrieve I/O. You have learned that the enabling technology for this programming mode is a fixed FPGA personality that marshals the data through the FPGA and up to the real-time host. This section examines how to use the FPGA in hybrid mode to access I/O directly with FPGA programmability while keeping other I/O under the scan mode. With hybrid mode, you can use the FPGA programming model to handle waveform buffered acquisition, inline processing, and certain modules that do not feature scan mode support.

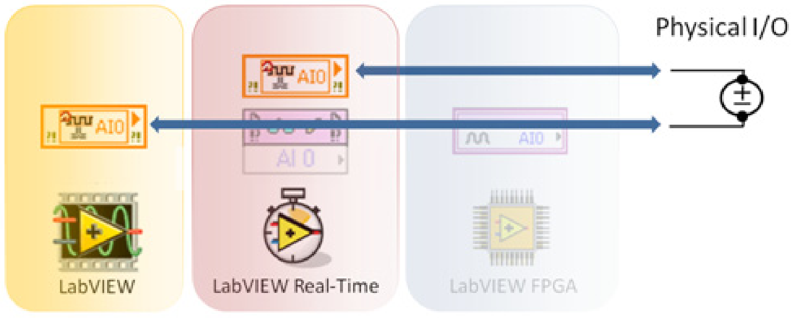


Figure 6.4. In hybrid mode, you can choose to acquire physical I/O through the scan mode or the FPGA interface.

Hybrid Mode on CompactRIO

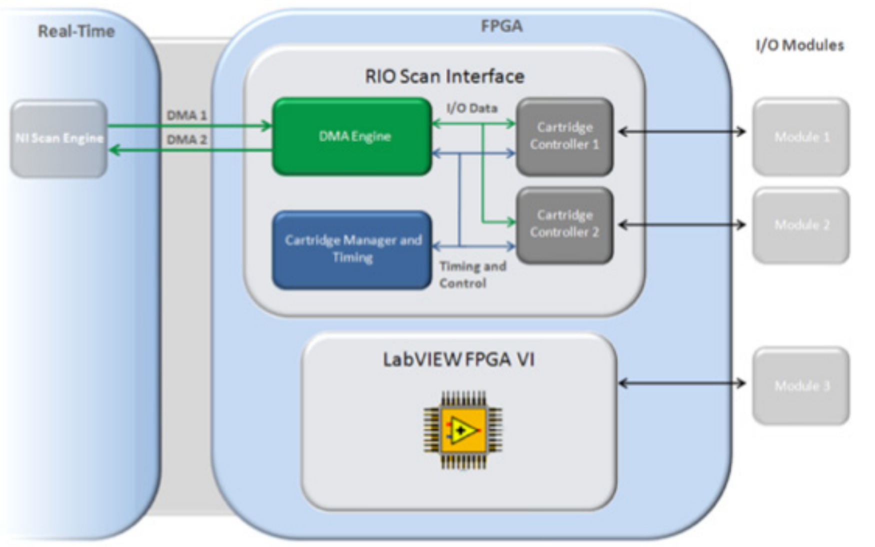
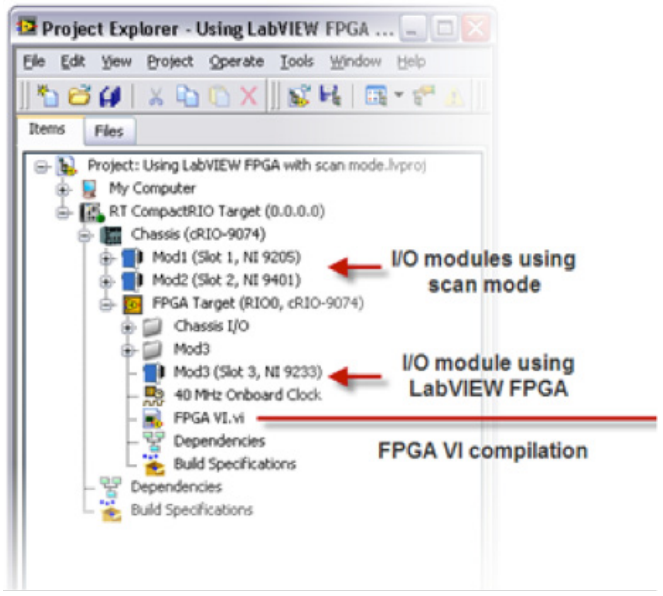


Figure 6.5. After activating hybrid mode, write an FPGA VI to interface with the module and pass data to the real-time host.

In hybrid mode, you can continue using the NI RIO Scan Interface on some modules while programming others directly on the FPGA. Activate FPGA programming for a particular module by dragging and dropping the module project item from under the CompactRIO chassis to under the FPGA target. By doing this, you can program the FPGA for custom code running in parallel to the scan interface for other modules.


Project Explorer - Using LabVIEW FPGA ...
File Edit View Project Operate Tools Window Help
Items
Files
I/O modules using scan mode
I/O module using LabVIEW FPGA
FPGA VI compilation

```
Project: Using LabVIEW FPGA with scan mode.lvproj
    My Computer
    RT CompactRIO Target (0.0.0.0)
        Chassis (cRIO-9074)
            Mod1 (Slot 1, NI 9205)
            Mod2 (Slot 2, NI 9401)
            FPGA Target (RIO0, cRIO-9074)
                Chassis I/O
                Mod3
                Mod3 (Slot 3, NI 9233)
                40 MHz Onboard Clock
                FPGA VI.vi
                Dependencies
                Build Specifications
        Dependencies
        Build Specifications
```

Figure 6.6. This project in hybrid mode shows some modules under the CompactRIO chassis and others (module 3 in this figure) under the FPGA.

Using hybrid mode requires a working knowledge of FPGA programming and using the FPGA interface on the real-time side because typical applications that require hybrid mode are intrinsically more complex.

Waveform Acquisition Example

Scan mode is a single-point acquisition method that closes the loop at rates up to 1 kHz. However, most modules can sample faster than this rate. To sample module input at rates above 1 kHz, you must complete the following steps:

1. Put the module in hybrid mode by placing it in the FPGA context within the project (discussed in the previous section)

2. Write an FPGA VI that:
   a. Reads data from an FPGA I/O node
   b. Uses LabVIEW structures and logic to define acquisition rates and triggering
   c. Packs the data into a DMA buffer for consumption on the host

3. Modify the host to read from DMA in arrays

4. Put the array data into the I/O scan (discussed in the “Adding I/O Outside the Scan Mode” section)

Reading from an I/O Node

To create a new FPGA VI, right-click on the FPGA Target in the project and create a new VI under the FPGA Target. By activating hybrid mode on a particular module, you can see a new folder under the FPGA that contains all of the I/O points for that module. Drag and drop an I/O point on the block diagram to get an I/O node.

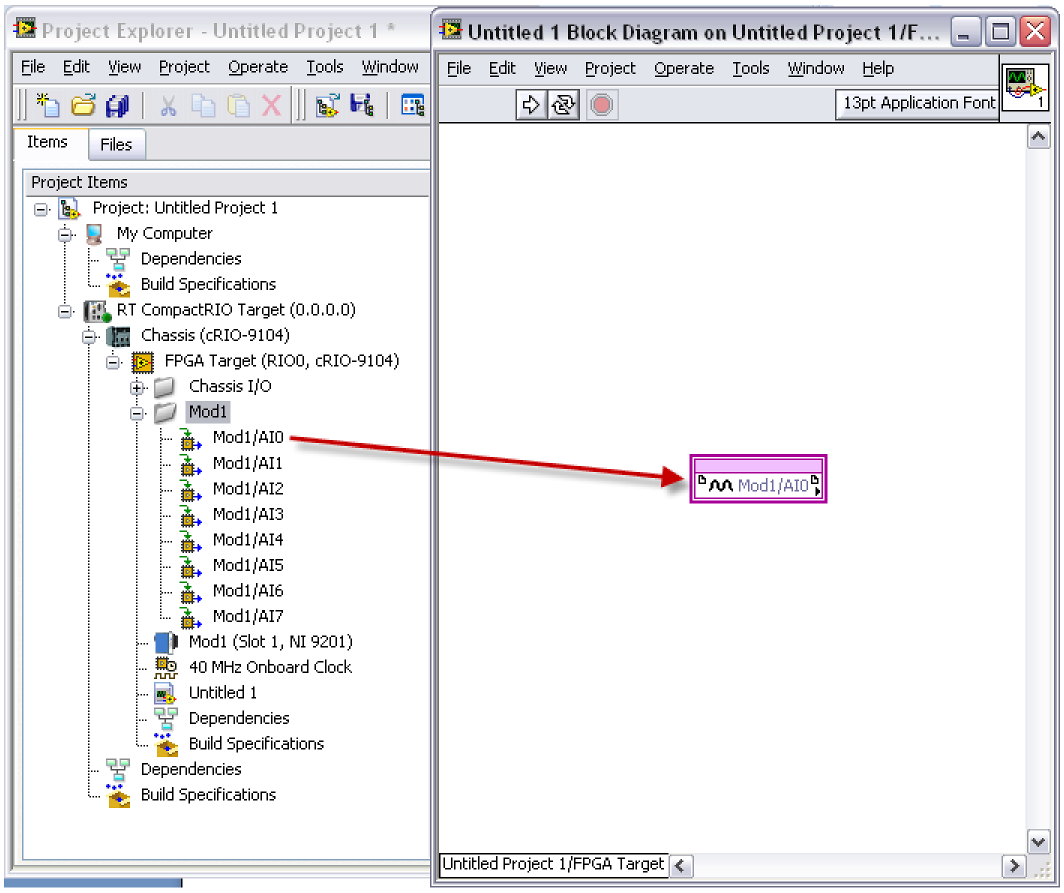


Figure 6.7. Drag and drop an I/O node to the FPGA block diagram.

Running the VI at this point starts the compile process. You probably want to use FPGA simulation during development. To switch the FPGA target to simulation, right-click on the target and select **Execute VI on...»Development Computer with Simulated I/O**.

Use LabVIEW Structures and Logic to Set Acquisition Rates and Triggering
The FPGA I/O Node returns single-point data when called. Use LabVIEW structures and logic to specify the sampling rate and triggering.

Use a loop with a loop timer to set the sampling rate. The general guideline for sampled acquisition is to put a loop timer in the first frame of a sequence structure and the FPGA I/O Node in the second with a loop around the whole thing. The sample delay in ticks sets the rate. Ticks are the pulses on the FPGA clock, which is 40 MHz by default. For example, implementing a sample delay of 20,000 ticks of the 40 MHz clock renders a sampling rate of 2 kHz. You can also specify the sample delay in microseconds or milliseconds by double-clicking the loop timer and adjusting the configuration panel. Of course, no matter what you specify, the module only samples up to the maximum rate specified in the module documentation.

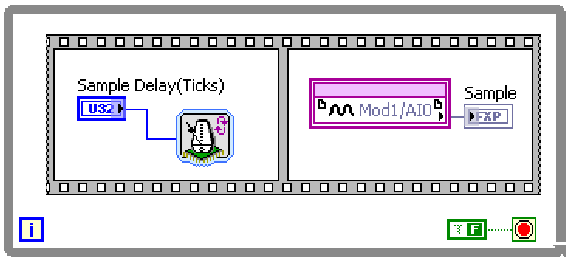


Figure 6.8. The Acquisition Scheme for a Standard Analog Input Module

To implement a triggered application, use a case structure gated on the trigger condition. This trigger condition can be an input from the host processor or it can be derived from logic directly on the FPGA. Figure 6.9 shows the simplest case of a one-shot trigger. With LabVIEW FPGA you can implement more complex triggering schemes such as retriggering, pause triggering, or any type of custom triggering.

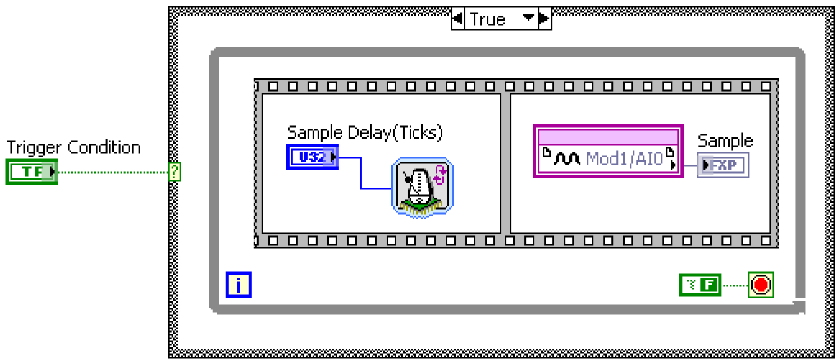


Figure 6.9. Simple One-Shot Trigger

Example – Stream Waveform Data to Real-Time Hardware from the FPGA

LabVIEW Example Code
is provided for this section

Configure Communication between the FPGA and Real-Time Hardware
You can use DM) to stream high-speed data between the FPGA and real-time hardware. To create a DMA buffer for streaming data, right-click on the FPGA target and select **New..»FIFO**.

Give the FIFO structure a descriptive name and choose target-to-host as the type. This means that data should flow through this DMA FIFO from the FPGA target to the real-time host. You can also set data type and FPGA FIFO depths. Pressing OK puts this new FIFO into your project, which you can drag and drop to the FPGA block diagram.

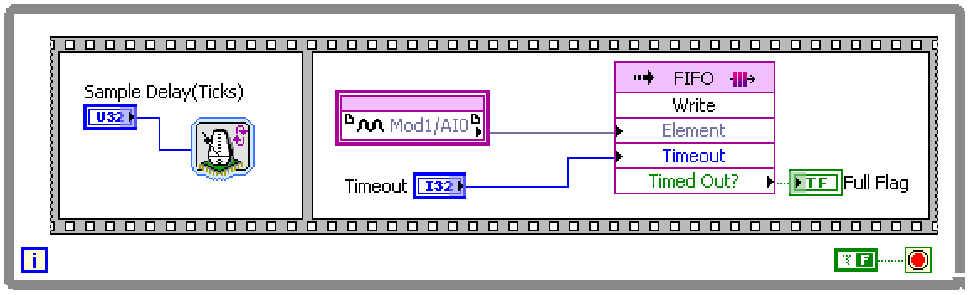


Figure 6.10. Simple DMA Transfer on One Channel

It is fairly simple to put a DMA FIFO on your diagram. However, complexities arise when the default settings on the DMA transfer are not sufficient. If missing data points create a bug in your system, you must monitor the full flag on the FPGA and latch it when a fault occurs. Simply sampling this register from the host is not sufficient to catch quick transitions on that variable. Figure 6.11 shows various latching techniques on the timeout (full flag).

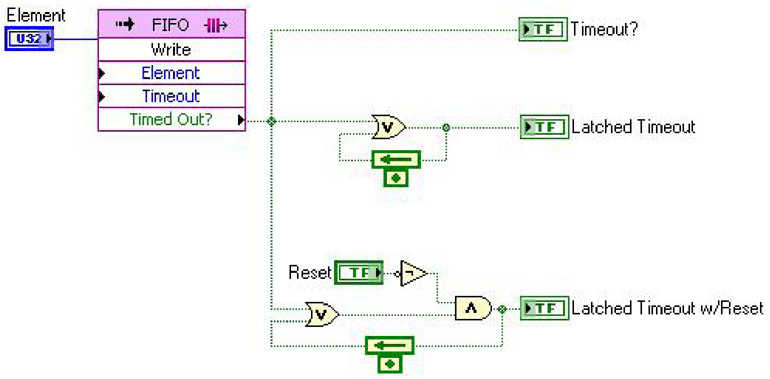


Figure 6.11. Example of No Latch, Simple Latch, and Latch with Reset

If you are receiving full flags, you need to either increase buffer size on the host, read larger chunks on the host, or read faster on the host. Keep in mind that many control and monitoring applications need only the most up-to-date data. Therefore losing data may not be an issue for a system as long as it returns the most recent data when called.

Another consideration for DMA transfer is using one DMA FIFO for multiple channels. In hybrid mode CompactRIO systems only have one DMA channel available. To pack multiple channels into one DMA FIFO, use an interleaving technique, and unpack using decimation on the host.

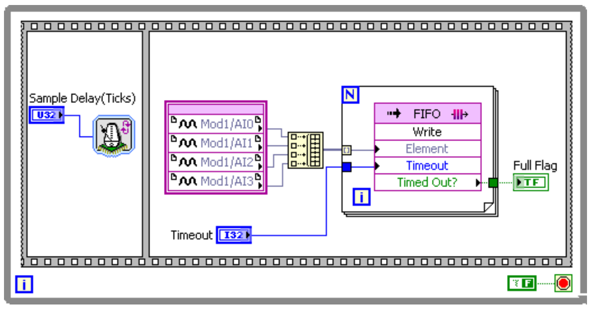


Figure 6.12. You can use build array and indexing on a for loop to implement an interleaved multichannel data stream.

Reading DMA Channels from the Real-Time Program

On your real-time program, you need another loop to read the array. To retrieve the data from the DMA buffer, you have to use the FPGA interface VIs.

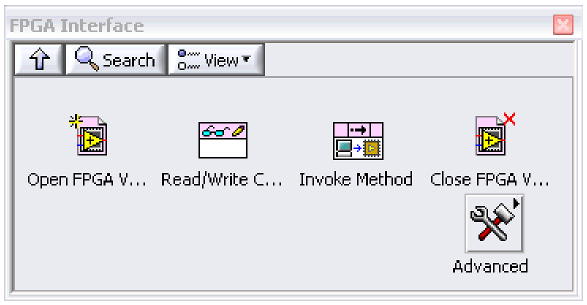


Figure 6.13 The FPGA interface Palette

Open FPGA VI Reference

Opens a reference to the FPGA VI or bitfile and FPGA target you specify. Right-click the Open FPGA VI Reference function and select Configure Open FPGA VI Reference from the shortcut menu to display the Configure Open FPGA VI Reference dialog box. Choose your FPGA VI from the dialog.

Read/Write Control

Reads a value from or writes a value to a control or indicator in the FPGA VI on the FPGA target. This can be a trigger condition, sampling rate, or any other data or parameter set by a control or indicator in the FPGA VI.

Invoke Method

Invokes an FPGA interface method or action from a host VI on an FPGA VI. Use these methods to implement the following: download, abort, reset, and run the FPGA VI on the FPGA target; wait for and acknowledge FPGA VI interrupts; read DMA FIFOs; and write to DMA FIFOs. The methods you can choose from depend on your target hardware and the FPGA VI. You must wire the FPGA VI Reference In input to view the available methods in the shortcut menu. For the waveform acquisition, use this node to read the DMA FIFO into an array.

Close FPGA VI Reference

Closes the reference to the FPGA VI and, optionally, resets or aborts the execution of the VI.

Open a reference to the FPGA VI, set parameters, use the Invoke Node in a task loop to read waveform data, and close the FPGA reference to implement the simplest DMA read. For many applications, default configurations are sufficient. However, as previously discussed, complexities can arise from buffer sizes, timeouts, and synchronization. You can search on advanced FPGA DMA topics to understand and learn how to deal with these issues.

Below is a simple real-time application that reads a waveform from the FPGA, performs an average calculation on the waveform and passes the data to a separate control loop that is running a PID loop to control a PWM output based on the waveform average. This type of application might be used to control a signal generator or laser that is tuned using a PWM input signal.

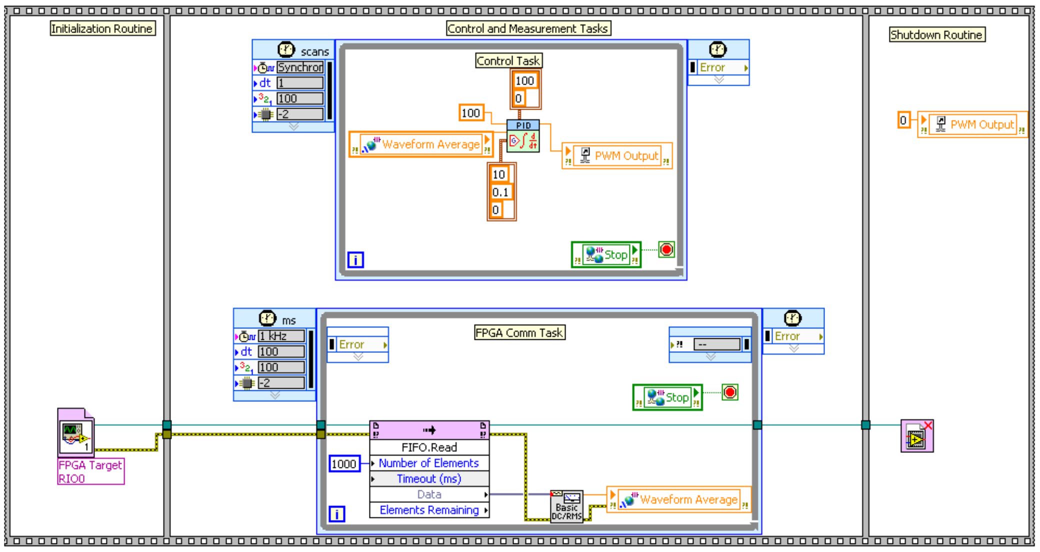


Figure 6.14. Simple DMA Read Using the FPGA Interface on the Real-Time Host

Example – Inserting Single-Point Data from an FPGA into a Real-Time Scan

LabVIEW Example Code
is provided for this section

Inline Processing

Besides waveform acquisition, another reason to use the FPGA is inline processing. A given system may not need all the data from the high-speed signal on the real-time controller but may require some type of processed data from an input signal. Examples of inline processing include filtering, averaging, or measurements. In the last example, you passed a waveform to real-time hardware and then performed a root-mean-square (RMS) calculation. Instead, you can use LabVIEW FPGA to do the RMS calculation and just pass the result.

LabVIEW FPGA has several built-in functions for processing as well as support for most LabVIEW constructs to help you create custom processing. Most of the procedure for implementing inline processing is the same as waveform acquisition. You still need to use an I/O module in FPGA hybrid mode by dragging it under the FPGA in the project. Create a new FPGA VI and add I/O in the same way. The main differences are which code you put in the FPGA VI and the necessity of buffered acquisition with DMA. Any control or indicator on the front panel of a LabVIEW FPGA VI can be read or written directly from RT.

Figure 6.15 shows the FPGA code to read channel 0, make an RMS calculation, and pass the results to the real-time program.

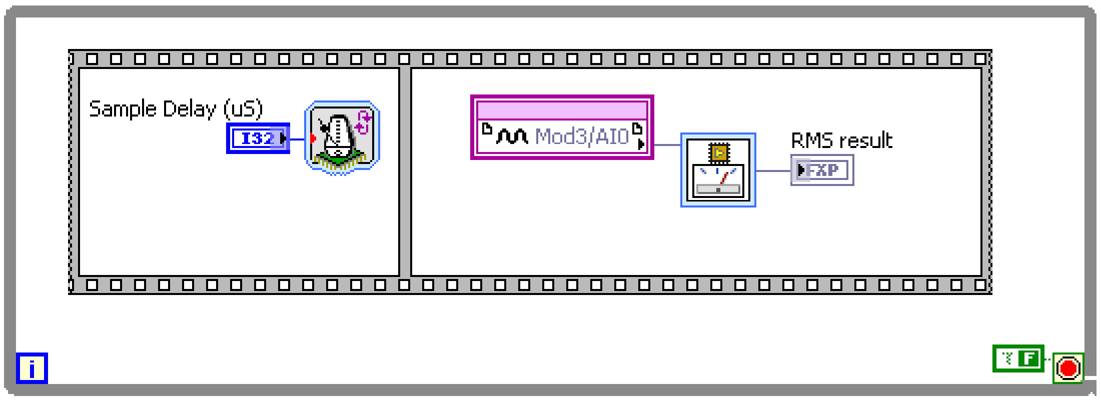


Figure 6.15. Inline Processing on LabVIEW FPGA – A high-speed analog signal is processed to determine the RMS measurement

You can modify the real-time code so you can read the RMS result directly, check to see if any errors or warnings occurred, and then pass it into the memory table using the single-process shared variable. Because this is a deterministic operation, you can run this in a timed loop with higher priority. This ensures that you get a new data point each time the PID loop runs.

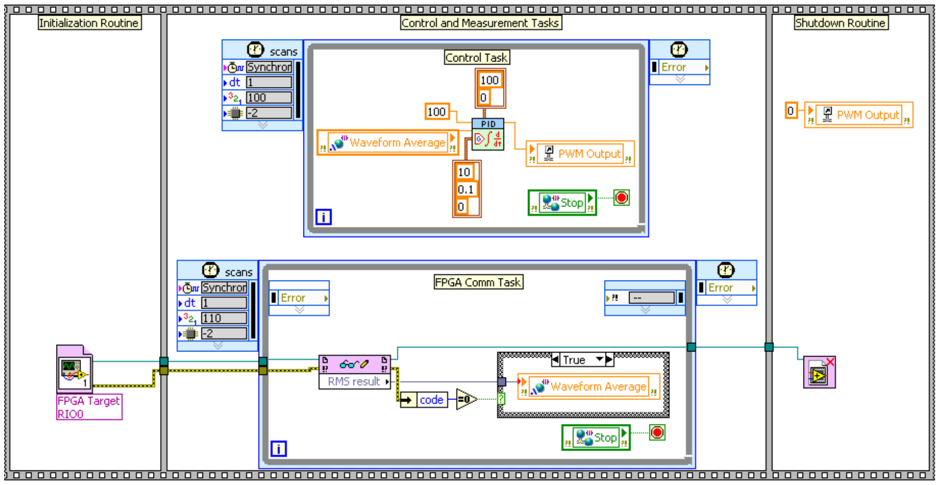


Figure 6.16. Deterministic data from the FPGA is read and inserted into the memory table.

C Series Modules without Scan Mode Support

Most C Series modules feature scan mode support, but some specialty modules such as motor drivers, CAN, serial, and SD card modules do not. Additionally CompactRIO is open, allowing customers and third parties to create C Series modules. To use these modules in your CompactRIO system, make sure they are under the FPGA in the project to place them in hybrid mode. Because these are not typical analog or digital modules, there is no generalized API for the FPGA, but NI modules have examples in the NI Example Finder that include LabVIEW FPGA and LabVIEW Real-Time host code and most third-party modules ship with examples.

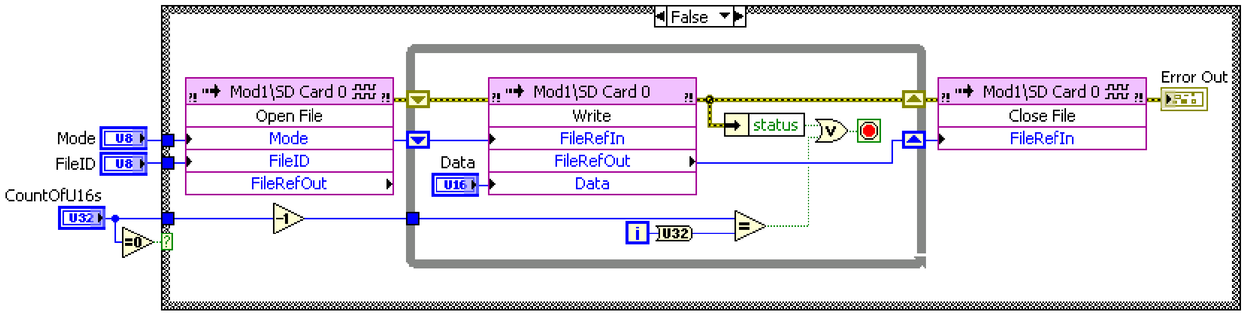


Figure 6.17 Write SD Card Example for the NI 9802

LabVIEW FPGA Development Best Practices

This section covers a number of advanced tips and tricks to cut your development time when creating high-performance control systems with the LabVIEW FPGA Module and CompactRIO. It examines debugging techniques such as simulation as well as several recommended programming practices, ways to avoid common mistakes, and numerous methods to create fast, efficient, and reliable LabVIEW FPGA applications.

To get this most from this section, you need to have a basic knowledge of LabVIEW FPGA programming techniques.

This section features examples developed to create a high-performance control system for a brushed DC motor. Explore a variety of the programming techniques used in the creation of LabVIEW FPGA subVIs for generating the PWM drive signal, decoding the digital pulses from the quadrature encoder sensor, and performing PID control to close the motor position loop. The end result is a high-performance control system with subnanosecond timing jitter and multiple 40 MHz processing loops that consumes only 17 percent of a 3M gate FPGA.

Now consider five key development techniques to create reliable, high-performance LabVIEW FPGA applications.

Technique 1. Use Single-Cycle Timed Loops (SCTLs)

The first development technique focuses on the use of single-cycle timed loops, or SCTLs, in the LabVIEW FPGA Module.

SCTLs, which tell the LabVIEW FPGA compiler to optimize the code inside, add the special timing constraint requiring the code to execute in a single tick of the FPGA clock. Code compiled inside an SCTL is more optimized and takes up less space on the FPGA compared to the same code inside a regular while loop. Code inside an SCTL also executes extremely fast. At the default clock rate of 40 MHz, one cycle is equal to just 25 ns.

Figure 6.18 shows two identical LabVIEW FPGA applications – the one on the left uses normal while loops and the one on the right uses SCTLs in its subVIs. This example illustrates the power of parallel processing. The upper loop is reading and processing the digital signals from a quadrature encoder sensor on a motor and the lower loop is performing PWM to control the amount of power sent to the motor. This application is written for the NI 9505 motor drive module, which controls brushed DC motors. This code runs extremely fast – the application on the right is running two different loops at a 40 MHz clock rate.

<u>While Loops</u> | <u>Single-Cycle Timed Loops</u>

Number of SLICEs: 3245 out of 14336 22% | **Number of SLICEs: 2456 out of 14336 17%**

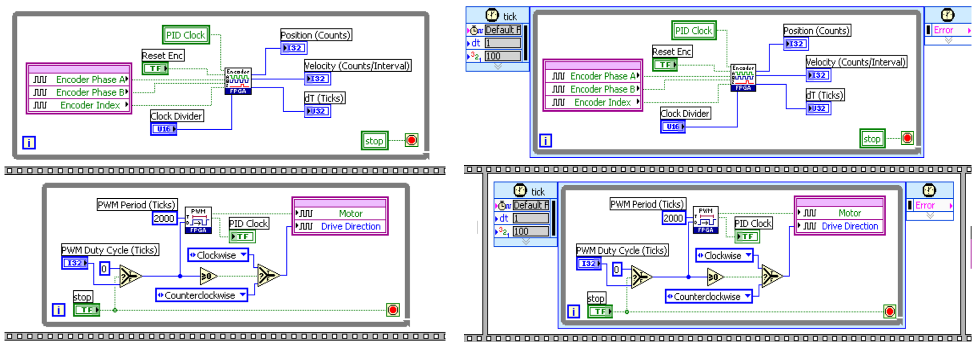


Figure 6.18. The Power of Parallel Processing

The results from your compile report are also shown. The application built with SCTLs uses fewer SLICEs, but it takes longer to compile because the compiler has to work harder to meet the timing constraints applied by the SCTL.

Now examine how the SCTL works in more depth.

When code is compiled in a normal while loop, LabVIEW FPGA inserts flip-flops to clock data from one function to the next, thereby enforcing the synchronous dataflow nature of LabVIEW and preventing race conditions. The flip-flops are marked in Figure 6.19 with the FF boxes drawn at the output of each function.

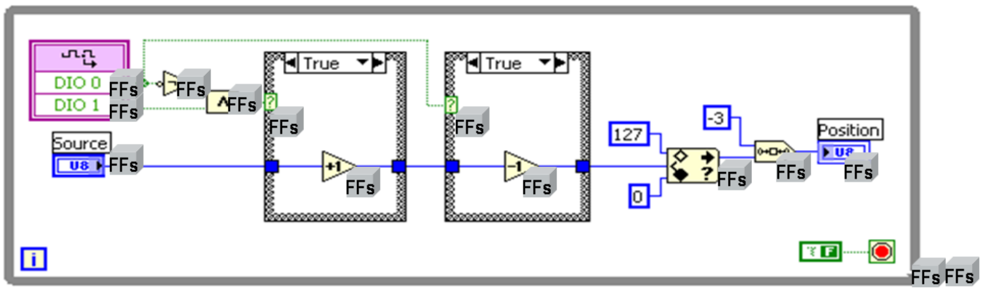


Figure 6.19. When code is compiled in a normal while loop, LabVIEW FPGA inserts flip-flops (marked as "FFs") to clock data from one function to the next.

Figure 6.20 shows the same code compiled into an SCTL. Only the inputs and outputs of the loop have flip-flops. The internal code is implemented in a more parallel fashion, and more logic reduction is implemented to optimize the code in between the inputs and outputs of the loop.

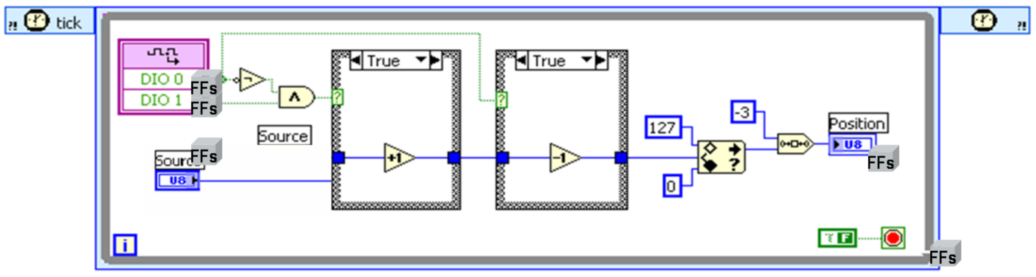


Figure 6.20. The Code from Figure 6.19 Compiled into an SCTL

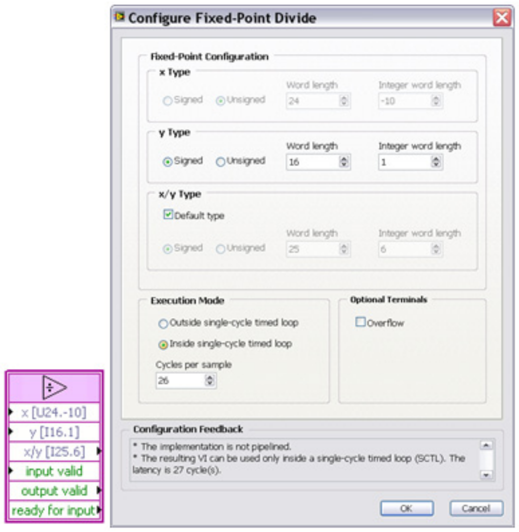
As you can see, SCTLs are a simple way to optimize your LabVIEW FPGA code, but there are some limitations to using them, as shown in Table 6.1.

Items Not Allowed in SCTL	Suggested Alternative
Long sequences of serial code	Make the code more parallel. Insert feedback nodes in the wires to add pipelining.
Quotient and remainder	Use a Scale by Power of 2 to do integer division, or use the fixed point math library
Loop timer, wait functions	Use a Tick Count function to trigger an event instead
Analog input, analog output	Place in a separate while loop and use local variables to send data
While loops	For nested subVIs, use feedback nodes to hold state

Table 6.1. Single-Cycle Timed Loop (SCTL) Limitations

To use the SCTL, all operations inside the loop must fit within one cycle of the FPGA clock. In the beginning of the compile process, the code generator emits an error message if the SCTL cannot generate the proper code for the compiler. That means that long sequences of serial code may not be able to fit in an SCTL. Serial code is code in which the results of one calculation are needed by the next operation, preventing parallel execution of the calculations. To fix this, you can rewrite the code to make it more parallel. For example, you can insert a Feedback Node ( ) to pass the results from one calculation to the next on the following iteration of the loop – this is known as pipelining. You can use the pipelining technique to reduce the length of each run through the SCTL by breaking up the code among multiple iterations of the loop.

You cannot use the **Quotient and Remainder** function in an SCTL. If you need to divide a number by an integer value, you can use the **Scale by Power of 2** function instead. With this function, you can multiply or divide by powers of two, that is, 2, 4, 8, 16, 32, and so on. For a fixed-point result, you can use the Fixed-Point Math Library for LabVIEW FPGA. Figure 6.21 shows the fixed-point divide subVI and configuration panel including the Execution Mode control that enables the function to be used within an SCTL.


Configure Fixed-Point Divide

Fixed-Point Configuration
x Type: Signed [✓ Unsigned]; Word length 24; Integer word length -10
y Type: [✓ Signed] Unsigned; Word length 16; Integer word length 1
x/y Type: ☑ Default type; [✓ Signed] Unsigned; Word length 25; Integer word length 6

Execution Mode: Outside single-cycle timed loop [✓ Inside single-cycle timed loop]
Cycles per sample: 26

Optional Terminals: ☐ Overflow

Configuration Feedback
* The implementation is not pipelined.
* The resulting VI can be used only inside a single-cycle timed loop (SCTL). The latency is 27 cycle(s).

OK Cancel

x [U24.-10]
y [I16.1]
x/y [I25.6]
input valid
output valid
ready for input

Figure 6.21. Fixed-Point Divide SubVI and Configuration Panel

The Fixed-Point Math Library contains LabVIEW FPGA IP blocks that implement a variety of elementary and transcendental math functions. These functions use the fixed-point data type introduced in LabVIEW 8.5, which extends the current offering of functions to include divide, sine, and cosine. All functions are verified for usage inside and outside an SCTL as well as in Windows and FPGA simulation on the development computer. The library features help documentation with information on each function. You can download it free of charge from **ni.com**.

If you are trying to create a subVI to implement inside an SCTL, you can use a feedback node to hold state information in the subVI. This eliminates the need to use a while loop inside an SCTL. The LabVIEW FPGA example in Figure 6.22 calculates one of the differential equations for a DC motor using functions from the Fixed-Point Math Library. After each fixed-point math function, you use a feedback node to pipeline the result and thereby pass the value from one iteration to the next. Then you use the **Tick Count** function (in the upper right corner) with a feedback node to calculate the loop rate of the subVI execution.

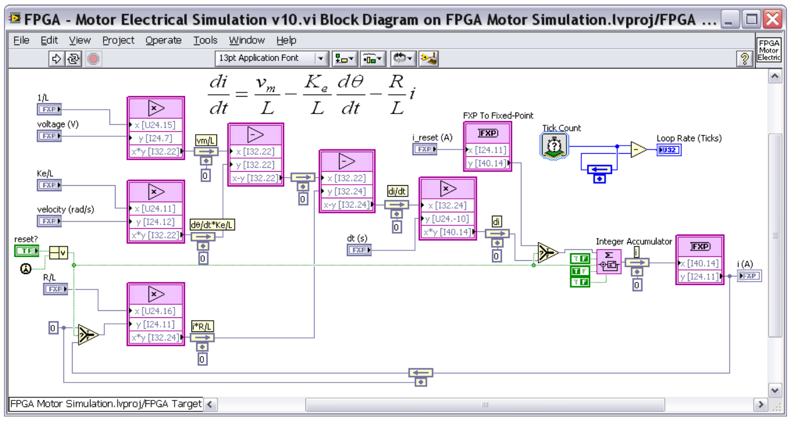

FPGA - Motor Electrical Simulation v10.vi Block Diagram on FPGA Motor Simulation.lvproj/FPGA ...
File Edit View Project Operate Tools Window Help
13pt Application Font
$\frac{di}{dt} = \frac{v_m}{L} - \frac{K_e}{L}\frac{d\theta}{dt} - \frac{R}{L}i$
1/L
voltage (V)
Ke/L
velocity (rad/s)
reset?
R/L
vm/L
dθ/dt*Ke/L
i*R/L
di/dt
dt (s)
di
i_reset (A)
FXP To Fixed-Point
Tick Count
Loop Rate (Ticks)
Integer Accumulator
i (A)
FPGA Motor Simulation.lvproj/FPGA Target

Figure 6.22. Calculating a Differential Equation for a DC Motor Using Fixed-Point Math Library Functions

In Figure 6.23, you can see the top-level SCTL in the FPGA application, which calls the motor simulation subVI. Because the subVI is nested within an SCTL, the **Loop Rate (Ticks)** value returned is always equal to 1. However, pipelining causes a six-tick latency from the **voltage (V)** input to the **i (A)** current output of the subVI.

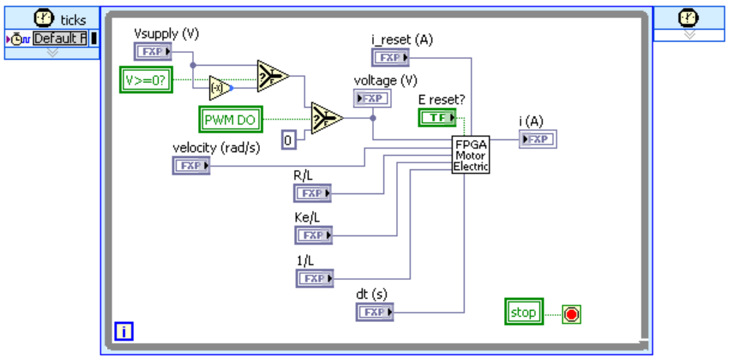


Figure 6.23. Pipelining causes a six-tick latency from the voltage (V) input to the i (A) current output of the subVI.

In addition to pipelining, you can use a state machine within the SCTL to better organize your code and run through a sequence of steps. The basic component of the state machine is a case structure, with each containing one state and using a shift register to determine the next state after each loop iteration. Each state must be able to run in one clock cycle if the subVI is to be placed in an SCTL. In addition, you can use shift registers and a counter value to implement the functionality of a **for loop** or add a specific number of **wait states** to your program execution.

Note: Adding a loop timer or wait function causes the code to execute slower than one tick, so you cannot use it in an SCTL. Analog input and analog output functions also take more than one clock tick to execute and cannot be used in an SCTL. However, you can put them in a normal while loop and use local variables to share data with the SCTLs.

Tip: Creating Counters and Timers

If you need to trigger an event after a period of time, use the Tick Count function to measure elapsed time as shown in Figure 6.24. Do not use the iteration terminal that is built into while loops and SCTLs because it eventually saturates at its maximum value. This happens after 2,147,483,647 iterations of the loop. At a 40 MHz clock rate, this takes only 53.687 seconds. Instead, make your own counter using an unsigned integer and a feedback node as well as the Tick Count function to provide time based on the 40 MHz FPGA clock.

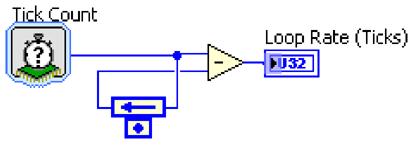


Figure 6.24. Using the Tick Count Function to Measure Elapsed Time

Because you used an unsigned integer for the counter value, your elapsed time calculations remain correct when the counter rolls over. This is because if you subtract one count value from another using unsigned integers, you still get the correct answer even if the counter overflows.

Another common type of counter is an iteration counter that measures the number of times a loop has executed. Unsigned integers are typically preferred for iteration counters because they give the largest range before rolling over. The unsigned 64-bit integer data type you are using for the counter provides a huge counting range – equivalent to about 18 billion-billion. Even with the FPGA clock running at 40 MHz, this counter will not overflow for more than 14,000 years.

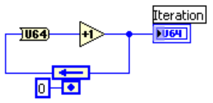


Figure 6.25. Unsigned integers are typically preferred for iteration counters because they give the largest range before rolling over.

Now examine another technique to create well-written and efficient LabVIEW FPGA code.

Technique 2. Write Your FPGA Code as Modular, Reusable SubVIs

The next major development technique is modular development – breaking your application into independent functions that you can individually specify, design, and test. It seems like a simple concept, but for FPGA development, it can have some especially nice benefits. For example, consider a function designed to measure the rate of the loop in which it is placed and count the number of times it executes. Inside the loop is a Tick Count function that reads the current FPGA clock and subtracts it from the previous value, which is stored in a shift register. In addition, you have a 64-bit counter that increments each time the function is called. This function uses an SCTL, so it takes only a single 25 ns clock tick to execute. Therefore, this subVI is designed to be placed inside a normal while loop without affecting its execution speed.

The front panel is shown in Figure 6.26. The indicators have been assigned to the two right terminals of the subVI so data can be passed to the upper-level LabVIEW FPGA application in which it is placed.

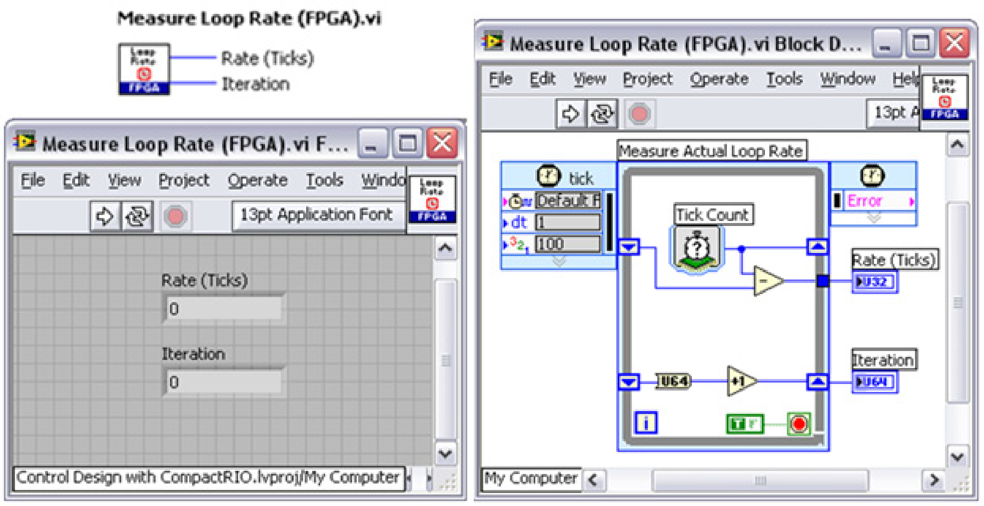

Measure Loop Rate (FPGA).vi
Rate (Ticks)
Iteration
Measure Loop Rate (FPGA).vi F...
File Edit View Project Operate Tools Windo
13pt Application Font
Rate (Ticks)
0
Iteration
0
Control Design with CompactRIO.lvproj/My Computer
Measure Loop Rate (FPGA).vi Block D...
File Edit View Project Operate Tools Window Help
13pt A
Measure Actual Loop Rate
tick
Default
dt 1
100
Tick Count
Error
Rate (Ticks)
U32
U64
+1
Iteration
U64
My Computer

Figure 6.26. Front Panel of a SubVI Designed to Be Placed inside a Normal While Loop without Affecting Its Execution Speed

The function is used in the application example in Figure 6.27. The subVI is placed inside another loop to measure the execution rate of the top-level code.

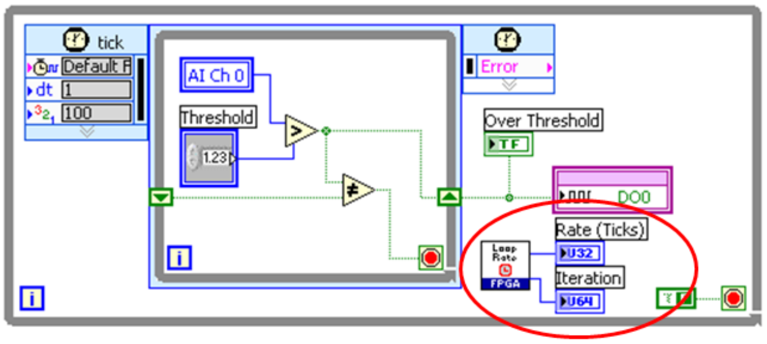


Figure 6.27 The subVI is placed inside another loop to measure the execution rate of the top-level code.

Review some of the top benefits of writing code this way in Table 6.2.

Benefit	**Explanation**
Easier to debug and troubleshoot	You can test code on Windows before compiling
Easier to document and track changes	You can include help information in the VI documentation
Creates a cleaner, more easily understood, top-level diagram	Code is more intuitive to other programmers

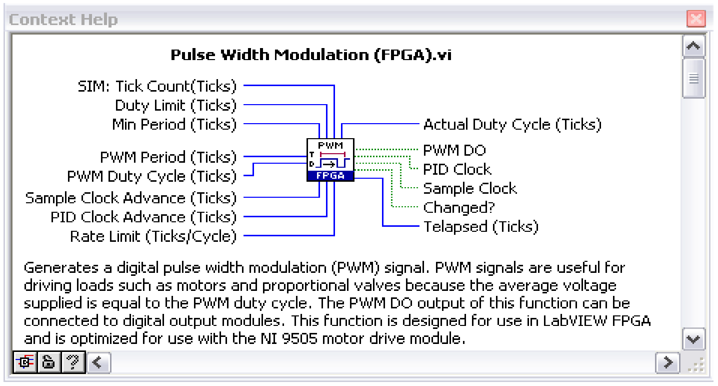
Table 6.2. Benefits of Writing Your FPGA Code as Modular, Reusable SubVIs

Writing modular code is almost always a good idea, but when you are designing FPGA logic, it has extra advantages.

First, the code is easier to debug and troubleshoot. One big benefit is that you can test the subVI on Windows before you compile it for the FPGA. Review examples of that later in this section.

Second, it is easier to document and track changes because the code is modular and you can include help information in the VI documentation.

Third, the intended functionality of the code is typically cleaner, easier to understand, and more reusable. The options you want to offer the programmer are typically made available as terminals on the subVI. Most often users do not need to modify the underlying code – they can just use the parameters you provide, such as in this **Pulse Width Modulation (FPGA). vi** example.



Generates a digital pulse width modulation (PWM) signal. PWM signals are useful for driving loads such as motors and proportional valves because the average voltage supplied is equal to the PWM duty cycle. The PWM DO output of this function can be connected to digital output modules. This function is designed for use in LabVIEW FPGA and is optimized for use with the NI 9505 motor drive module.

Figure 6.28. Programmers can use the parameters you provide in this VI.

Now consider some tips for creating modular, reusable subVIs for LabVIEW FPGA.

Tip: Separate Logic from I/O

The first tip is to keep I/O nodes out of your subVIs. This makes them more modular and portable and makes the top-level diagram more readable. Particularly for control applications, it is nice to have all of the I/O operations clearly visible when viewing the top-level diagram for the application, as shown in the PWM loop written for the NI 9505 motor drive module in Figure 6.29.

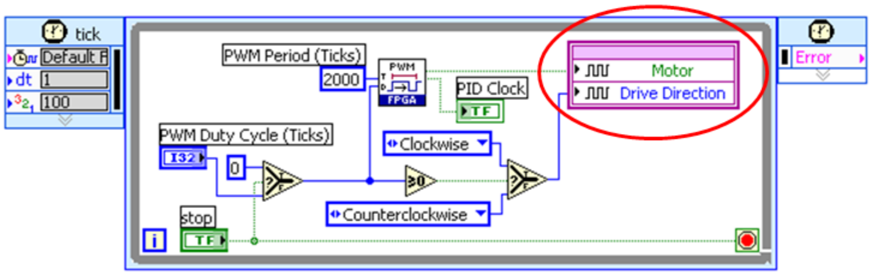


Figure 6.29. Keep I/O nodes out of your subVIs and make sure they are clearly visible in the top-level diagram for your application.

Rather than embedding the I/O node in the subVI, a terminal is used to pass the data from the subVI to the top-level diagram. This makes the FPGA code easier to debug because you can test the subVI individually in Windows using simulated I/O.

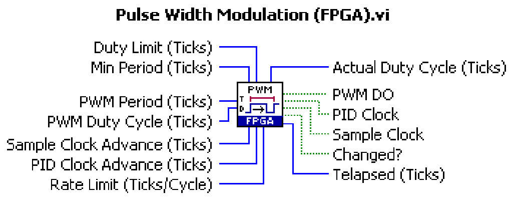


Figure 6.30. Rather than embedding the I/O node in the subVI, a terminal is used to pass the data from the subVI to the top-level diagram.

Taking this approach also reduces extraneous I/O node instances that might otherwise be included multiple times in the subVI, resulting in unnecessary gate usage due to the need for the compiler to add the extra arbitration logic necessary to handle multiple callers accessing the same shared resource.

Finally, this approach makes the top-level application more readable – all I/O read and write operations are explicitly shown on the top-level diagram and not hidden from view.

Often when you are writing function blocks like this, the subVI needs some local memory capability so it can hold state values, such as elapsed time, and pass that information from one iteration to the next.

Tip: Hold State Values in a Function Block

Figure 6.31 shows how you can add shift register nodes to your loop to pass information from one iteration of the loop to the next. The Iteration counter increments each time the function block is called.

Notice that the **Loop Condition** terminal has a constant wired to it that causes the loop to run for only one iteration each time it is called. You are not really looping in this case – you are simply using the SCTL to optimize the code and hold the state values with shift registers.

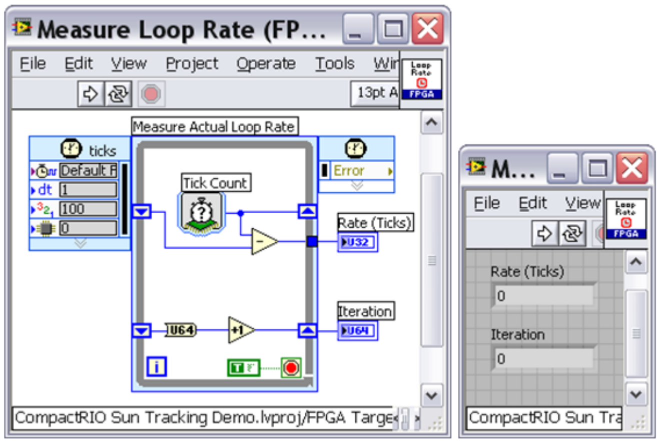


Figure 6.31. Add shift register nodes to your loop to pass information from one iteration of the loop to the next.

Note: The shift register must be uninitialized for the subVI to hold state this way. On first call, the shift register value is the default value for the data type – for integers, that is 0, for Booleans that is False. If you need to initialize the value to something else, use a **First Call?** function and a **Select** function .

You may be wondering how to create a modular function block that works inside an SCTL because you are not allowed to nest one SCTL within another.

You can use feedback nodes to accomplish the same task, as shown in Figure 6.32. The main benefit of this approach is that you can easily initialize the feedback nodes and place the subVI within a top-level SCTL because it contains no nested loop structure.

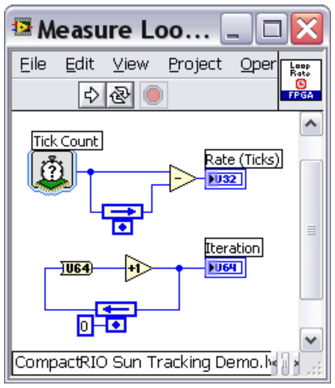


Figure 6.32. You can use feedback nodes to create a modular function block that works inside an SCTL.

A third option is the use of VI-scoped memory. This is a block of memory that the subVI can use locally and that does not have to be manually added to the project. This makes the code more modular and portable when moving it between projects.

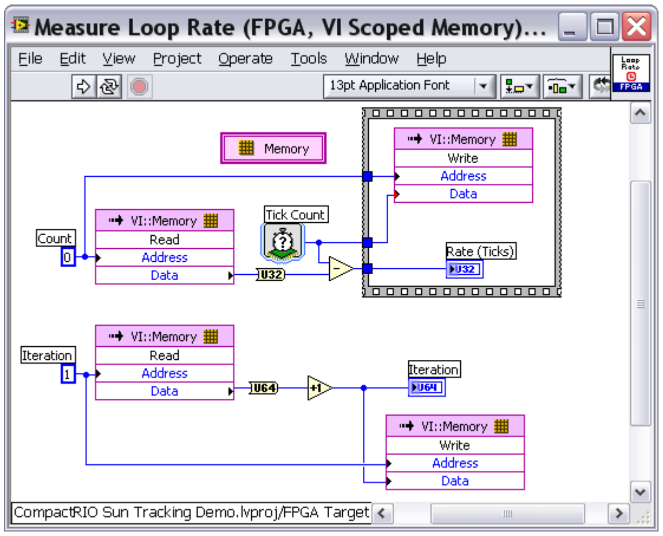


Figure 6.33. Use the subVI with VI-scoped memory to make the code more modular and portable.

In this simple example, using VI-scoped memory is probably overkill for the application. You have only two memory locations and are storing only one data point in each memory location. However, VI-scoped memory is a powerful tool for applications that need to store arrays of data. In general, you should always avoid using large front panel arrays as a data storage mechanism – use VI-scoped memory instead.

Run-Time Updatable Lookup Table (LUT)

A common use for local memory in FPGA-based control applications is to store table data, such as the calibration table for a nonlinear sensor, a precalculated math formula (such as log or exponential), or an arbitrary waveform that you can replay by indexing through the table addresses. Figure 6.34 shows an FPGA-based lookup table (LUT) configured to store 10,000 fixed-point values and perform linear interpolation between stored values. Because VI-scoped memory is used, you can change the LUT values while the application is running and without the need to recompile the FPGA.

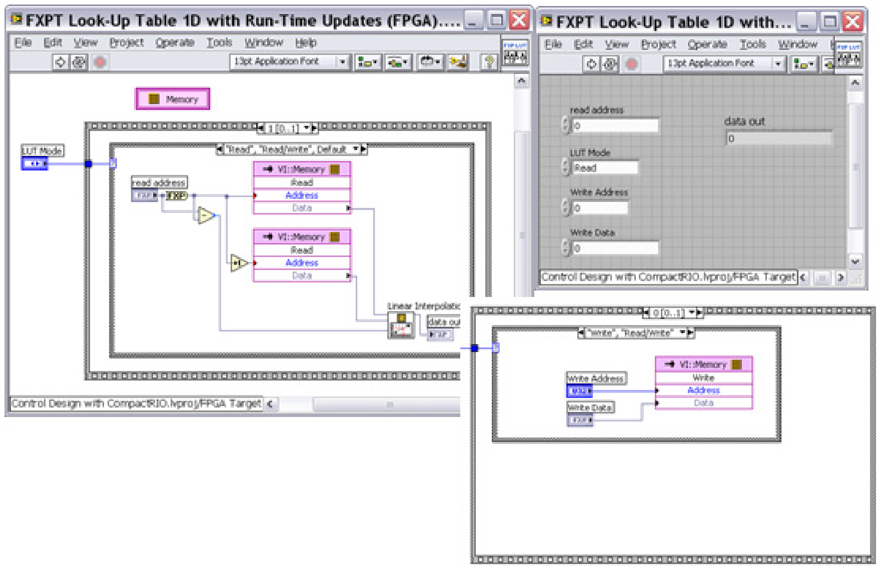
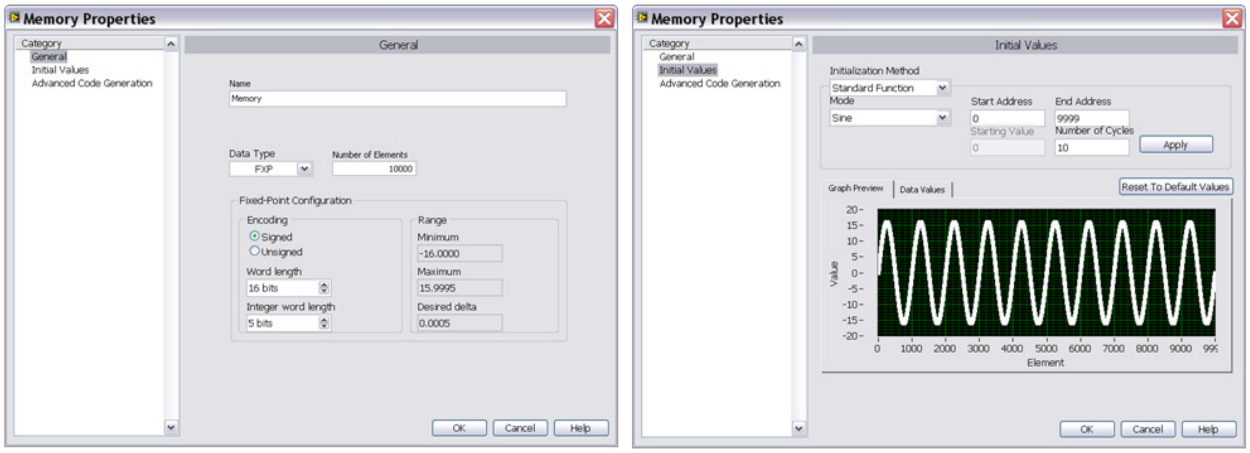


Figure 6.34. FPGA-Based LUT Configured to Store 10,000 Fixed-Point Values and Perform Linear Interpolation between Stored Values

Now examine the configuration pages for the VI-Scoped Memory block in this example. You can configure the depth and data type and even define initial values for the memory elements.



Memory Properties

Category: General

Name: Memory

Data Type: FXP

Number of Elements: 10000

Fixed-Point Configuration

Encoding: Signed

Word length: 16 bits

Integer word length: 5 bits

Range

Minimum: -16.0000

Maximum: 15.9995

Desired delta: 0.0005

Memory Properties

Category: Initial Values

Initialization Method: Standard Function

Mode: Sine

Start Address: 0

End Address: 9999

Starting Value: 0

Number of Cycles: 10

Figure 6.35. Configuration Pages for the VI-Scoped Memory Block in This Example

Consider another tip for creating modular FPGA subVIs that has to do with the timing of how the code runs.

Tip: Do Not Place Delay Timers in the SubVI

In general, it is a good idea to avoid using Loop Timer or Wait functions within your modular subVIs. If the subVI has no delays, it executes as fast as possible, thereby making inherent the timing properties of the calling VI rather than slowing down the caller. Also, you can typically more easily adapt in an SCTL if it has no internal functions that cause delays.

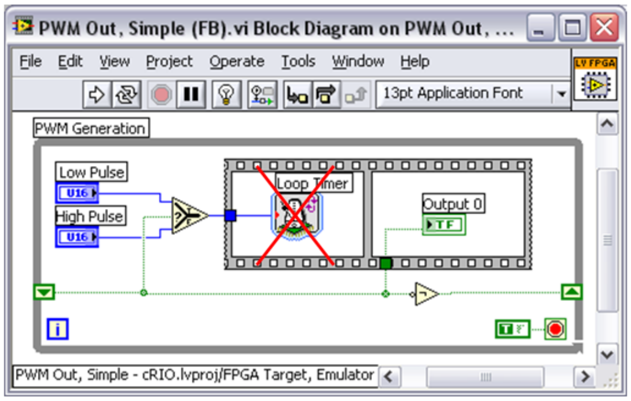

PWM Out, Simple (FB).vi Block Diagram on PWM Out, ...
File Edit View Project Operate Tools Window Help
13pt Application Font
PWM Generation
Low Pulse
U16
High Pulse
Loop Timer
Output 0
TF
PWM Out, Simple - cRIO.lvproj/FPGA Target, Emulator

Figure 6.36. Avoid using Loop Timer or Wait functions within your modular subVIs.

The left side of Figure 6.37 shows how you can adapt PWM code to use a Tick Count function rather than a Loop Timer function. Using a feedback node to hold an elapsed time count value, you turn the output on and off at the appropriate times and reset the elapsed time counter at the end of the PWM cycle. The code may look a bit more complicated, but you can drop it inside a top-level loop without affecting the overall timing of the loop – it is more portable.

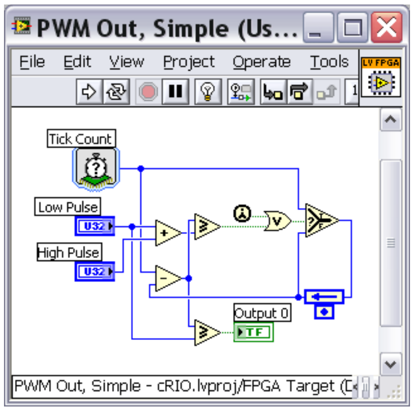
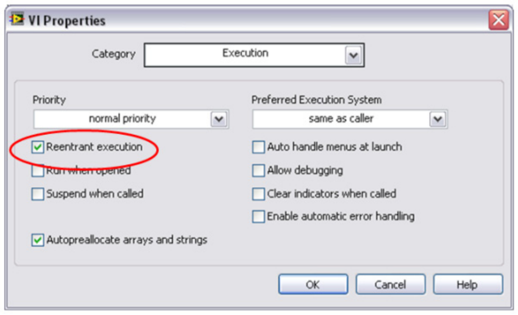

PWM Out, Simple (Us...
File Edit View Project Operate Tools
Tick Count
Low Pulse
U32
High Pulse
Output 0
TF
PWM Out, Simple - cRIO.lvproj/FPGA Target

Figure 6.37. Adapt PWM code to use a Tick Count function rather than a Loop Timer function.

Now examine one more tip before moving on to the next topic – make code so that you can place multiple copies of a subVI in the same application and each copy is independent of the others.

Tip: Take Advantage of Reentrancy

Reentrancy is a setting in the subVI Execution properties that you can use to execute multiple copies of the function block in parallel with distinct and separate data storage.



VI Properties

Category: Execution

Priority: normal priority

Preferred Execution System: same as caller

[✓] Reentrant execution

[ ] Run when opened

[ ] Suspend when called

[ ] Auto handle menus at launch

[ ] Allow debugging

[ ] Clear indicators when called

[ ] Enable automatic error handling

[✓] Autopreallocate arrays and strings

OK | Cancel | Help

Figure 6.38. Use reentrant execution to make multiple copies of the function block in parallel with distinct and separate data storage.

Figure 6.39 illustrates an example. In this case, your subVI is set to reentrant, so all four of these loops run simultaneously and any internal shift registers, local variables, or VI-scoped memory data are unique to each instance.

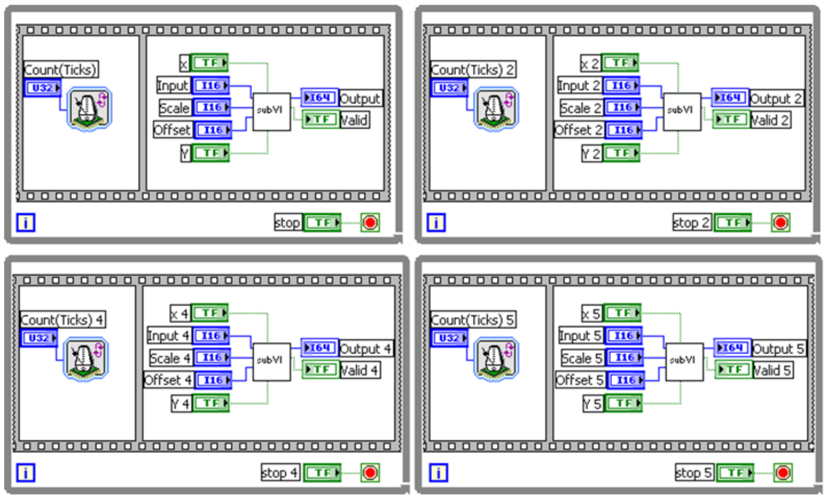


Figure 6.39. A subVI is set to reentrant, so all four of these loops run simultaneously and any internal shift registers, local variables, or VI-scoped memory data are unique to each instance.

In the case of LabVIEW FPGA, this also means that each copy of the function uses its own FPGA slices – so reentrancy is great for code portability but it does use more gates.

If you are really squeezed for FPGA gates, you can make your function multiplexed rather than reentrant. This advanced topic is not covered in this section, but it basically involves using local memory to store the register values for each of the calling loops, which identify themselves with an integer "ID tag" value. Because the loops all use the same underlying FPGA slices (with different memory addresses for the data), each caller blocks the other callers, resulting in slower execution. However, gate usage is much less since the same hardware SLICE logic is reused. For many control applications where the FPGA is already much faster than the I/O, this is a nice option for saving gates. Several functions on the LabVIEW FPGA palette use multiplexing techniques to enable high-channel-count operation with minimal FPGA gate usage. These include the **PID**, **Butterworth Filter**, **Notch Filter**, and **Rational Resampler** functions. To see how this works, drop one of these functions onto the block diagram and configure it for multiple channels. Then right-click on the function and select **Convert to SubVI** to reveal the underlying code.

Now take a look at a major development benefit you get from writing your LabVIEW FPGA code as described in the previous sections.

Technique 3. Use Simulation before You Compile

This third development technique is really powerful because it provides a way to get around the longer compilation time and more limited debugging capabilities of LabVIEW FPGA. One of the most important aspects of LabVIEW code for embedded developers is the portability of the code. Code written for LabVIEW FPGA is still just LabVIEW code – you can run it on Windows or other devices and operating systems. The main difference between these processing targets is the speed at which the code runs and whether the targets support true parallel processing like an FPGA or simulated parallel processing like a multithreaded operating system for a microprocessor.

LabVIEW FPGA includes the ability to run the entire LabVIEW FPGA application in simulation mode, which you can do in conjunction with the host processor application for testing purposes with either random data used for FPGA I/O read operations or with a custom VI to generate the simulated I/O signals. This is particularly useful for testing FPGA to host communication including DMA data transfers.

However, the disadvantage of this approach is that the entire FPGA application is simulated. For the development and testing of new LabVIEW functions, it can be advantageous to test the code one function at a time. This section focuses on a capability called functional simulation, which enables a "divide and conquer" approach to debugging so you can test each function individually before compiling to the FPGA . Figure 6.40 features screens from two functional simulation examples running on Windows that were used for testing and debugging purposes.

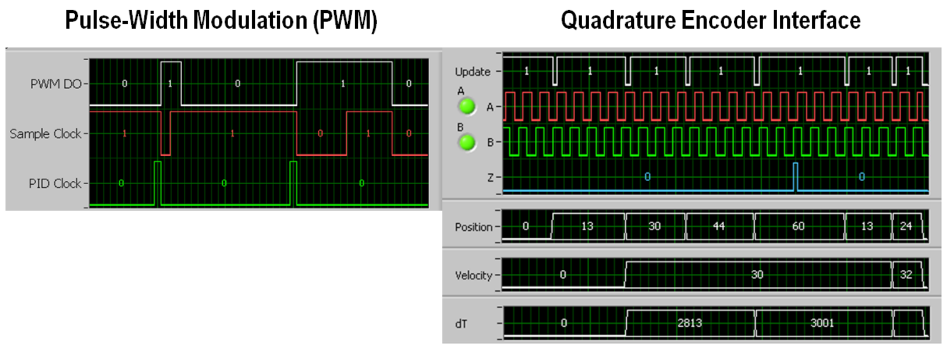


Figure 6.40. Screens from Two Functional Simulation Examples Running on Windows That Were Used for Testing and Debugging Purposes

The example in Figure 6.41 shows the front panel and block diagram of a test application used to debug a LabVIEW FPGA subVI for PWM. The test application is located in the **My Computer** section of the LabVIEW Project, and when it is opened, it runs in Windows.

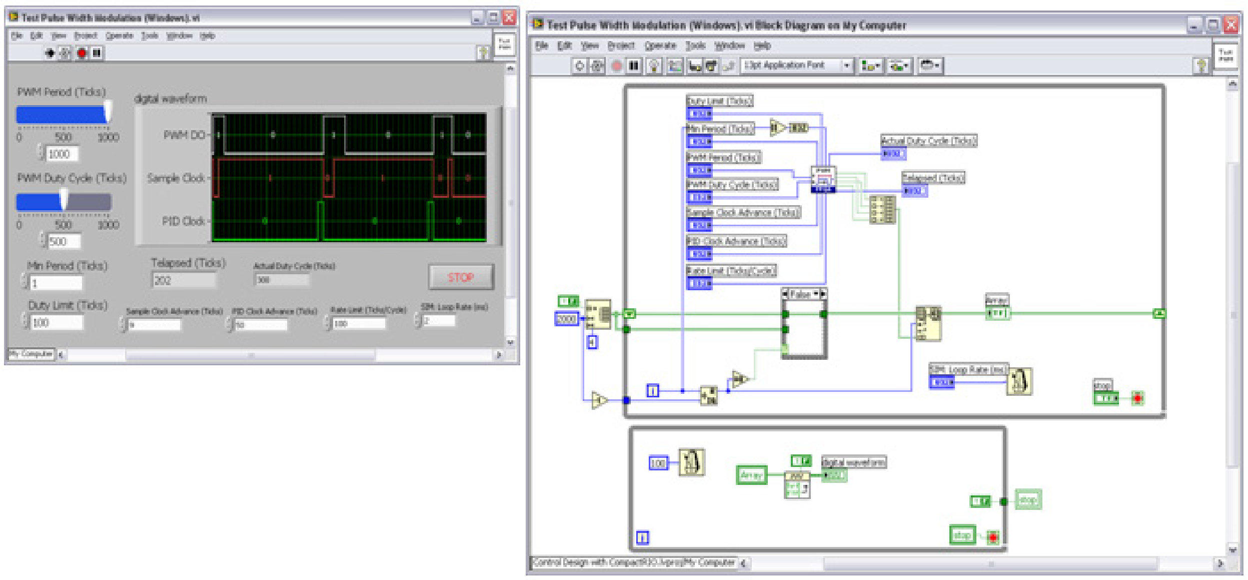

Test Pulse Width Modulation (Windows).vi
Test Pulse Width Modulation (Windows).vi Block Diagram on My Computer
PWM Period (Ticks)
PWM Duty Cycle (Ticks)
digital waveform
Min Period (Ticks)
Telapsed (Ticks)
Actual Duty Cycle (Ticks)
Duty Limit (Ticks)
STOP
Sample Clock Advance (Ticks)
PID Clock Advance (Ticks)
Rate Limit (Ticks/Cycle)
SIM Loop Rate (ms)
Elapsed (Ticks)
stop
Control Design with CompactRIO.lvproj/My Computer

Figure 6.41. Front Panel and Block Diagram of a Test Application Used to Debug a LabVIEW FPGA SubVI for PWM

Tip: Provide Tick Count Values for Simulation
With the Conditional Disable Structure in LabVIEW, you modify which underlying code is used when the subVI is compiled for different processing targets. In this case, you have a Tick Count function that is executed when the code is compiled for the FPGA and a front panel control that is executed when the code is executed on Windows. Because of this, you can use a "simulated" tick count value when you test the code in Windows to help create both bit-accurate and cycle-accurate simulations.

This technique is used in the PWM test example in Figure 6.41– when the subVI is executed in Windows, a simulated FPGA clock is passed to the subVI by using the Iteration terminal of the top-level while loop.

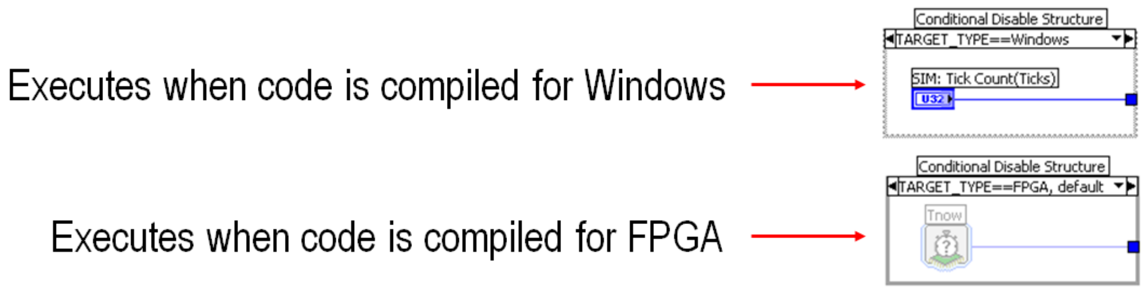

Executes when code is compiled for Windows
Conditional Disable Structure
TARGET_TYPE==Windows
SIM: Tick Count(Ticks)
U32
Executes when code is compiled for FPGA
Conditional Disable Structure
TARGET_TYPE==FPGA, default
Tnow

Figure 6.42. A Tick Count function is executed when the code is compiled for the FPGA, and a front panel control is implemented when the code is executed on Windows.

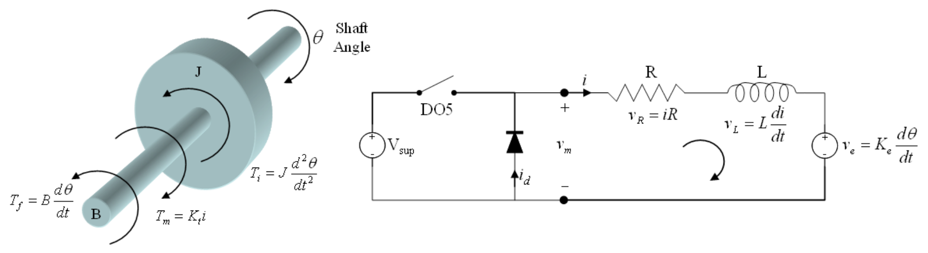
As you have seen, you can use functional simulation to test, iterate, and be confident in your FPGA logic before you compile. It also helps you use the full LabVIEW debugging toolset while the code is running, and you can create "test patterns" to verify the code under a variety of conditions that otherwise might be hard to test. Using simulation as a step in your development process, you can do the following:

- Quickly iterate and add features
- Be confident in your LabVIEW FPGA code before you compile
- Use full LabVIEW debugging capabilities (probes, highlight execution, and so on)
- Verify the code under a variety of conditions

Now take simulation a step further and create a simulation that accurately mimics the dynamic closed-loop behavior of the physical system within which your LabVIEW FPGA code is connected.

Tip: Test the LabVIEW FPGA Code Using the LabVIEW Control Design and Simulation Module

The LabVIEW Control Design and Simulation Module includes state-of-the-art technology for simulating mechatronics systems like the DC motor you are controlling with your LabVIEW FPGA application. Figure 6.43 shows the theoretical model equations for a brushed DC motor driven by a PWM chopper circuit and connected to a simple inertial load with viscous friction.



Sum of Forces

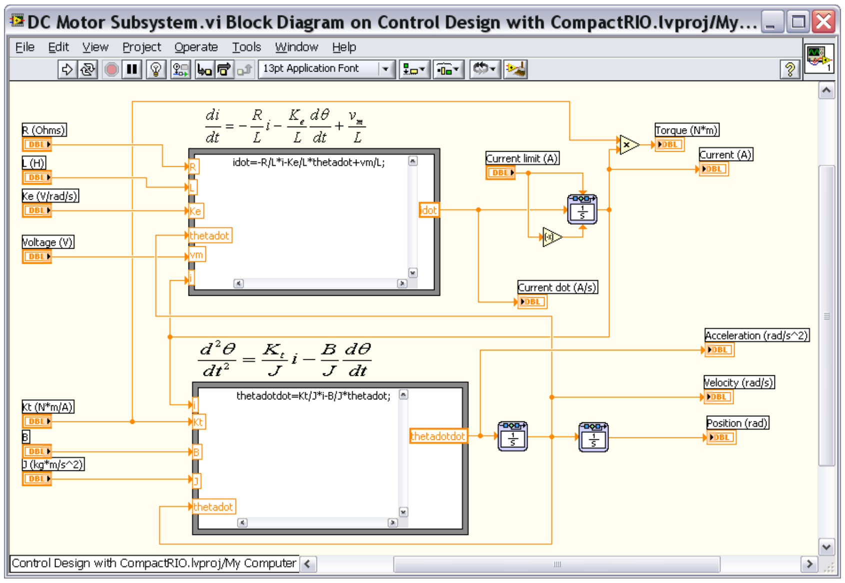
$$\frac{d^2\theta}{dt^2} = \frac{K_t}{J}i - \frac{B}{J}\frac{d\theta}{dt}$$

Kirchhoff's Voltage Law

$$\frac{di}{dt} = -\frac{R}{L}i - \frac{K_e}{L}\frac{d\theta}{dt} + \frac{v_m}{L}$$

Figure 6.43. Theoretical Model Equations for a Brushed DC Motor Driven by a PWM Chopper Circuit and Connected to a Simple Inertial Load with Viscous Friction

You implement this using a LabVIEW Control Design and Simulation subsystem containing a formula node. Then you enter the two differential equations shown in Figure 6.43 into the formula nodes in a text format as shown in Figure 6.44. Use Integrator functions ( )to convert from higher-order derivatives, such as from acceleration to velocity and from velocity to position.


DC Motor Subsystem.vi Block Diagram on Control Design with CompactRIO.lvproj/My...

$$\frac{di}{dt} = -\frac{R}{L}i - \frac{K_e}{L}\frac{d\theta}{dt} + \frac{v_m}{L}$$

```
idot=-R/L*i-Ke/L*thetadot+vm/L;
```

$$\frac{d^2\theta}{dt^2} = \frac{K_t}{J}i - \frac{B}{J}\frac{d\theta}{dt}$$

```
thetadotdot=Kt/J*i-B/J*thetadot;
```

Control Design with CompactRIO.lvproj/My Computer

Figure 6.44. Text Format of the Two Differential Equations Shown in Figure 6.43

Then place the Brushed DC Motor.vi subsystem within a top-level simulation loop and connect it to the LabVIEW FPGA function to simulate the pulsed voltage signal used to drive the motor. The result is a high-fidelity closed-loop simulation of how the LabVIEW FPGA code behaves when connected to the real-world electromechanical system.

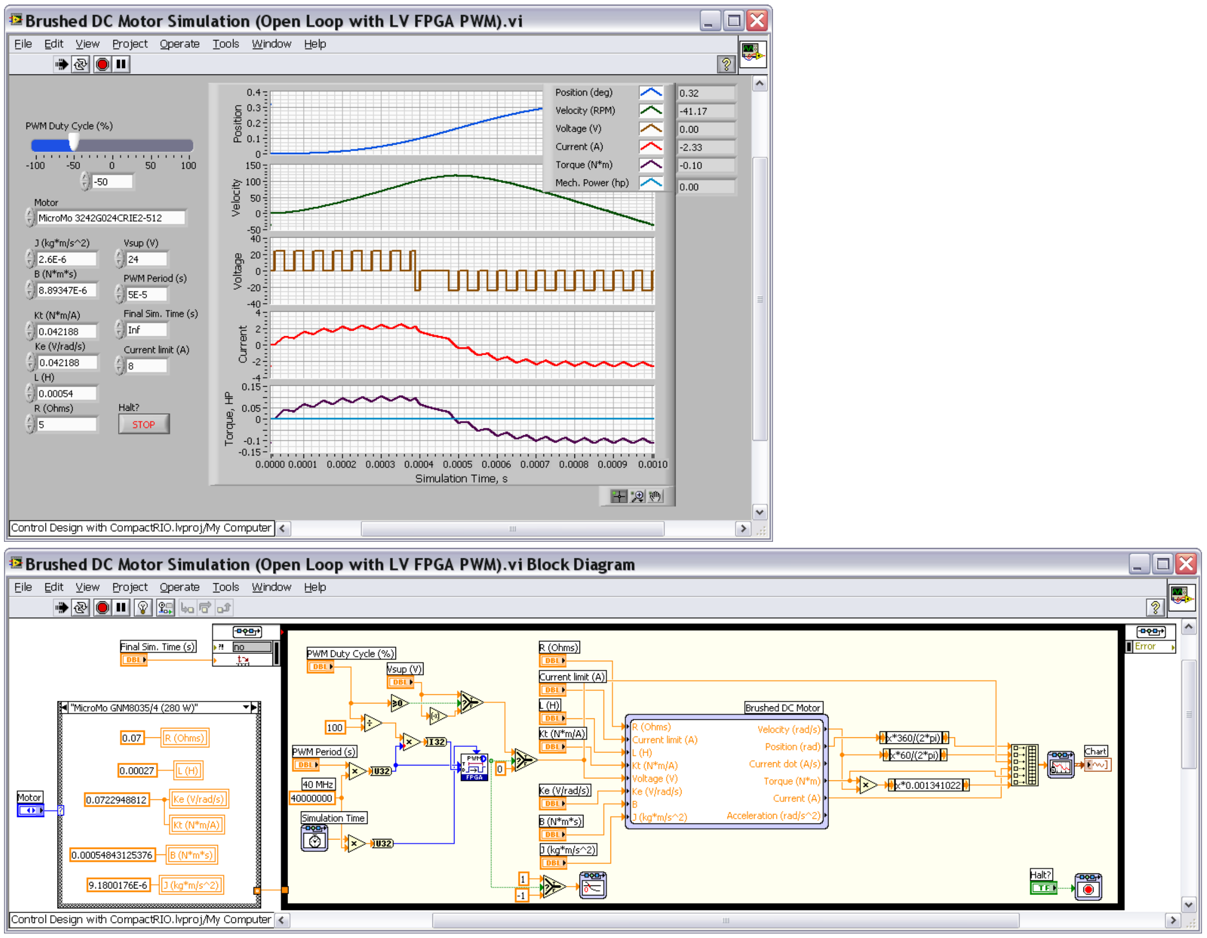


Figure 6.45. High-Fidelity Closed-Loop Simulation of How the LabVIEW FPGA Code Behaves When Connected to the Real-World Electromechanical System

The simulation results have been validated against actual measurements from the deployed LabVIEW FPGA application controlling a motor using the NI 9505 motor drive module, which showed a nearly identical match between the simulated and measured waveforms.

With this approach, you can take code validation way beyond basic functional validation. Think of this as a virtual machine emulator that you can use to anticipate how your code performs in the real world. You can use simulation to help make design decisions, evaluate performance, select components, and test worst-case conditions. You can even tune the PID control loops for your control system in simulation and see how well that tuning works with different motors and load conditions. Simulation can also help you select the right physical components for your system, such as picking the best motor to meet your performance requirements.

Technique 4. Synchronize Your Loops

Now for the fourth development technique: controlling the timing and synchronization of your LabVIEW FPGA code.

For most control applications, the timing of when the code executes is very important to the performance and reliability of the system. Fortunately, LabVIEW FPGA gives you both unprecedented speed and complete control over the timing of the code. Unlike a processor, an FPGA executes code in a truly parallel fashion rather than being able to execute only one instruction at a time. That makes programming easier because you do not have to worry about setting priorities and sharing the processor time among the different tasks. Each control loop is like a custom-designed processor that is completely dedicated to its task. The result is high-reliability, high-performance code. One of the benefits of this performance is that control loops are typically more stable, easier to tune, and more responsive to disturbances when they run at a fast rate.

In this motor control example, you have two different clock signals – a sample clock and a PID clock. These are Boolean signals you generate in the application to provide synchronization among the loops. You can trigger on either the rising or falling edge of these clock signals.

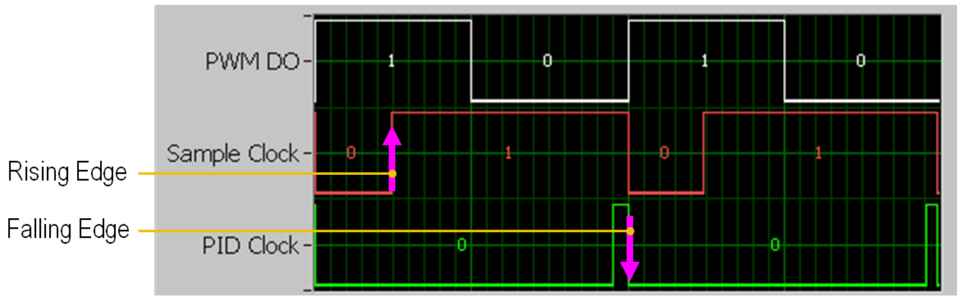
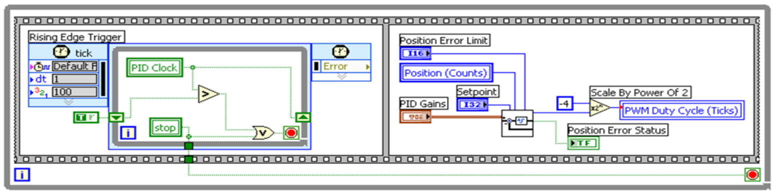


Figure 6.46. Motor Control Example with Two Different Clock Signals

Now consider the LabVIEW FPGA code used to monitor these signals and trigger on either the rising or falling edge.

Typically triggering a loop based on a Boolean clock signal works like this: first wait for the rising or falling edge to occur and then execute the LabVIEW FPGA code that you want to run when the trigger condition occurs. Engineers often implement a sequence structure for which they use the first frame of the sequence to wait for the trigger and the second frame to execute the triggered code, as shown in Figure 6.47.

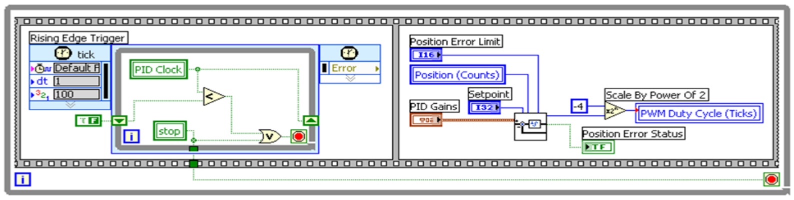
Rising Edge Trigger: In this case, you are looking for the trigger signal to transition from False (or 0) to True (or 1). This is done by holding the value in a shift register and using the **Greater Than?** function. (Note: A True constant is wired to the iteration terminal to initialize the value and avoid an early trigger on the first iteration.)



Rising Edge Trigger

Figure 6.47.Rising Edge Trigger Example

Falling Edge Trigger: In this case, use a **Less Than?** function to detect the transition from True (or 1) to False (or 0). (Note: A False constant is wired to the iteration terminal to initialize the value.)



Falling Edge Trigger

Figure 6.48.Falling Edge Trigger Example

Analog Level Trigger: Use a **Greater Than?** function to detect when the analog signal is greater than your analog threshold level, and then use the Boolean output of the function as your trigger signal. This case actually is a rising **or** falling edge detector because you are using the **Not Equal?** function to detect any transition.

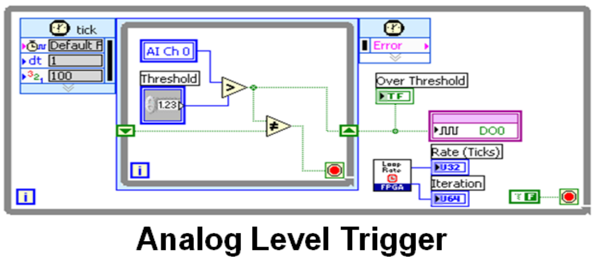


Figure 6.49.Analog Level Trigger Example

Now examine another common triggering use case – latching the value of a signal when a trigger event occurs.

Tip: Latch Values

In this case, you use a rising edge trigger to latch the **Analog Input** value from another loop into the **Latched Analog Input** register. This value is held constant until the next trigger event occurs. In this example, the actual analog input operation is occurring in another loop, and you are using a local variable for communication between the loops. (Note: Local variables are a good way to share data between asynchronous loops in LabVIEW FPGA.)

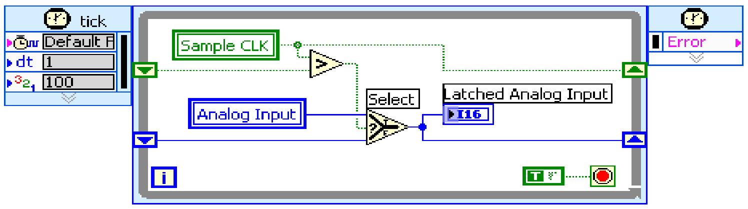


Figure 6.50. In this example, the actual analog input operation is occurring in another loop, and you are using a local variable for communication between the loops.

Synchronization with Triggering and Latching

Figure 6.51 shows an example of these triggering and latching techniques in practice. LabVIEW FPGA offers true parallel execution. In this case, you have three independent loops. This is like having three custom-designed processors running at the same time within the chip. Each loop is completely dedicated to its own task, resulting in the highest level of reliability. It also makes programming control in an FPGA easier to architect – unlike with a processor, you do not have to worry about your code slowing down when you add new code.

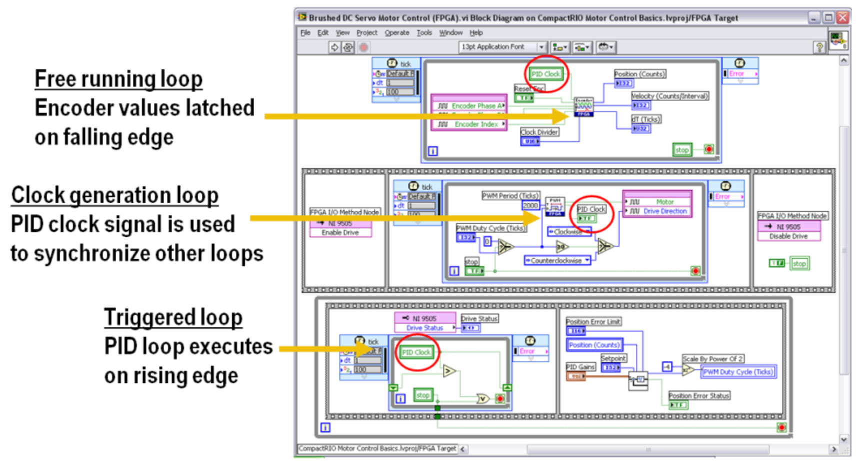

Free running loop
Encoder values latched on falling edge
Clock generation loop
PID clock signal is used to synchronize other loops
Triggered loop
PID loop executes on rising edge
Brushed DC Servo Motor Control (FPGA).vi Block Diagram on CompactRIO Motor Control Basics.lvproj/FPGA Target
PID Clock

Figure 6.51. Each of these three independent loops is completely dedicated to its own task, resulting in the highest level of reliability.
Observations

- One loop is used to generate synchronization clocks used by other loops.
- The encoder function needs to run at full speed to avoid missing any digital pulses. This function runs at 40 MHz but latches the Position (Counts) and Velocity (Counts/Interval) signals to synchronize the data with the other loops.
- The PID function needs to run at a specific speed (20 kHz or 2000 ticks) and avoid any jitter in its timing. This is because the integral and derivate gains depend on the time interval, Ts. If the time interval varied, or if the same old value was passed several times into the function, then the integral and derivative gains are incorrect.
- In the bottom loop, you can see that the execution is triggered by the rising edge of the PID clock signal. Read a local variable for the signal in this SCTL and exit the loop when you detect a rising edge. Then execute the 32-bit PID algorithm that is included with the NI SoftMotion Development Module for LabVIEW. This reads the commanded position, compares it to the position measured by the encoder, and generates a command for the PWM loop. In this case, you are using a **Scale by Power of 2** function to divide the PID output signal by 2^-4, which is equivalent to dividing by 16. This scales the value to the ±2,000 ticks range needed by the PWM function. A value of 1,000 ticks is equal to a 50 percent duty cycle because the PWM period is 2,000 ticks.
- Note that the upper two loops are running at a 40 MHz loop rate. The lower loop is triggered to run at a 20 kHz loop rate by the PWM clock signal. (When triggered, the NI SoftMotion PID function takes 36 ticks to execute.)

Technique 5. Avoid "Gate Hogs"

Now that you understand four key techniques useful for developing LabVIEW FPGA code, consider one last technique: avoiding gate hogs. These are often innocent-looking code that eats up lots of your FPGA gates (also known as slices). Three of the most common offenders include the following:

Large Arrays or Clusters: Creating a large array or cluster with a front panel indicator or control is one of the most common programming mistakes that eats up lots of FPGA gates. If you do not need a front panel indicator for communication with the host processor, then do not create one. If you need to transfer more than a dozen or so array elements, use DMA instead as a way to pass the data. Also avoid using array manipulation functions like this **Rotate 1D Array** function whenever possible.

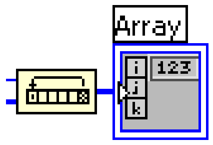


Figure 6.52. Avoid using array manipulation functions like this Rotate 1D Array function whenever possible.

Quotient and Remainder: This function implements integer division. The quotient output, **floor(x/y),** is **x** divided by **y,** rounded down to the closest integer. The remainder output, **x-y*floor(x/y)**, is whatever is left over. For example, 23 divided by 5 gives a quotient of 4 and a remainder of 3. This function is gate-intensive and takes multiple clock cycles to execute, so you cannot use it in an SCTL. Be sure to wire the minimum data type you need to the terminals when working with this function and use constants rather than controls when possible.

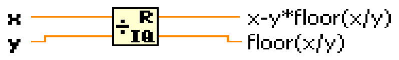


Figure 6.53. Quotient and Remainder Function

Scale by Power of 2: If the **n** terminal is positive, this function multiplies the **x** input by 2 to the power of n (2^**n**). If **n** is negative, the function divides by 2^**n**. For example, setting n to **+4** would multiply by **16**, while setting it to **-4** would divide by 16. This function is much more efficient than the **Quotient and Remainder** function. However, use a constant of the minimum data size needed for the **n** terminal whenever possible.

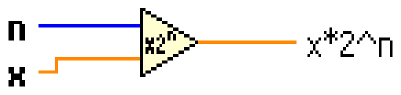


Figure 6.54. Scale by Power of 2 Function

Note: DMA is a better way to send an array of data to the host than creating a front panel indicator for the array and using the **FPGA Read/Write** method. Arrays are useful for collecting a set of simultaneously sampled data to be fed into a DMA buffer for transfer to the host computer. It is OK to use an array to collect the data points together for indexing into the DMA Write function as long as you do not create a front panel indicator for the array. Using autoindexing on the for loop used to write the data into the DMA buffer is fine as long as you do not create a front panel indicator for the array because the compiler does a good job of optimizing arrays passed into **for loops** for indexing purposes.

Tip: Avoid Front-Panel Arrays for Data Transfer

When optimizing your code for the amount of space it uses on the FPGA, you should consider the front panel controls and indicators you are implementing. Each front panel object and the data it represents takes up a significant portion of the FPGA space. By reducing the number of these objects and decreasing the size of any arrays used on the front panel, you can significantly reduce the FPGA space required by the VI.

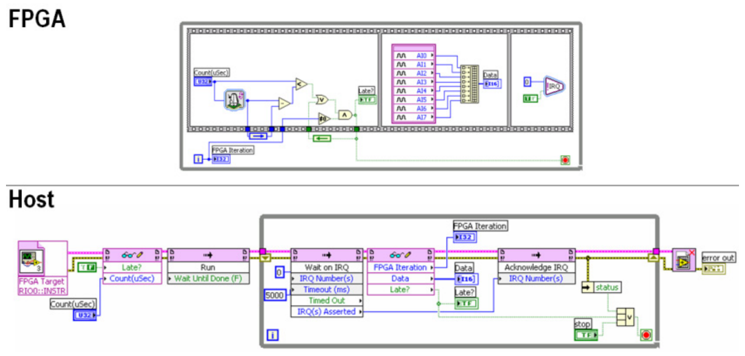


Figure 6.55. Creating Large Arrays to Store Data and Transfer It to the Host Application

Instead of creating large arrays to store data and transfer it to the host application (shown in Figure 6.55), use DMA to transfer an array of analog input samples to the host processor as shown in Figure 6.56.

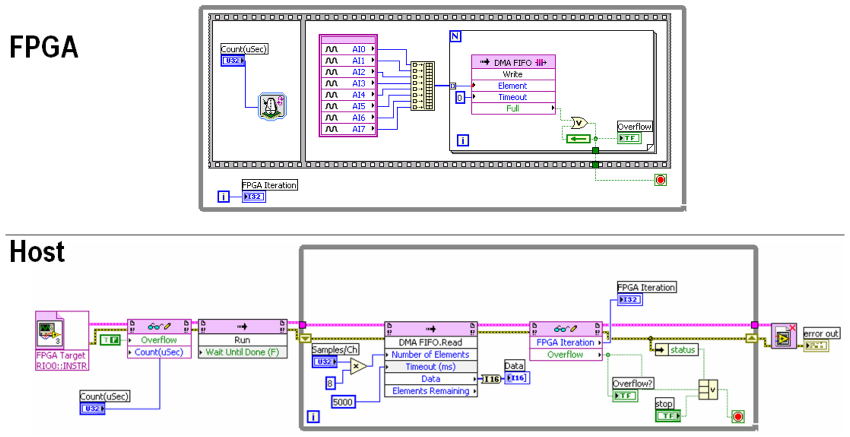


Figure 6.56. Using DMA to Transfer an Array of Analog Input Samples to the Host Processor

Tip: Use DMA for Data Transfer

DMA uses FPGA memory to store data and then transfer it at a high speed to host processor memory with very little processor involvement. This uses much fewer processor cycles when sending large blocks of data compared to the front panel indicator with the FPGA Read/Write method.

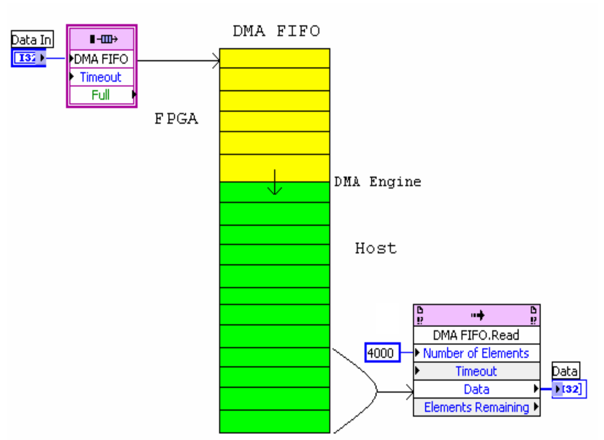


Figure 6.57. DMA uses FPGA memory to store data and then transfer it at a high speed to host processor memory with very little processor involvement.

Programming Tips for Implementing DMA

- When setting the FPGA buffer size, you can use the default size (1023). Creating a larger FPGA memory buffer typically does **not** have benefits.
- You should set the host buffer size to a larger value than the default size. By default, the host buffer size is two times bigger than the FPGA buffer. You should actually set it to at least two times the **Number of Elements** you plan to use.
- If you are passing an array of data, the **Number of Elements** input should always be an integer multiple of the array size. For example, if you are passing an array of eight elements, the **Number of Elements** should be an integer multiple of eight (such as 80, which gives 10 samples of eight elements each).
- Each DMA transaction has overhead, so reading larger blocks of data is typically better. The **DMA FIFO.Read function** automatically waits until the **Number of Elements** you requested become available, minimizing processor usage.
- Packing 16-bit channel data into a U32 (since DMA uses U32 data type) typically does not have benefits on CompactRIO because the PCI bus has very high bandwidth for sending DMA data, so you most likely are nowhere near to using up all of the bus bandwidth. Instead, typically the processor is the bottleneck in processing the data being streamed. Packing the data in the FPGA means it has to be unpacked on the processor, which adds more processor overhead. In general, you should send each channel as a U32 even if you are acquiring 16-bit data.
- The **Full** output on the DMA FIFO Write function is actually an error indicator. This should never occur under normal operation, so NI recommends you stop the application if you see this error and reset the FPGA before restarting.

Tip: Use the Minimum Data Type Necessary

Remember to use the minimum data type necessary when programming in LabVIEW FPGA. For example, using a 32-bit integer (I32) to index a case structure is probably overkill because it is unlikely that you are writing code for **2 billion different cases**. Usually an unsigned 8-bit integer (U8) does the trick because it works for up to 256 different cases.

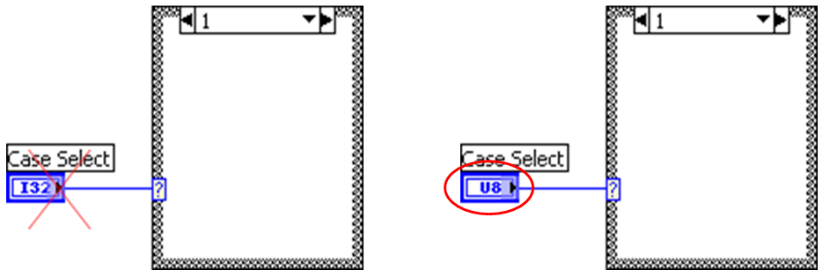


Figure 6.58. Use an unsigned 8-bit integer (U8) to index a case structure because it works for up to 256 different cases.

Tip: Optimize for Size

This FPGA application is too large to compile. Why?

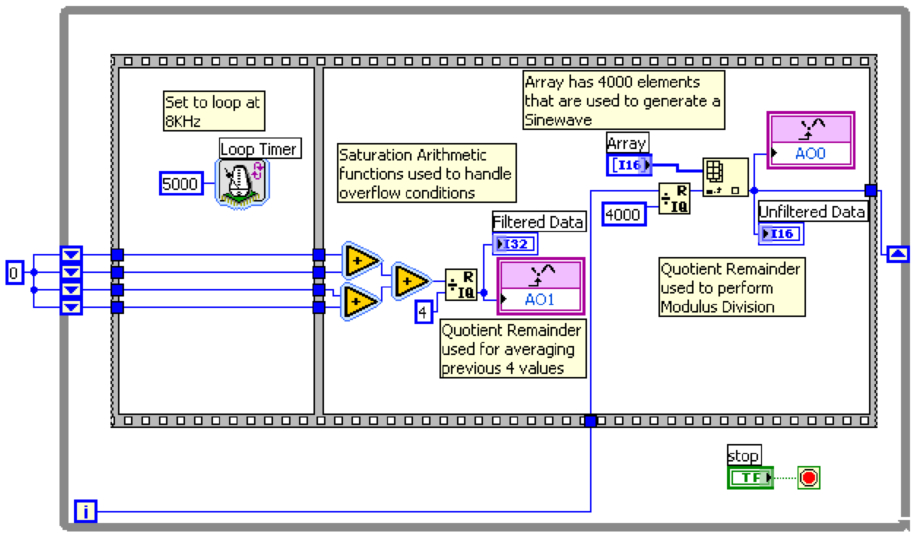

Set to loop at 8KHz
Loop Timer
5000
0
Saturation Arithmetic functions used to handle overflow conditions
Array has 4000 elements that are used to generate a Sinewave
Array
I16
AO0
4000
Unfiltered Data
I16
Filtered Data
I32
AO1
4
Quotient Remainder used for averaging previous 4 values
Quotient Remainder used to perform Modulus Division
stop
TF
i

Figure 6.59. This VI is too large to compile.

The application uses an array to store sine wave data. The array is indexed to get a value. In addition, four previous points are stored in shift registers. The previous four values are averaged. This VI is too large to compile. What can be done to help optimize this code?

Gate hogs found in the code: Large front panel arrays, Quotient and Remainder functions.

To improve the application, replace the array with an LUT as shown in Figure 6.60.

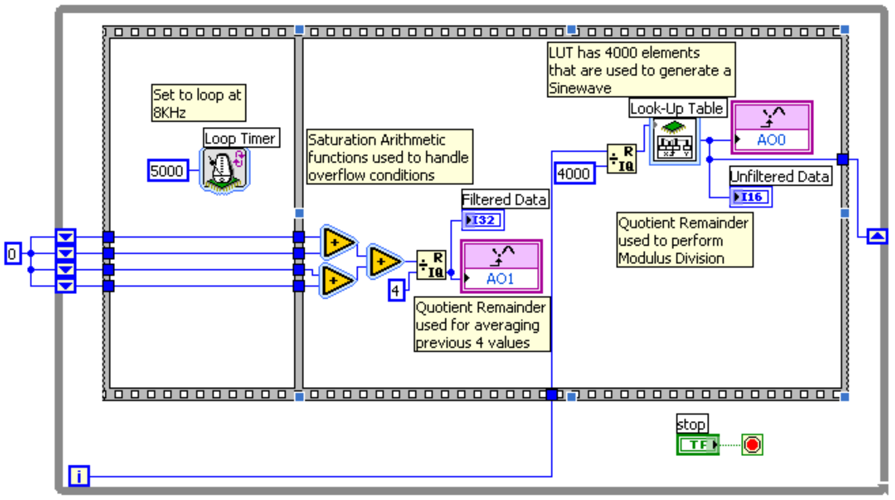

Set to loop at 8KHz
Loop Timer
5000
Saturation Arithmetic functions used to handle overflow conditions
LUT has 4000 elements that are used to generate a Sinewave
Look-Up Table
AO0
4000
Unfiltered Data
I16
Filtered Data
I32
AO1
4
Quotient Remainder used to perform Modulus Division
Quotient Remainder used for averaging previous 4 values
0
stop
TF
i

Figure 6.60. Improve the application by replacing the array with an LUT.

With this change alone, you can compile the program so it uses only 18 percent of a 1M gate FPGA. Can you further optimize the program?

Next remove both Quotient and Remainder functions. One of them was used to index through the LUT. This was replaced by a shift register counter operation, which is a very common technique in FPGA. The other Quotient and Remainder function was replaced by a scale by 2 to the power of n. Because the scale by 2 has a constant input, it uses very little FPGA space. Note: Scale by 2^ -2 is equal to dividing by 4.

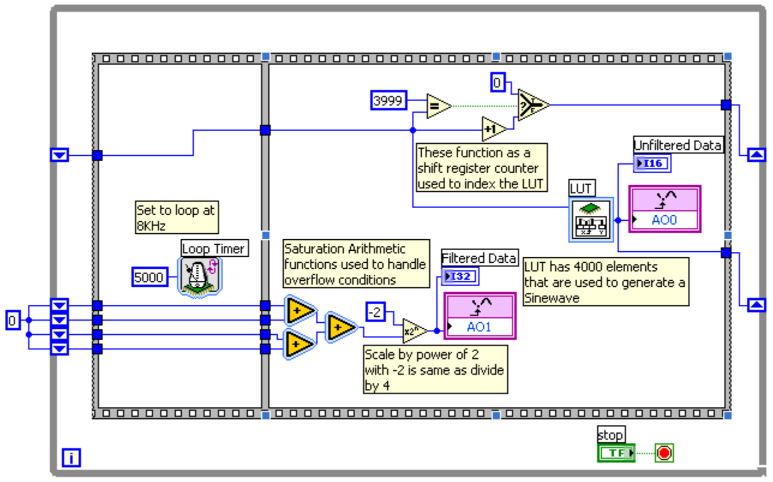
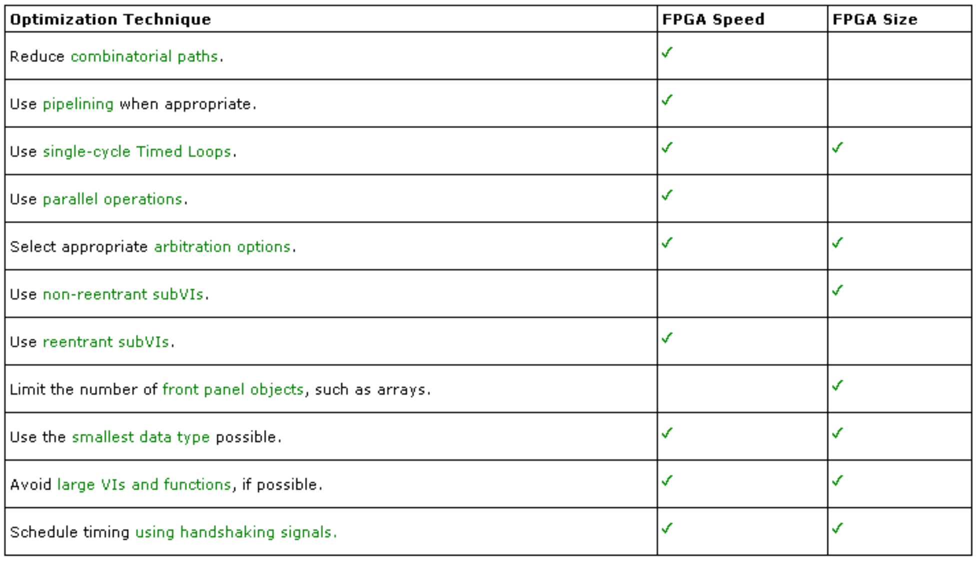

3999
0
These function as a shift register counter used to index the LUT
Unfiltered Data
I16
LUT
AO0
Set to loop at 8KHz
Loop Timer
5000
Saturation Arithmetic functions used to handle overflow conditions
Filtered Data
I32
LUT has 4000 elements that are used to generate a Sinewave
-2
AO1
Scale by power of 2 with -2 is same as divide by 4
0
stop
TF
i

Figure 6.61. Remove both Quotient and Remainder functions to further optimize the program.

Now the application takes only 9 percent of the FPGA gates.

Tip: Additional Techniques to Optimize Your FPGA Applications
For more information on this topic, view the "Optimizing FPGA VIs for Speed and Size" white paper in NI Developer Zone. It contains detailed information on more than 10 techniques you can use to optimize your LabVIEW FPGA applications.



Optimization Technique	FPGA Speed	FPGA Size
Reduce combinatorial paths.	✓	
Use pipelining when appropriate.	✓	
Use single-cycle Timed Loops.	✓	✓
Use parallel operations.	✓	
Select appropriate arbitration options.	✓	✓
Use non-reentrant subVIs.		✓
Use reentrant subVIs.	✓	
Limit the number of front panel objects, such as arrays.		✓
Use the smallest data type possible.	✓	✓
Avoid large VIs and functions, if possible.	✓	✓
Schedule timing using handshaking signals.	✓	✓

Table 6.3. More FPGA Application Optimization Techniques

CHAPTER 7
Creating a Networked User Interface to CompactRIO

Building User Interfaces and HMIs Using LabVIEW

This document shows one software design architecture for building an operator interface with a scalable navigation engine for cycling through different HMI pages.

You can use this architecture to build an HMI based on LabVIEW for any HMI hardware targets including the NI TPC-2512 touch panel computer running the Windows XP Embedded OS or the NI TPC-2106 running the Windows CE OS and the NI PPC-2115 panel PC running the Windows XP OS. LabVIEW is a full programming language that provides one solution for a variety of development tasks ranging from HMI/SCADA systems to reliable and deterministic control applications. The LabVIEW Touch Panel Module offers a graphical programming interface that you can use to develop an HMI in a Windows development environment and then deploy it to a National Instruments touch panel computer (TPC) or any HMIs running Windows CE. This document offers a framework for engineers developing HMIs based on both Windows Vista/XP and Windows CE.

Basic HMI Architecture Background

An HMI can be as simple or complex as the functionality you require. Choosing the right software architecture defines the functionality of an HMI as well as its ability to expand and adapt to future technologies. A basic HMI has three main routines:

1. Initialization and shutdown (housekeeping) routines
2. I/O scan (memory table and I/O and comm drivers) loop
3. Navigation (user interface) loop

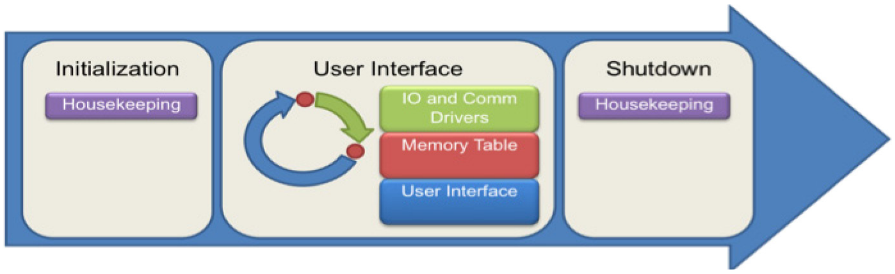


Figure 7.1. Routines for a Typical HMI

Initialization and Shutdown

Before executing the I/O scan loop and the navigation loop, the HMI needs to perform an initialization routine. This initialization routine sets all controls, indicators, internal variables, and variables communicating with the hardware (controller) to default states. You can add more logic to prepare the HMI for operations such as logging files. Before stopping the system, the shutdown task closes any references and performs additional tasks such as logging error files.

I/O Scan Loop
The overall HMI architecture is very similar to the CompactRIO architecture and uses both a memory table and a command based architecture to pass data between tasks.

Memory Table
In this architecture, an I/O scan loop reads and writes data from the network and puts it into a memory table. Other tasks, such as UI pages read and write data from the memory table. The memory table is created using single-process shared variables and the I/O scan reads data from network-published shared variables. This minimizes overhead and creates a more scalable application.

Command-Based Architecture
The I/O Scan also runs a command handler that transfers user commands into network commands for the real-time system. To ensure reliable operation, every user command on an HMI needs to be reliably sent to the real-time controller. To do this a command-based architecture with a command FIFO is used.

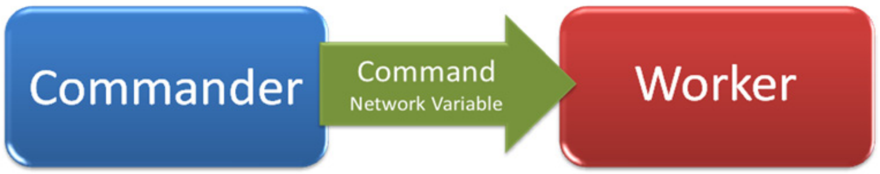


Figure 7.2. Simple Command Architecture Using Network Variables

A network-published shared variable with buffering enabled creates a FIFO and is used as the transport layer for the commands from the HMI to the controller. A LabVIEW queue is used to pass commands from the user interface pages to the command handler in the IO Scan.

Refer to the "Communication" section for more information network commands and on building the worker architecture for receiving the parsing the commands from the HMI.

Navigation Loop
This is the loop responsible for managing UIs (display pages). It includes the UI pages and the navigation engine that helps you cycle through the different pages available in the HMI.

UI Pages
Each UI page is a LabVIEW VI. The most common components in a UI page include the following: navigation buttons, action buttons, numeric indicators, graphs, Boolean controls and indicators, and images. The UI page reads the values to be displayed from the single-process shared variables.

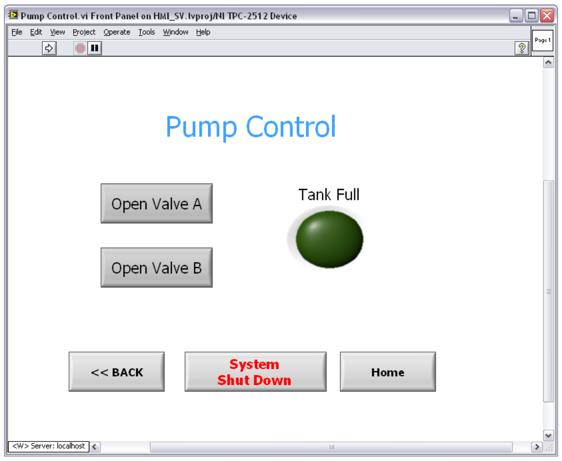

Pump Control.vi Front Panel on HMI_SV.lvproj/NI TPC-2512 Device
File Edit View Project Operate Tools Window Help
Pump Control
Open Valve A
Open Valve B
Tank Full
<< BACK
System Shut Down
Home
<W> Server: localhost

Figure 7.3. Example of a Typical UI Page in LabVIEW

User actions, such as pushing a button, are commands for either the navigation engine or for the real-time controller. You capture the commands using a LabVIEW event structure either and send them to the navigation engine or use a queue to pass them to a command handler that puts them onto the network. To aid in creating an event driven UI you can use the standard LabVIEW UI-handling templates. Translate the UI event into the appropriate command by passing data into the queue to be handled by the command handler.

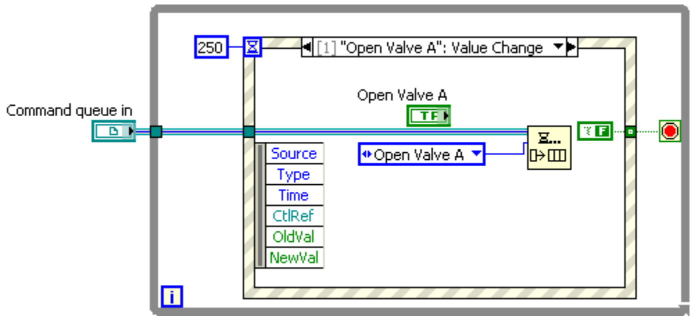


Figure 7.4. A User Interface Page Passing Data into the Command Queue

Navigation Engine

The navigation engine is a sub architecture that is part of the navigation loop and is in charge of managing the UI pages. It is a LabVIEW state machine built with a while loop and a case structure. Each case encloses a UI page, which is a LabVIEW VI set to open when called.

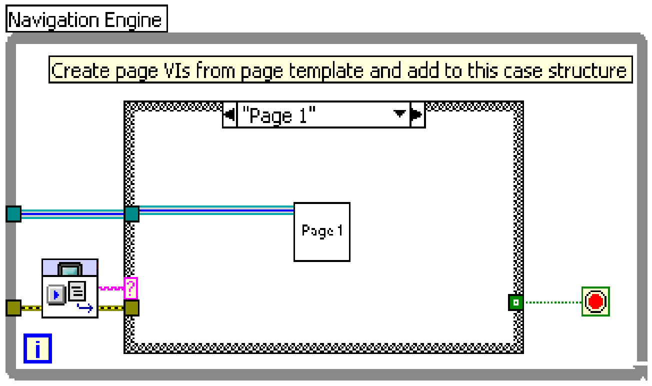

Navigation Engine
Create page VIs from page template and add to this case structure
"Page 1"
Page 1

Figure 7.5. Block Diagram of the Navigation Loop

The navigation engine uses VIs from the Touch Panel Navigation Palette.

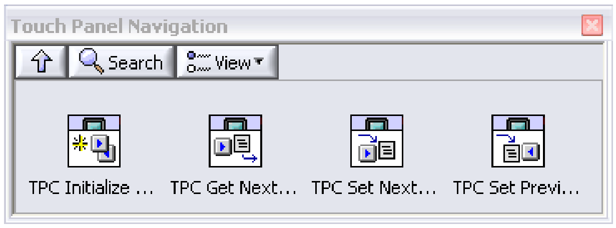


Figure 7.6. The Navigation Palette

TPC Initialize Navigation

This VI initializes the navigation engine by setting the navigation history depth and the name of the first page to be displayed.

TPC Get Next Page

This VI returns the name of the next page and passes it to the case structure in the HMI navigation engine.

TPC Set Next Page

This VI sets the name of the next page to be displayed. Use it within HMI pages to support navigation buttons.

TPC Set Previous Navigation Page

This VI returns the name of the previous page from the page history. Use it within HMI pages to support the operation of a "back" navigation button.

The page state and history are stored in a functional global variable that each of these VIs accesses.

Basic HMI Architecture for Windows XP, XP Embedded, and CE Operating Systems

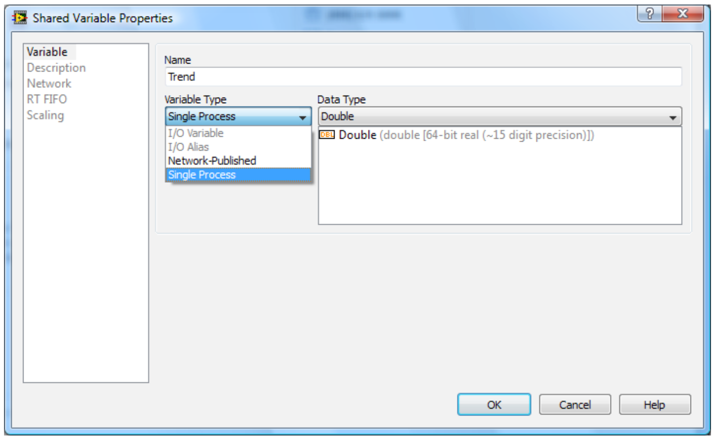
Windows XP and XP Embedded are recommended for full-featured HMI applications. The graphics capabilities and LabVIEW flexibility are much greater on XP and XP Embedded. Additionally, you can use Windows XP to more easily integrate other I/O or communicate with devices using other mechanisms such as OPC. However to provide easy scalability the overall architecture is the same for Windows XP, XP Embedded, and CE platforms.

To demonstrate this architecture, build a basic HMI application. This application contains two UI pages – a home page and a pump control page. Each of these UI pages has navigation buttons that take you to the other page. In addition to the navigation button, the UI pages have a control and indicators, with values being read from the controller through the network-published shared variables in the I/O scan loop and stored locally in single-process shared variables. You can use this framework to add many more pages that scale well with your application.

I/O Table

All process data is stored in a memory table on the HMI. This memory table is built using single-process shared variables and can be read and written by the IO scan or by individual UI pages. To create the I/O table, complete the following actions:

- Add a touch panel target to the LabVIEW Project.
- Underneath the touch panel target, create a subVI folder. Inside this folder, create an I/O Memory Table folder.
- Right-click on the folder and select **New»Variable**.
- Create your variables as single-process variables. Because the touch panel does not run a real-time operating system these variables do not have the real-time FIFO enabled.



Name: Trend
Variable Type: Single Process
Data Type: Double
Double (double [64-bit real (~15 digit precision)])

Figure 7.7. Creating Single-Process Variables

- Save your project and your library.

Initialization Task

When the HMI starts-up this task sets the controls, indicators, and internal variables to default states. It also creates the queue for internal commands.

1. Open a top-level VI and create a flat sequence structure.
2. In the first frame create a subVI and save it as Default HMI.
3. Open the subVI and give default values for the memory table by writing to the single-process shared variable. Save the changes.

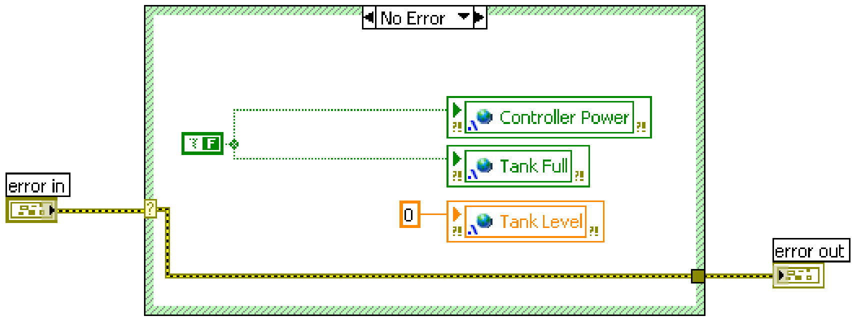


Figure 7.8. Assigning Default Values to Single-Process Shared Variables

4. The initialization task is also where the command queue is created so that any commands that need to be sent over to the controller are queued up without missing any commands. To make your programming more scalable make the data type to be enqueued a type def enum. This way if you need to add commands, the changes automatically propagate through all of your code.

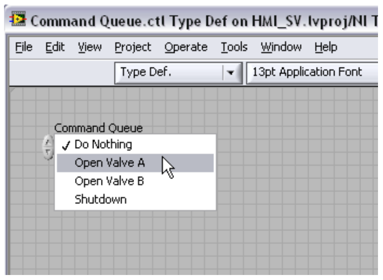

Command Queue.ctl Type Def on HMI_SV.lvproj/NI T
File Edit View Project Operate Tools Window Help
Type Def.
13pt Application Font
Command Queue
✓ Do Nothing
Open Valve A
Open Valve B
Shutdown

Figure 7.9. A type def enum provides scalability.

5. In the first frame also create a command queue and wire the type def enum you just created.

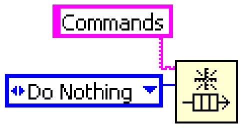


Figure 7.10. Creating an Enum Queue for Commands

I/O Scan Loop

The I/O scan loop transfers data and commands between the network and the memory table and command queue. To create the scan loop, complete the following actions:

1. In the second frame of the sequence structure, drop a while loop on the diagram. Use the **Wait(ms)** function to configure the communication loop rate to a rate that matches your application needs.
2. On the diagram, create two subVIs: Network Communication.vi and HMI Command Handler.vi.

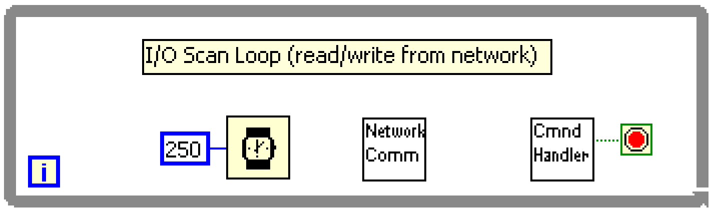


Figure 7.11. I/O Scan Loop

3. Open the Network Communication subVI. On the block diagram, drop the single-process shared variables and write them to the appropriate network-published shared variables (hosted on the CompactRIO controller). In this example , implement simple error checking so that if any error or warnings come off the network variable, you do not place any data in the memory table (single-process shared variable).

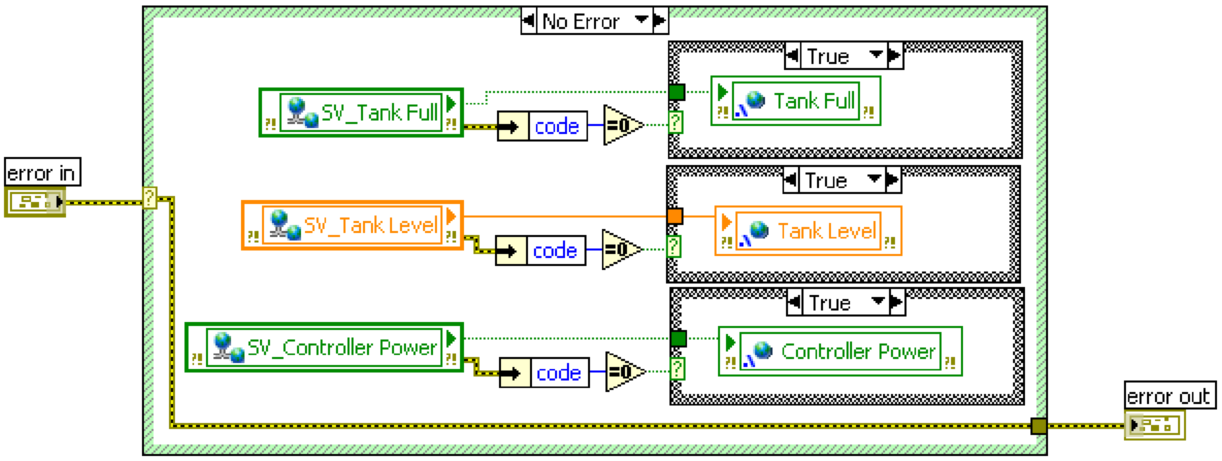


Figure 7.12. Network data is inserted into the memory table.

4. Open the HMI Command Handler subVI. This VI is in charge of sending any commands onto the network for the real-time controllers. The commands that are queued in UI pages are dequeued, and the respective commands sent over to the controller using the "Command" network variable are hosted on the real-time controller. Create a block diagram as shown in Figure 7.13, with one case for every command to be sent and a default case that is a "Do Nothing" command.

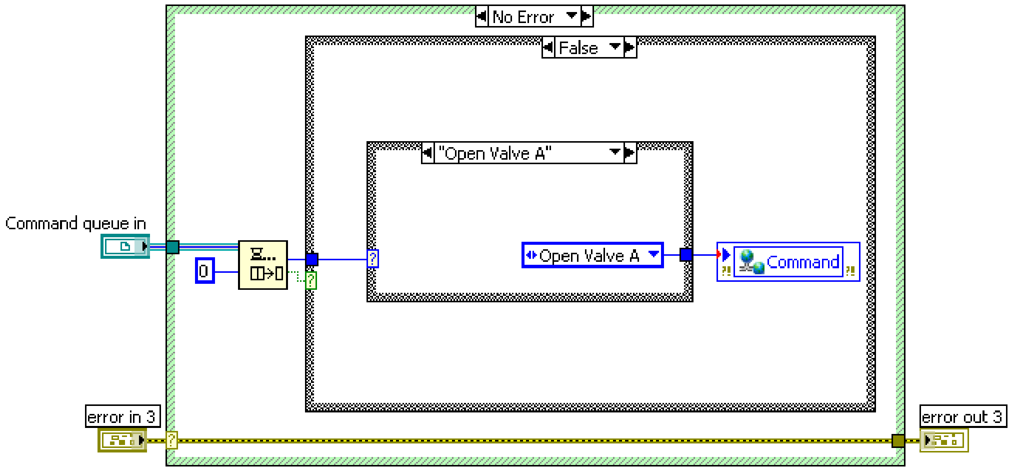


Figure 7.13. Command Handler SubVI

5. Save and close the two subVIs.

6. Wire up the initialization subVI and scan loop. In your application, the command handler also stops the loop if it receives a shutdown command.

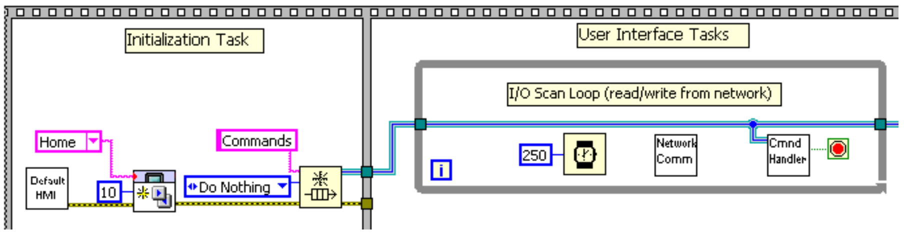


Figure 7.14. The I/O Scan and Initialization Task

Navigation Loop

The navigation loop consists of the navigation engine and the UI pages.

Navigation Engine

Use the navigation engine state machine architecture to manage the UI pages. This state machine uses an API that is shipped with the LabVIEW Touch Panel Module 8.6 and later. If you are working with a previous version you can download the VIs from **ni.com**.

The easiest way to write the navigation engine is to copy the code from an example.

1. Open the template VI located at C:\Program Files\National Instruments\LabVIEW 8.6\examples\TouchPanel\navigation\Design Pattern Template

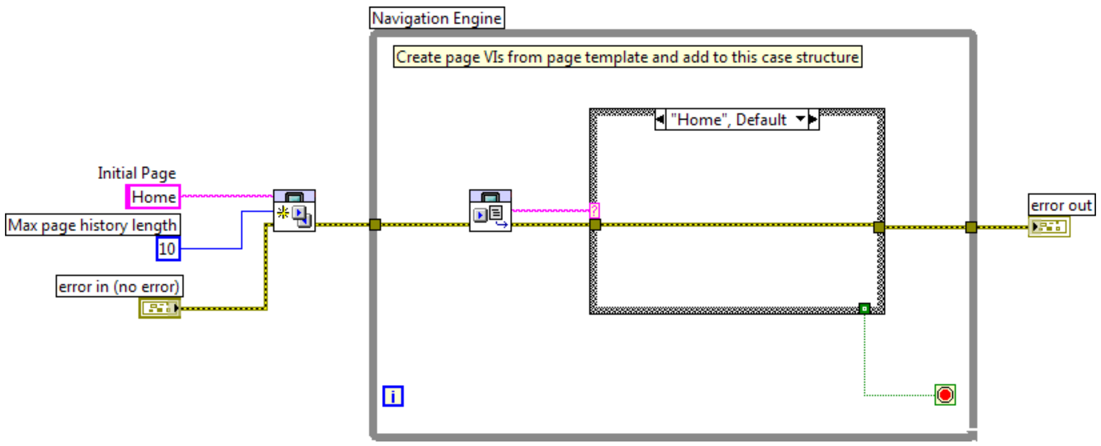


Figure 7.15. Template VI

2. Copy the entire block diagram from the template and paste it in the second sequence as a parallel loop to the scan loop.

3. Create one case for each UI page and one more case for the shutdown.

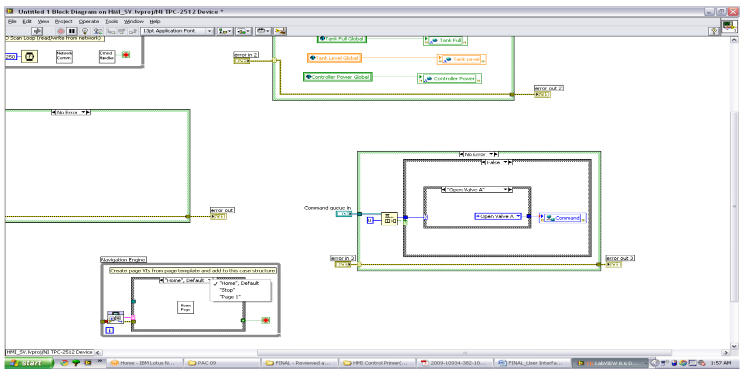


Figure 7.16. Adding Cases to Your HMI

4. Decide which page you want to open when the HMI starts for the first time. In this application, the UI page in the "Home" is the page that acts as the startup page. Use the TPC Initialize Navigation.vi to set the startup page as the home page before entering the while loop.

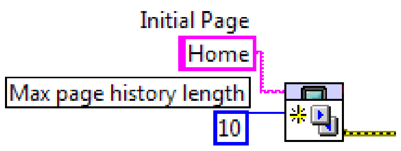


Figure 7.17. TPC Initialize Navigation.vi

5. Go to the stop case and enqueue "Shutdown" to the command queue that was created in the initialization loop. This shuts down the controller and HMI.

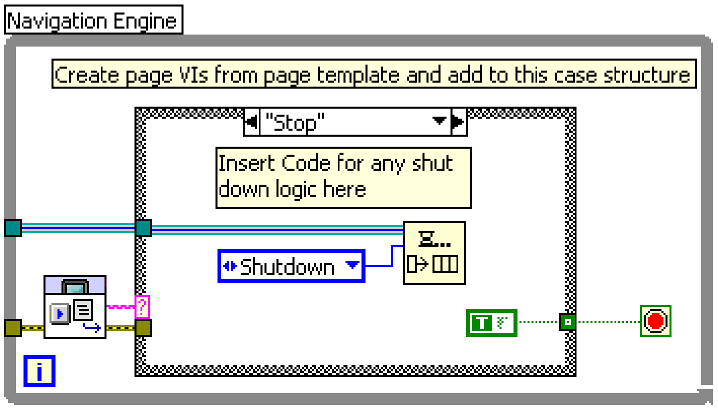

Navigation Engine
Create page VIs from page template and add to this case structure
"Stop"
Insert Code for any shut down logic here
Shutdown

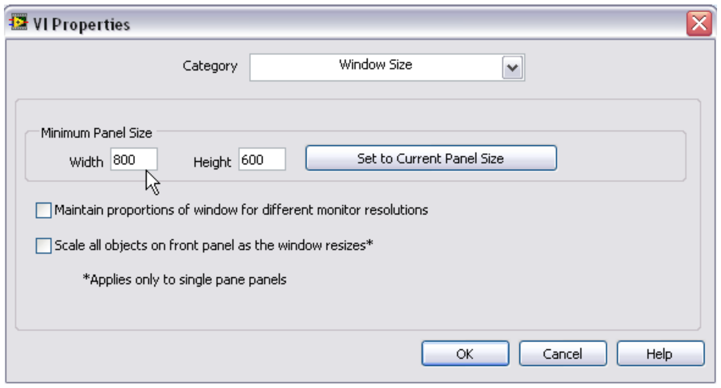
Figure 7.18. Queuing Up the Shutdown Command in the Command Queue

This completes the development of the navigation engine. The next step is to develop the UI pages.

UI Pages

The first thing to do when building an HMI page is to set the front panel dimensions. Because a front panel can be bigger than the HMI's display, it is important to constrain your UI elements to specific boundaries. For example, when using the NI TPC-2512 HMI, the size of the LabVIEW front panel needs to match the TPC's resolution, which is 800x600 pixels.

1. Open a new VI and set the dimensions based on the size of your hardware target. In this example, the VIs are set for the TPC-2512 target.



VI Properties

Category: Window Size

Minimum Panel Size

Width 800 Height 600 Set to Current Panel Size

☐ Maintain proportions of window for different monitor resolutions

☐ Scale all objects on front panel as the window resizes*

*Applies only to single pane panels

Figure 7.19. Set the front panel dimensions to your touch panel resolution.

2. The next step is to select the UI elements for implementing the functions controlled by the page. This example pump control page has some Boolean controls and indicators and some navigation buttons.

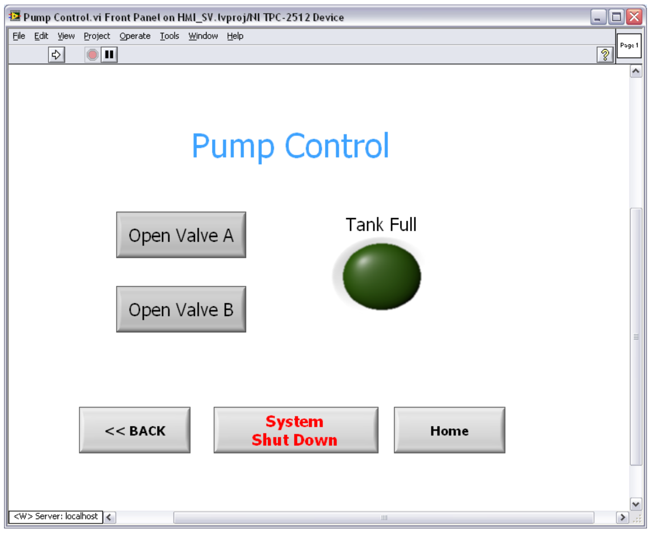

Pump Control.vi Front Panel on HMI_SV.lvproj/NI TPC-2512 Device
File Edit View Project Operate Tools Window Help
Pump Control
Open Valve A
Open Valve B
Tank Full
<< BACK
System Shut Down
Home
<W> Server: localhost

Figure 7.20. Example UI Page

3. The Navigation buttons are Back, which takes you to the last page visited, and Home, which takes you to the UI page that is in "home" case of the navigation engine that you put together in the earlier section.

4. In the block diagram of the UI page, drop a while loop and an event structure. Wire 250 in the timeout input at the top left of the event structure. This means the timeout case is executed every 250 ms if no other user event is detected.

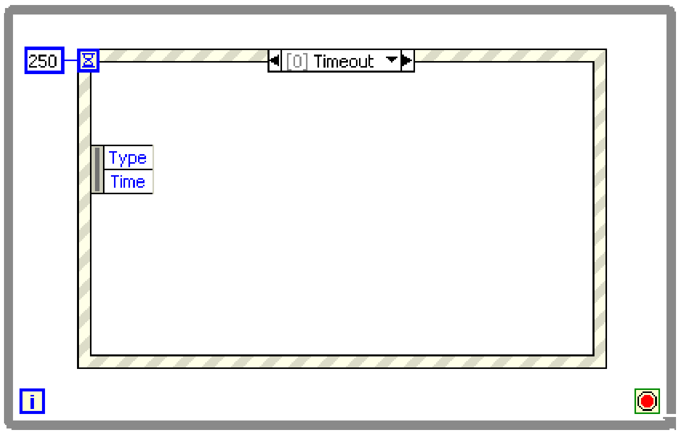


Figure 7.21. The timeout case is executed every 250 ms if no other user event is detected.

5. The timeout event case reads/writes the values from the single-process shared variables to the controls and indicators of this page. Drop the "Tank Full" single-process shared variable and wire it to the respective control. Note the "False" constant wired to the stop terminal of the while loop. This makes sure that the loop continues running if this case is executed.

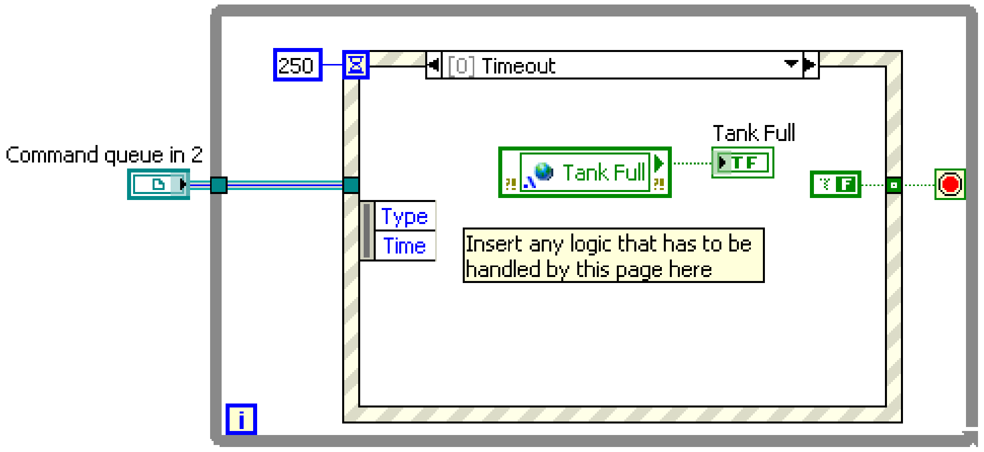
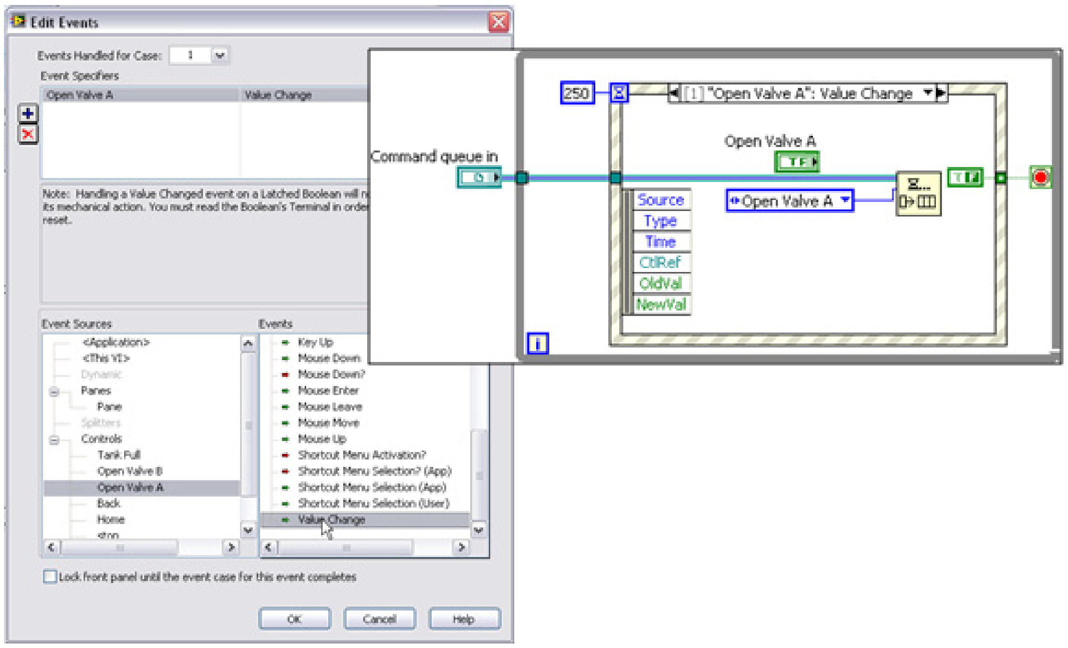


Figure 7.22. The "False" constant wired to the stop terminal of the while loop makes sure that the loop continues when the timeout event case is executed.

6. Now the commands "Open Valve A" and "Open Valve B" have to be enqueued in the command queue that was created in the initialization loop. Add event cases for to send commands if the value of the button changes. In the corresponding event case enqueue the specific command. Be sure to read the button control in the same event case so the button resets (if button is configured for latching mechanical action).



Edit Events

Events Handled for Case: 1

Event Specifiers

Open Valve A	Value Change

Note: Handling a Value Changed event on a Latched Boolean will r its mechanical action. You must read the Boolean's Terminal in order reset.

Lock front panel until the event case for this event completes

Figure 7.23. "Open Valve A" Command Enqueued in the Command Queue

7. Add event cases for each command or button.

8. In the Home event case, drop the Home Boolean button and the TPC Set Navigation Page from the Touch Panel Navigation Palette and wire in a string constant called Home. This sets the next page to be visited as the VI in the "Home" case of the navigation engine. Wire a "True" constant to the stop terminal of the while loop from this event case. This stops the loop, closes this UI, and returns to the navigation engine so that it can open the next page set by the user.

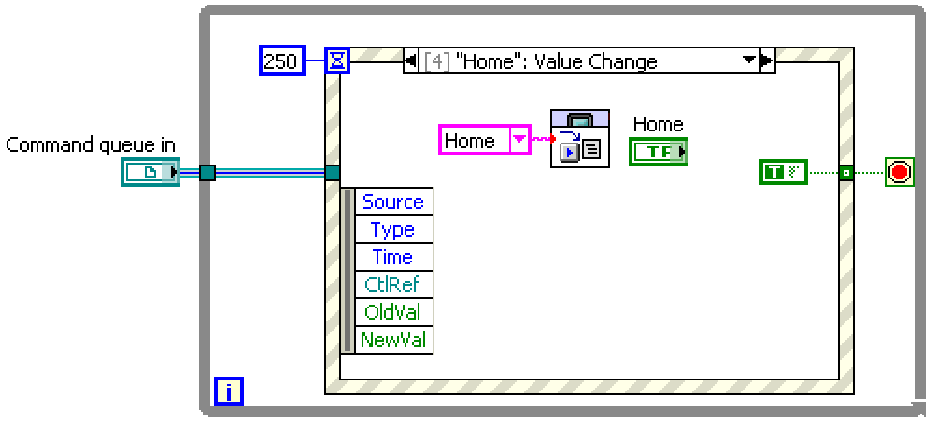


Figure 7.24. The "True" constant wired to the stop terminal terminates the loop and returns to the navigation engine so that it can open the next page.

9. For the event case of the Back button, use the TPC Set Previous Navigation Page.vi to go back to the last page visited and stop the VI.

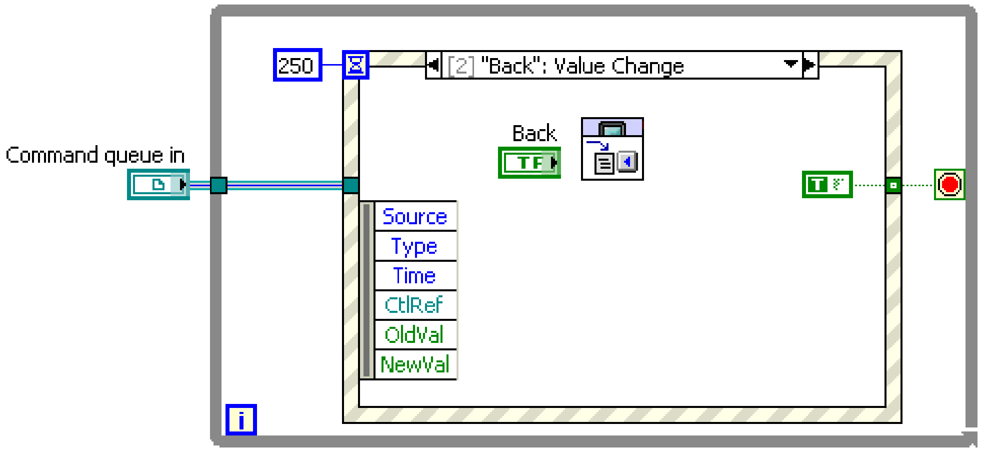


Figure 7.25. Using the TPC Set Previous Navigation Page.vi to Return to Last Page Visited

10. Finally, to the system shutdown case, wire in the "stop" case to the TPC Set Next Navigation Page.vi.

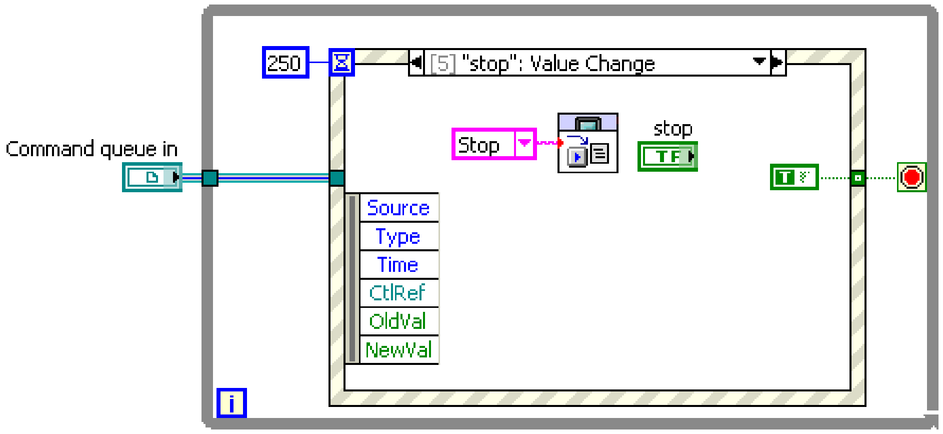


Figure 7.26. Wiring in the "Stop" Case to the TPC Set Next Navigation Page

11. Save this VI and drop it in the "Page 1" case structure of the navigation engine. This ensures that the page opens when the user presses the button to open the pump control page.

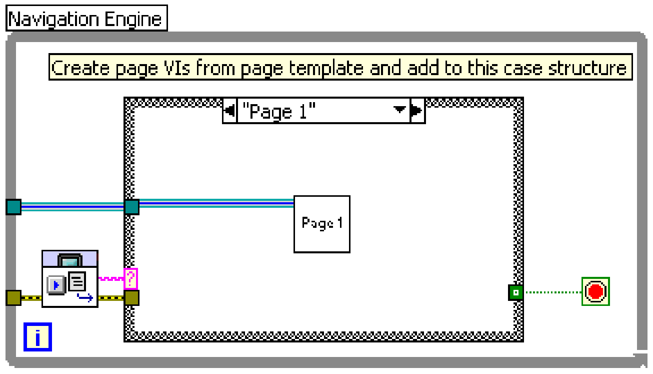
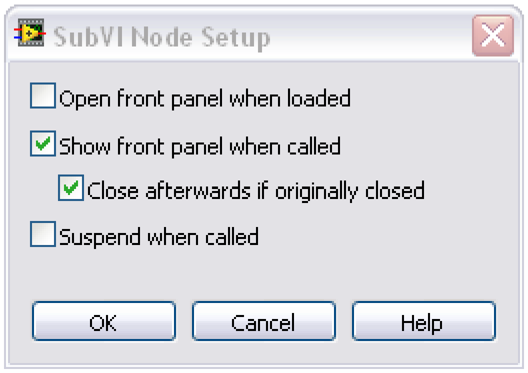

Navigation Engine
Create page VIs from page template and add to this case structure
"Page 1"
Page 1

Figure 7.27. Ensuring the Pump Control Page Opens When the User Presses the Button

12. Right-click on the subVI, access SubVI Node Setup, and check the options so the front panel appears when called and disappears when closed.


SubVI Node Setup

- [ ] Open front panel when loaded
- [✓] Show front panel when called
  - [✓] Close afterwards if originally closed
- [ ] Suspend when called

Figure 7.28. Checking SubVI Options

13. Repeat the steps in this section to construct VIs for the other UI pages.

14. This is the final application. Add any necessary shutdown logic such as closing the queue.

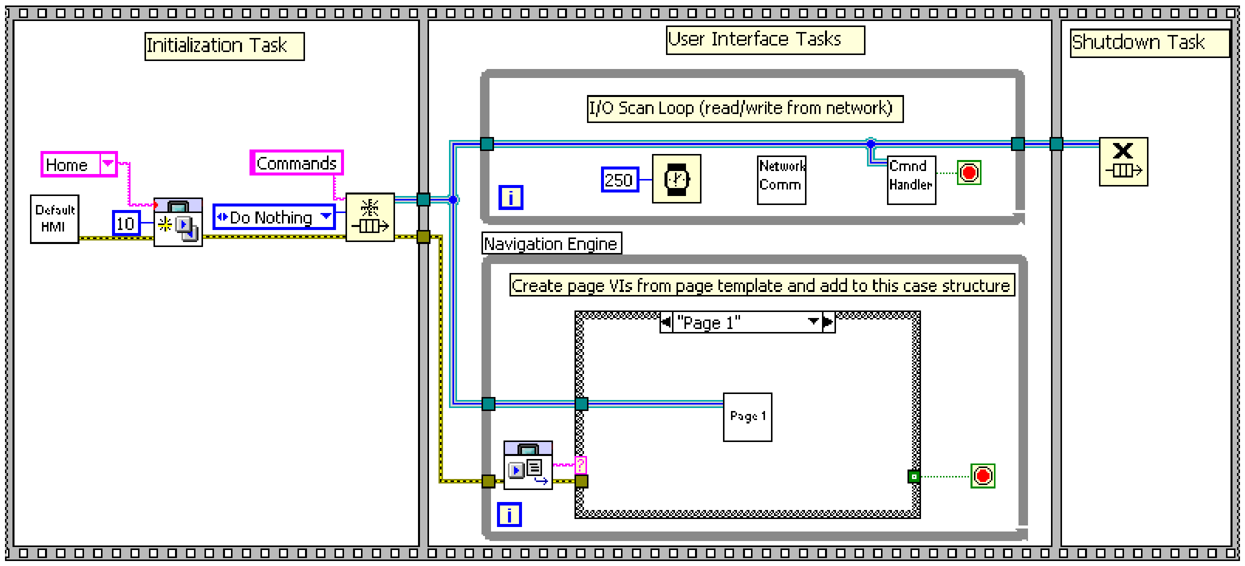


Figure 7.29. Scan Loop and Navigation Engine Initialization in a Sequence Structure

Getting Started – Modifying an Example

Example

LabVIEW Example Code
is provided for this section

The easiest way to get started with this design is to modify an existing example. To modify the previous example follow these main steps:

Step 1. Modify the Memory Table

1. Edit IO_Table.lvlib, which contains the single-process shared variables to reflect the process data you need to read or write from the UI pages.

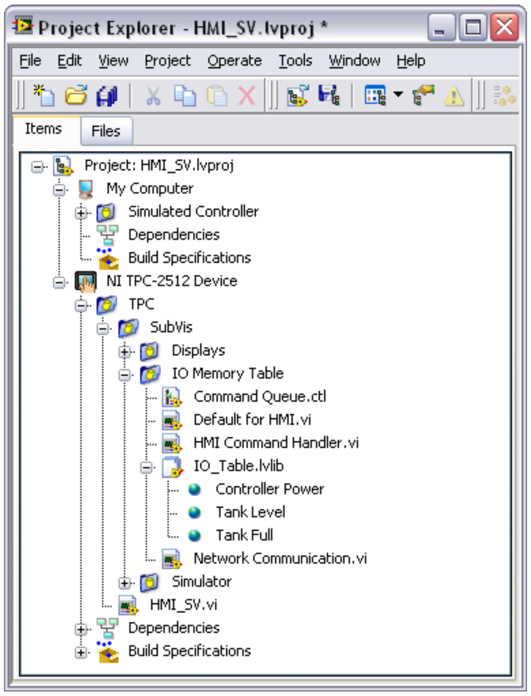


Figure 7.30. Add single-process shared variables to the IO_Table.lvlib.

2. Modify Default HMI.vi to write the default values for your controls and indicators.

Step 2. Modify the Command Type Def

1. Edit the command enum type def to represent the commands you need to pass within your application.

Step 3. Edit the I/O Scan Loop

1. Modify the I/O scan loop to read and write from the appropriate network variables and pass the data into the memory table (single-process shared variables).

2. Edit the command handler to read your UI commands and to send appropriate network commands to the command variable hosted on the CompactRIO.

Step 4. Edit the Navigation Loop

1. Add cases to the navigation engine to include more pages.

2. Create new VIs for each UI page and add them to the appropriate cases.

CHAPTER 8
Deploying and Replicating Applications

Application Deployment

All LabVIEW development for real-time targets and touch panel targets is done on a Windows PC. To run the code embedded on the targets you need to deploy the applications. Real-time controllers and touch panels, much like a PC, have both volatile memory (RAM) and nonvolatile memory (hard drive). When you deploy your code you have the option to deploy to either the volatile memory or nonvolatile memory.

Deployment onto Volatile Memory

If you deploy the application onto the volatile memory on a target, the application does not remain on the target after you cycle power. This is useful while you are developing your application and testing your code.

Deployment onto Nonvolatile Memory

If you deploy the application onto the nonvolatile memory on a target, the application remains after you cycle the power on the target. It is also possible to set applications stored on nonvolatile memory to start up automatically when the target boots. This is useful when you have finished code development and validation and want to create a stand-alone embedded system.

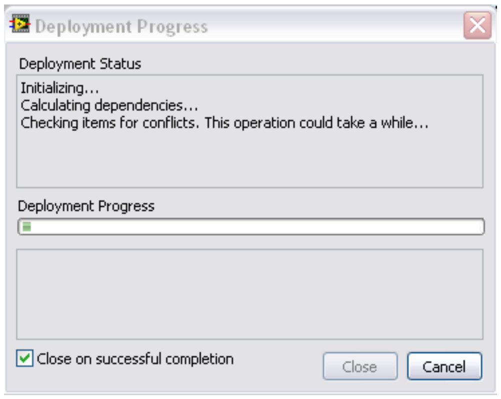
Deploying Applications to CompactRIO

Deploy a LabVIEW VI onto Volatile Memory

When you deploy an application into the nonvolatile memory of a CompactRIO controller, LabVIEW collects all the necessary files and downloads them over Ethernet to the CompactRIO controller. To deploy an application you need to

- Target the CompactRIO controller in LabVIEW
- Open a VI under the controller
- Press the "run" button

LabVIEW verifies that the VI and all subVIs are saved, deploys the code to the nonvolatile memory on the CompactRIO controller, and starts embedded execution of the code.



```
Initializing...
Calculating dependencies...
Checking items for conflicts. This operation could take a while...
```

Figure 8.1. LabVIEW Deploying an Application onto the Nonvolatile Memory of the Controller

Deploy a LabVIEW VI onto Nonvolatile Memory

Once you have finished developing and debugging your application, you likely want to deploy your code onto the nonvolatile memory on the controller so that it persists through power cycles and configure the system so the application runs on startup. To deploy an application onto the nonvolatile memory, you first need to build the VI into an executable.

Building an Executable from a VI

The LabVIEW Project provides the ability to build an executable real-time application from a VI. To build an executable real-time application, you create a build specification under the real-time target in the LabVIEW Project Explorer. By right-clicking on Build Specifications, you are presented with the option of creating a Real-Time Application along with a Source Distribution, Zip File, and so on.

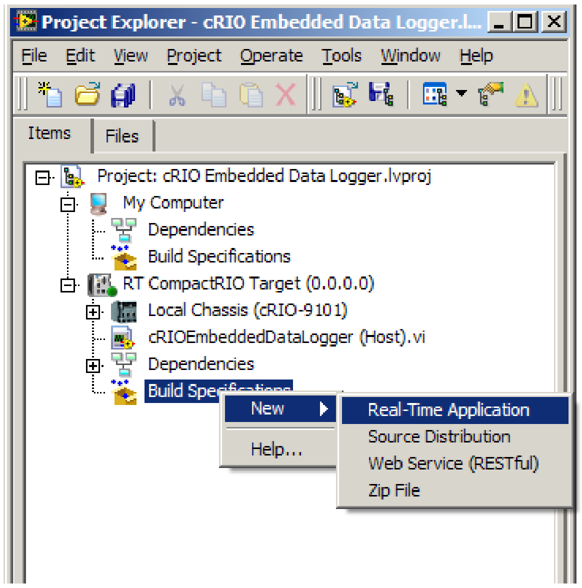
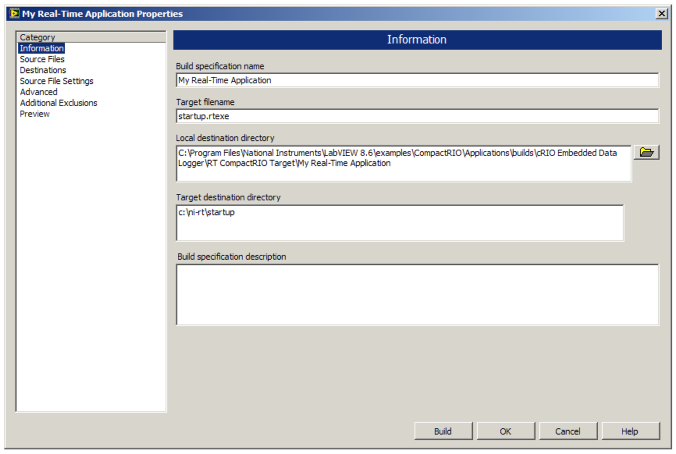


Figure 8.2. Create a new real-time application build specification.

After selecting **Real-Time Application,** you see a dialog box featuring two main categories that are most commonly used when building a real-time application: Information and Source Files. The Destinations, Source File Settings, Advanced, and Additional Exclusions categories are rarely used when building real-time applications.

The Information category contains the build specification name, executable filename, and destination directory for both the real-time target and host PC. You can change the build specification name and local destination directory to match your nomenclature and file organization. You normally do not need to change the target filename or target destination directory.



Build specification name
My Real-Time Application

Target filename
startup.rtexe

Local destination directory
C:\Program Files\National Instruments\LabVIEW 8.6\examples\CompactRIO\Applications\builds\cRIO Embedded Data Logger\RT CompactRIO Target\My Real-Time Application

Target destination directory
c:\ni-rt\startup

Build specification description

Figure 8.3. The Information Category in the Real-Time Application Properties

The Source Files category is used to set the startup VIs and include additional VIs or support files. You need to select the top-level VI from your Project Files and set it as a Startup VI. For most applications, a single VI is chosen to be a Startup VI. You do not need to include lvlib or set subVIs VIs as Startup VIs or Always Included unless they are called dynamically in your application.

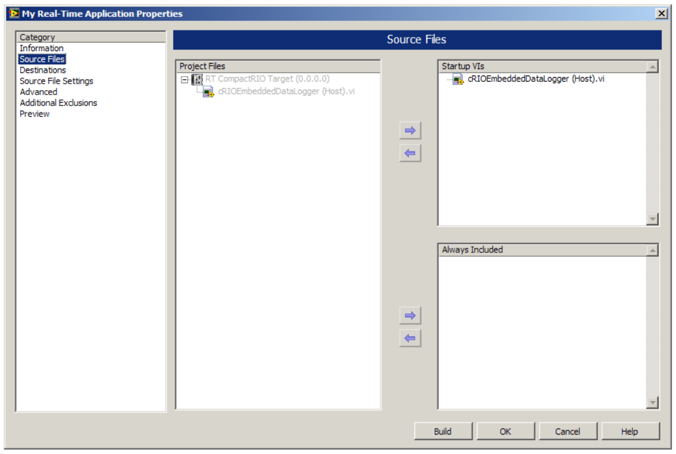


Figure 8.4. Source Files Category in the Real-Time Application Properties (In this example, the cRIOEmbeddedDataLogger (Host).vi was selected to be a Startup VI.)

After all options have been entered on the required category tabs, you can click OK to save the build specification or you can directly build the application by clicking the Build button. You can also right-click on a saved build specification and select **Build** to build the application.

When you build the application, an executable is created and saved on the hard drive of your development machine in the local destination directory.

etting an Executable Real-Time Application to Run on Startup

After an application has been built, you can set the executable to automatically start up as soon as the controller boots. To set an executable application to start up, you should right-click the Real-Time Application option (under Build Specifications) and select Set as startup. When you deploy the executable to the real-time controller, the controller is also configured to run the application automatically when you power on or reboot the real-time target. You can select Unset as Startup to disable automatic startup.

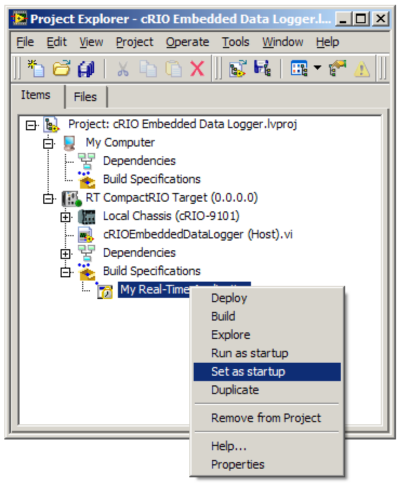
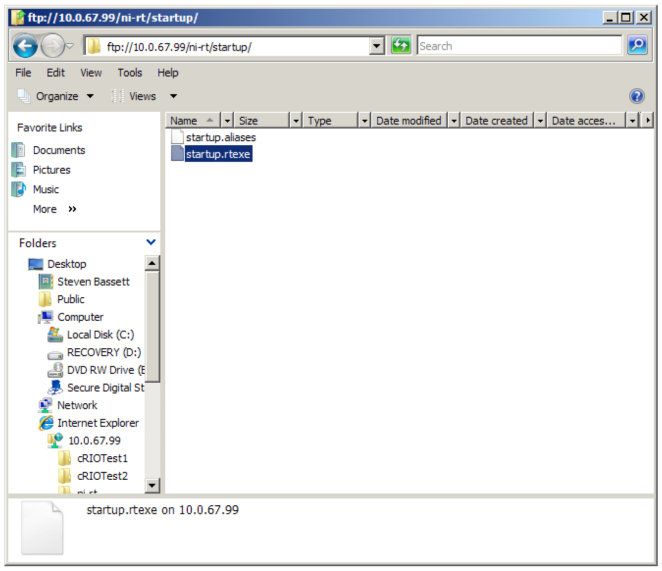

Project Explorer - cRIO Embedded Data Logger.l...
File Edit View Project Operate Tools Window Help
Items
Files
Project: cRIO Embedded Data Logger.lvproj
My Computer
Dependencies
Build Specifications
RT CompactRIO Target (0.0.0.0)
Local Chassis (cRIO-9101)
cRIOEmbeddedDataLogger (Host).vi
Dependencies
Build Specifications
My Real-Time
Deploy
Build
Explore
Run as startup
Set as startup
Duplicate
Remove from Project
Help...
Properties

Figure 8.5. Configuring a Build Specification to Run When an Application Boots

Deploy Executable Real-Time Application to the Nonvolatile Memory on a CompactRIO System

After configuring and building your executable, you now need to copy the executable and supporting files to the nonvolatile memory on the CompactRIO and configure the controller so the executable runs on startup. To copy the files and configure the controller, right-click on the Real-Time Application option and select Deploy. Behind the scenes, LabVIEW copies the executable files onto the nonvolatile memory on the controller and modifies the ni-rt.ini file to set the executable to run on startup. If you rebuild an application or change application properties (such as configuring it not to run on startup), you must redeploy the real-time application for the changes to take effect on the real-time target.

At some point, you may want to remove an executable you stored on your real-time target. The easiest way to do this is to use FTP to access the real-time target and delete the executable file that was deployed to the target. If you used the default settings, the file is located in the NI-RT\Startup folder with the name supplied in the target filename box from the Information category and the extension .rtexe.



startup.aliases
startup.rtexe

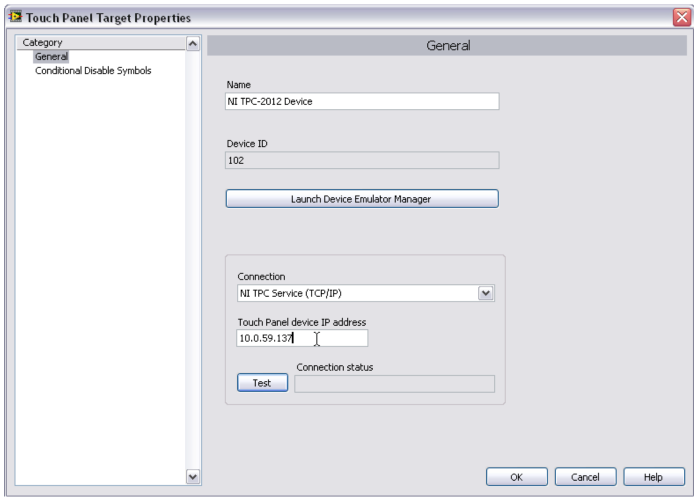
startup.rtexe on 10.0.67.99

Figure 8.6. Deleting the startup.rtexe from a CompactRIO Controller

Deploying Applications to a Touch Panel

Configure the Connection to the Touch Panel

Although it is possible to manually copy built applications to a touch panel device, it is recommended that you use Ethernet and allow the LabVIEW Project to automatically download the application. National Instruments touch panels all ship with a utility called the NI TPC Service that allows the LabVIEW Project to directly download code over Ethernet. To configure the connection right click on the touch panel target in the LabVIEW Project and select properties. In the General category choose the connection as NI TPC Service and enter the IP address of the touch panel. Test the connection to make sure the service is running.


Touch Panel Target Properties

Category: General; Conditional Disable Symbols

General

Name: NI TPC-2012 Device

Device ID: 102

Connection: NI TPC Service (TCP/IP)

Touch Panel device IP address: 10.0.59.137

Connection status:

Figure 8.7. Connect to a touch panel through Ethernet using the NI TPC Service.

You can find the IP address of the touch panel by going to command prompt on the TPC and typing ipconfig. To get to the command prompt go to the Start menu and select Run… In the popup window enter cmd.

Deploy a LabVIEW VI onto Volatile or Nonvolatile Memory

The steps to deploy an application onto a Windows XP Embedded touch panel and onto a Windows CE touch panel are nearly identical. The only difference is on an XP Embedded touch panel , you can deploy an application onto only the nonvolatile memory, and, on a Windows CE touch panel, you can deploy onto volatile or nonvolatile memory, depending on the destination directory you select. To run a deployed VI in either volatile or nonvolatile memory on a touch panel, you must first create an executable.

Building an Executable from a VI for an XP Embedded Touch Panel

The LabVIEW Project provides the ability to build an executable Touch Panel Application from a VI. To build an executable Touch Panel Application you create a build specification under the touch panel target in the Project Explorer. By right-clicking on Build Specifications, you can select the option of creating a Touch Panel Application, Source Distribution, Zip File, and so on.

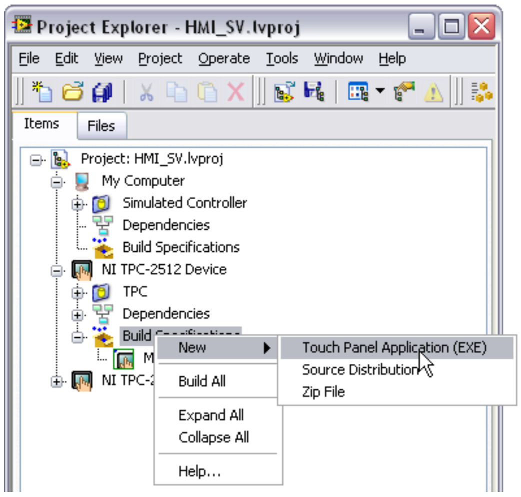
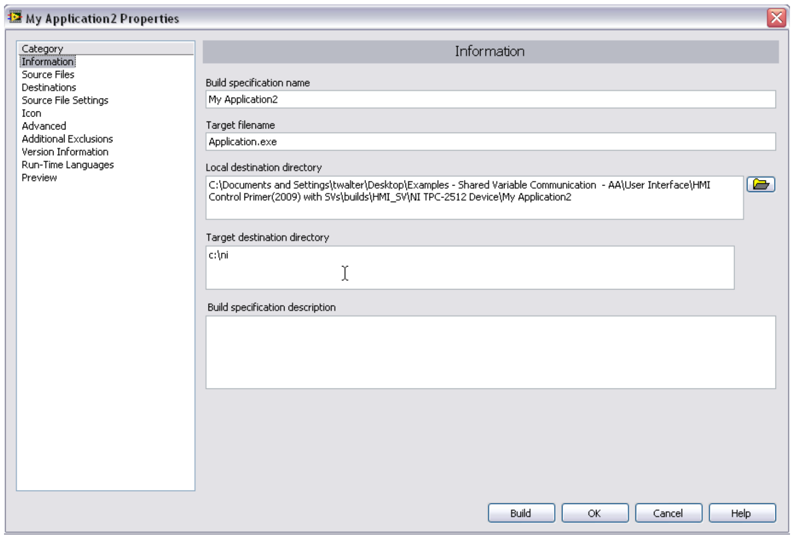


Figure 8.8. Create a touch panel application using the LabVIEW Project.

After selecting the **Touch Panel Application**, you are presented with a dialog box. The two main most commonly used categories when building a touch panel application are Information and Source Files. The other categories are rarely changed when building touch panel applications.

The Information category contains the build specification name, executable filename, and destination directory for both the touch panel target and host PC. You can change the build specification name and local destination directory to match your nomenclature and file organization. You normally do not need to change the target filename or target destination directory.



Build specification name
My Application2

Target filename
Application.exe

Local destination directory
C:\Documents and Settings\twalter\Desktop\Examples - Shared Variable Communication - AA\User Interface\HMI Control Primer(2009) with SVs\builds\HMI_SV\NI TPC-2512 Device\My Application2

Target destination directory
c:\ni

Build specification description

Figure 8.9. The Information Category in the Touch Panel Application Properties

The Source Files category is used to set the startup VIs and include additional VIs or support files. You need to select the top-level VI from your Project Files and set it as a Startup VI. For most applications, a single VI is chosen to be a Startup VI. You do not need to include lvlib or set subVIs VIs as Startup VIs or Always Included unless they are called dynamically in your application.

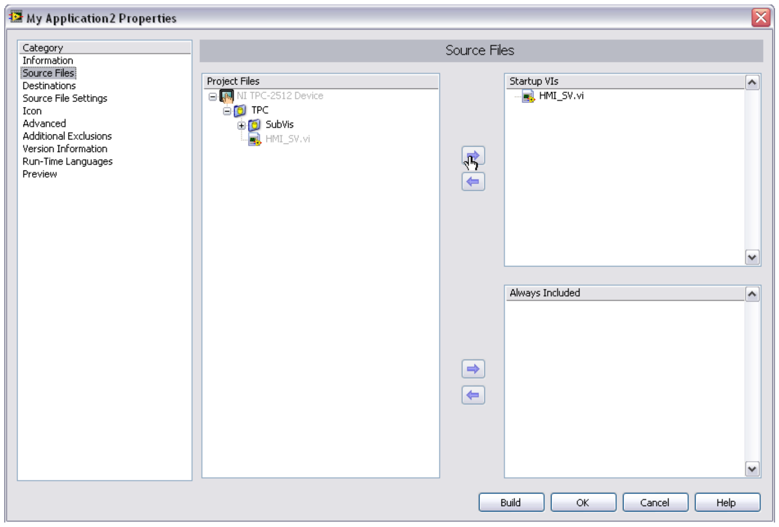


Figure 8.10. Source Files Category in the Touch Panel Application Properties (In this example, the HMI_SV.vi was selected to be a Startup VI).

After all options have been entered on the required category tabs, you can click OK to save the build specification or you can directly build the application by clicking the Build button. You can also right-click on a saved build specification and select **Build** to build the application.

When you build the application, an executable is created and saved on the hard drive of your development machine in the local destination directory.

Building an Executable from a VI for a Windows CE Touch Panel

The LabVIEW Project provides the ability to build an executable touch panel application from a VI. To build this application, you create a build specification under the touch panel target in the LabVIEW Project Explorer. By right-clicking on Build Specifications, you can select the option of creating a Touch Panel Application, Source Distribution, Zip File, and so on.

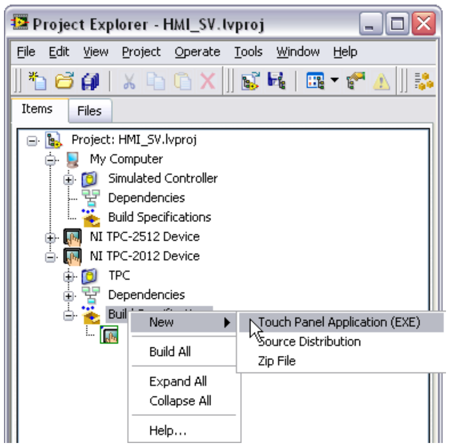
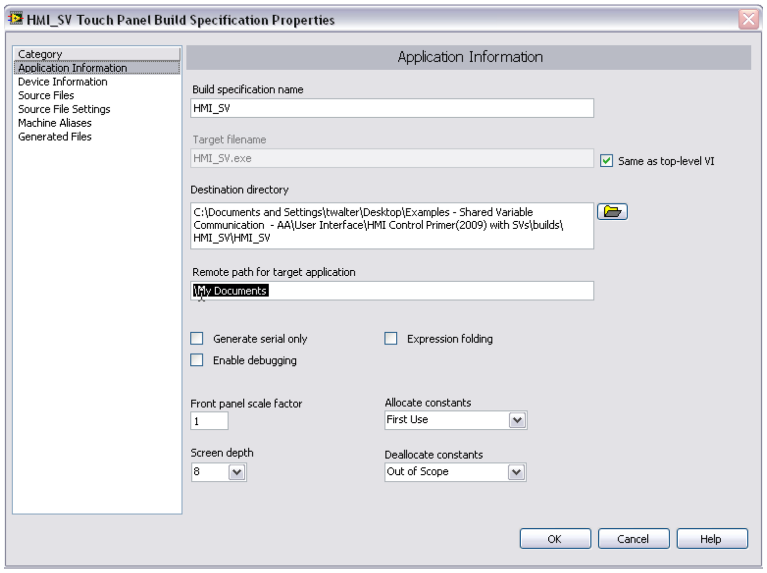


Figure 8.11. Creating a Touch Panel Application Using the LabVIEW Project

After selecting **Touch Panel Application**, you see a dialog box with the three main categories that are most commonly used when building a touch panel application for a Windows CE target: Application Information, Source Files, and Machine Aliases. The other categories are rarely changed when building Windows CE touch panel applications.

The Application Information category contains the build specification name, executable filename, and destination directory for both the touch panel target and host PC. You can change the build specification name and local destination directory to match your nomenclature and file organization. You normally do not need to change the target filename. The target destination determines if the deployed executable runs in volatile or nonvolatile memory. On a Windows CE device

- \My Documents folder is volatile memory. If you deploy the executable to this memory location, it does not persist through power cycles.
- \HardDisk is nonvolatile memory. If you want your application to remain on the Windows CE device after a power cycle, you should set your "remote path for target application" to a directory on the \HardDisk such as \HardDisk\Documents and Settings



HMI_SV Touch Panel Build Specification Properties

Category: Application Information, Device Information, Source Files, Source File Settings, Machine Aliases, Generated Files

Application Information

Build specification name: HMI_SV

Target filename: HMI_SV.exe [✓ Same as top-level VI]

Destination directory: C:\Documents and Settings\twalter\Desktop\Examples - Shared Variable Communication - AA\User Interface\HMI Control Primer(2009) with SVs\builds\HMI_SV\HMI_SV

Remote path for target application: \My Documents

☐ Generate serial only
☐ Expression folding
☐ Enable debugging

Front panel scale factor: 1

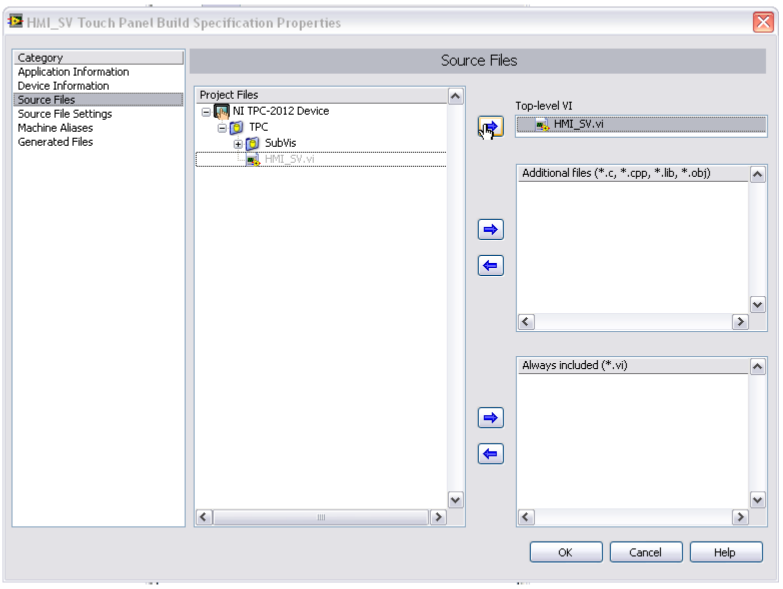
Allocate constants: First Use

Screen depth: 8

Deallocate constants: Out of Scope

Figure 8.12. The Information Category in the Touch Panel Application Properties

The Source Files category is used to set the startup VI and include additional VIs or support files. You need to select the top-level VI from your Project File. The top-level VI is the startup VI. For Windows CE touch panel applications, you can select only a single VI to be the top-level VI. You do not need to include lvlib or subVIs as Always Included.



HMI_SV Touch Panel Build Specification Properties

Category: Application Information, Device Information, Source Files, Source File Settings, Machine Aliases, Generated Files

Source Files

Project Files:
- NI TPC-2012 Device
  - TPC
    - SubVis
    - HMI_SV.vi

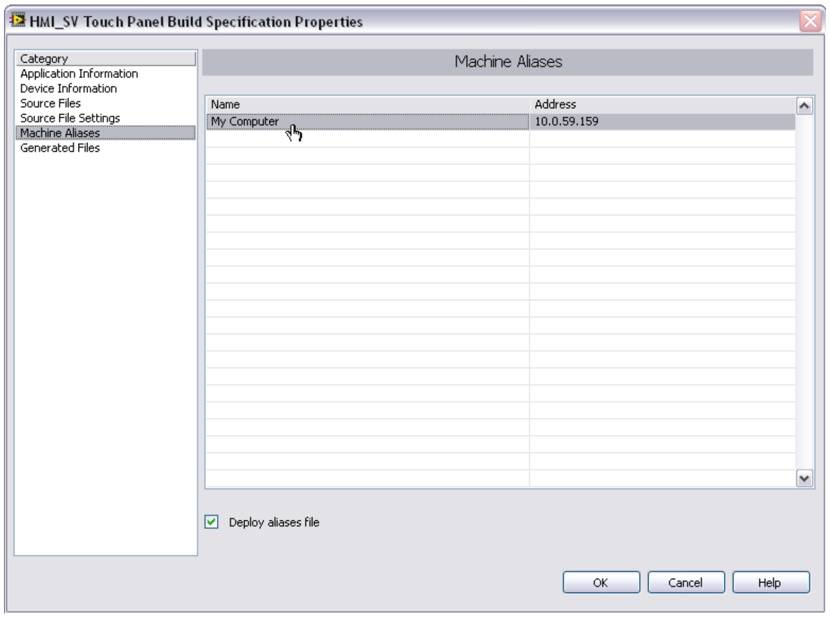
Top-level VI: HMI_SV.vi

Additional files (*.c, *.cpp, *.lib, *.obj)

Always included (*.vi)

Figure 8.13. Source Files Category in the Touch Panel Application Properties
(In this example, the HMI_SV.vi was selected to be the top-level VI.)

The Machine Aliases category is used to deploy an alias file. This is required if you are using network-published shared variables for communication to any devices. Be sure to check the "Deploy alias file" check box. The alias list should include your network-published shared variable servers and their IP addresses (normally CompactRIO or Windows PCs). More details on alias files and deploying applications using network-published shared variables are covered in the deployment section of this document.



Name	Address
My Computer	10.0.59.159

☑ Deploy aliases file

Figure 8.14. The Machine Aliases Category in the Touch Panel Application Properties
(Be sure to check the deploy aliases file check box if using network-published shared variables.)

After all options have been entered on the required category tabs, you can click OK to save the build specification or you can directly build the application by clicking the Build button. You can also right-click on a saved build specification and select **Build** to build the application.

When you build the application, an executable is created and saved on the hard drive of your development machine in the local destination directory.

Deploy an Executable Touch Panel Application to a Windows CE or XP Embedded Target

After configuring and building your executable you now need to copy the executable and supporting files to the memory on the touch panel. To copy the files, right-click on the Touch Panel Application and select Deploy. Behind the scenes, LabVIEW copies the executable files onto the memory on the touch panel. If you rebuild an application you must redeploy the touch panel application for the changes to take effect on the touch panel target.

The Run Button

If you click the run button on a VI targeted to a touch panel target, LabVIEW guides you through creating a build specification (if one does not exist) and deploys the code to the touch panel target.

Setting an Executable Touch Panel Application to Run on Startup

After you have deployed an application to the touch panel, you can set the executable so it automatically starts up as soon as the touch panel boots. Because you are running on a Windows system, you do this using standard Windows tools. In Windows XP Embedded, you should copy the executable and paste a shortcut into the Startup directory on the Start Menu. On Windows CE, you need to go to the STARTUP directory on the hard disk and modify the startup.ini file to list the path to the file (\HardDisk\Documents and Settings\HMI_SV.exe). You can alternatively use the Misc tab in the Configuration Utility (**Start»Programs»Utilities»Configuration Utilities**) to configure a program to startup on boot. This utility modifies the startup.ini file for you.

Deploying Applications that Use Network-Published Shared Variables

Network Shared Variable Background

The term **network shared variable** refers to a software item that exists on the network and can communicate between programs, applications, remote computers, and hardware.

There are three pieces that make the network variable work in LabVIEW.

Network Variable Nodes

A network variable node is the block diagram representation for network reads and writes. Each variable node references a software item on the network (the actual network variable) hosted by the Shared Variable Engine. Figure 8.15 shows a network variable node, its actual network path, and its respective item in the project tree.

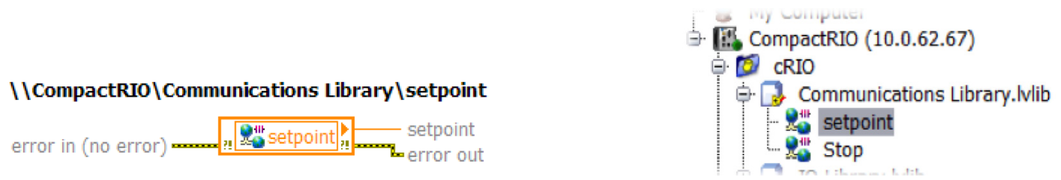


Figure 8.15. Network Variable Node and Its Network Path

Shared Variable Engine

The Shared Variable Engine is a software component that publishes data over Ethernet. The engine must run on real-time targets or Windows PCs where network shared variables are hosted. On Windows, the Shared Variable Engine is a service launched at system startup. On a real-time target, it is a driver that loads when the system boots.

When the shared variable engine starts it reads data stored on the nonvolatile memory to determine what variables it should publish on the network.

Publish-Subscribe Protocol (PSP)

The Shared Variable Engine uses the NI Publish-Subscribe Protocol (NI-PSP) to communicate data. The NI-PSP is a networking protocol built using TCP where each shared variable client subscribes to data hosted by a Shared Variable Engine.

Deploy Shared Variable Libraries to a Target That Hosts Variables

A CompactRIO system starts the Shared Variable Engine when it boots, and the engine accesses the nonvolatile memory to determine what if any libraries it needs to deploy. Shared variable libraries are automatically deployed when you run a VI that accesses any of the variables and when you deploy an application that accesses any of the variables. However, it is possible that no libraries have ever been deployed to the system. If this is the case, the engine does not make any variables available on the network.

You can choose from two methods to explicitly deploy a shared variable library to a target device.

1. You can target the CompactRIO system in the LabVIEW Project, place the library below the device, and deploy the library. This writes information to the nonvolatile memory on the CompactRIO controller and causes the variable engine to create new data items on the network.

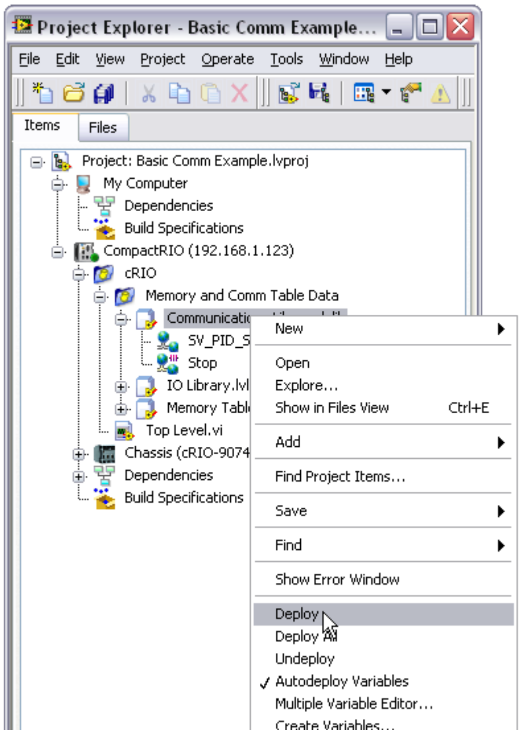


Figure 8.16. Deploy libraries to real-time targets by selecting Deploy from the right-click menu.

2. You can programmatically deploy the library from a LabVIEW application running on Windows using the Application invoke node.
   - On the block diagram, right click to bring up the programming palette and go to Programming » Application Control and place the Invoke Node on the block diagram
   - Using the hand tool, click on Method and select Library » Deploy Library

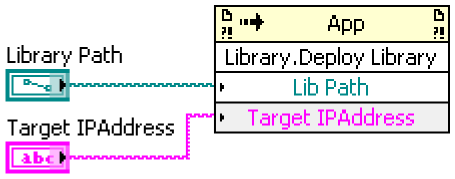


Figure 8.17 You can programmatically deploy libraries to real-time targets using the application invoke node on a PC.

   - Use the Path input of the Deploy Library invoke node to point to the library(s) containing your shared variables. Also specify the IP address of the real-time target using the Target IP Address input.

Undeploy a Network Shared Variable Library

Once a library is deployed to a Shared Variable Engine, those settings persist until you manually undeploy them. To undeploy a library:

1. Launch the NI Distributed System Manager (From **LabVIEW»Tools** or from the Start Menu)
2. Add the real-time system to "My Systems" (Actions»Add System to My Systems)
3. Right-click on the library you wish to undeploy and select Remove Process

Deploy Applications That Are Shared Variable Clients

Running an executable that is only a shared variable client (not a host) does not require any special deployment steps to deploy libraries. However the controller does need a way to translate the name of the system that is hosting the variable into the IP address of the system that is hosting the variable.

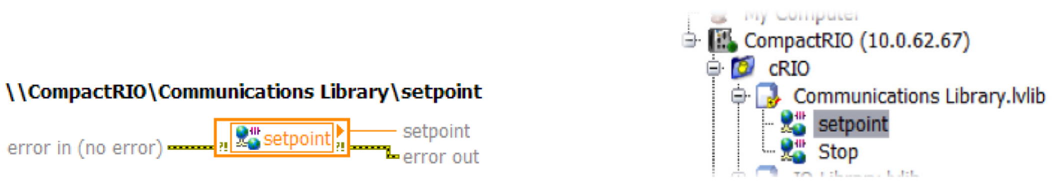
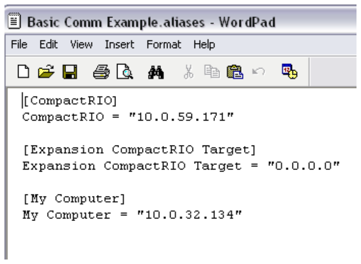


Figure 8.18. Network Variable Node and Its Network Path

To provide scalability this information is not hard-coded into the executable. Instead this information is stored in a file on the target called an alias file. An alias file is a human readable file that lists the logical name of a target (CompactRIO) and the IP address for the target (10.0.62.67). When the executable runs it reads the alias file and replaces the logical name with the IP address. If you later change the IP addresses of deployed systems you only need to edit the alias file to relink the two devices. For real-time and Windows XP Embedded targets, the build specification for each system deployment automatically downloads the alias file. For Windows CE targets, you need to configure the build specification to download the alias file.


Basic Comm Example.aliases - WordPad
File Edit View Insert Format Help

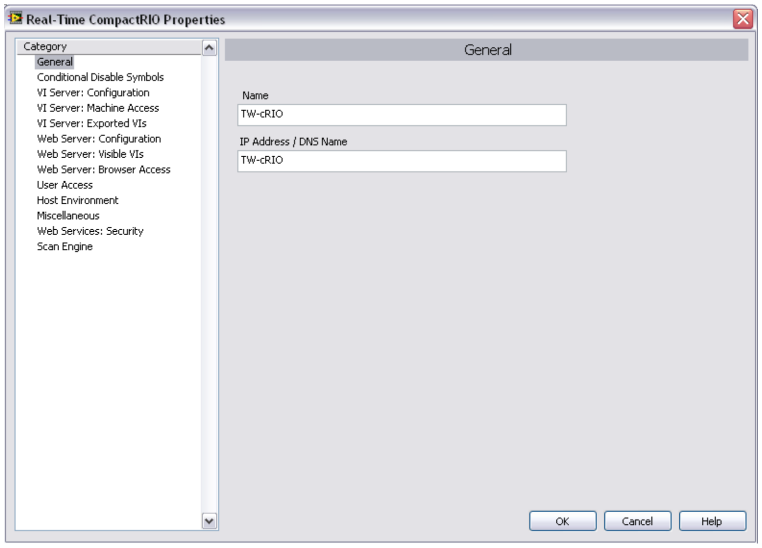
```
[CompactRIO]
CompactRIO = "10.0.59.171"

[Expansion CompactRIO Target]
Expansion CompactRIO Target = "0.0.0.0"

[My Computer]
My Computer = "10.0.32.134"
```

Figure 8.19. The Alias file is a human-readable file that lists the target name and IP address.

If you are deploying systems that have dynamic IP addresses using DHCP, you can use the DNS name instead of the IP address. In the LabVIEW Project, you can type the DNS name instead of the IP address in the properties page of the target.



General

Name
TW-cRIO

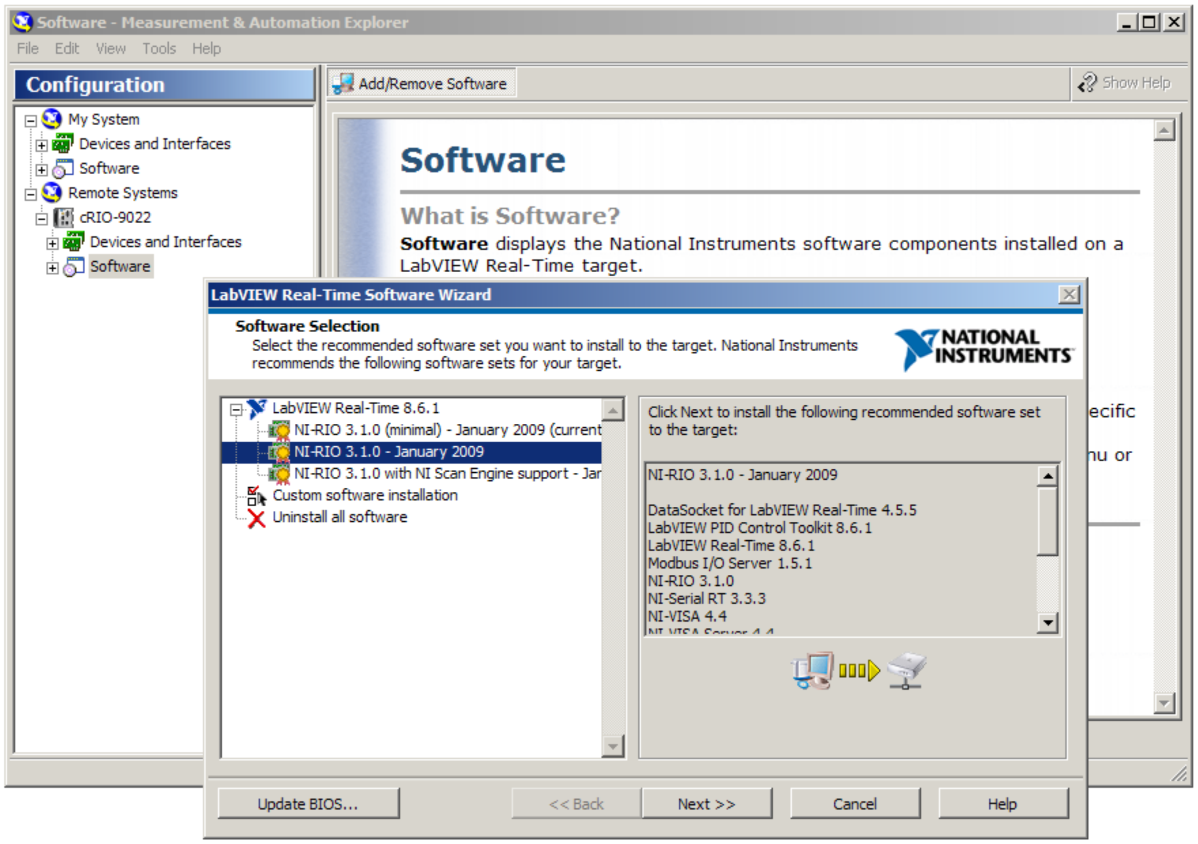
IP Address / DNS Name
TW-cRIO

Figure 8.20. For systems using DHCP, you can enter the DNS name instead of the IP address.

One good approach if you need scalability is to develop using a generic target machine (you can develop for remote machines that don't exist) with a name indicating its purpose in the application. Then as part of the installer, you can run an executable which either prompts the user for the IP addresses for the remote machine and "My Computer" or pulls them from another source such as a database. Then the executable can modify the aliases file to reflect these changes.

Recommended Software Stacks for CompactRIO

National Instruments also provides several sets of commonly used driver sets called Recommended software sets. Recommended software sets can be installed onto CompactRIO controllers from Measurement and Automation Explorer.



Software

What is Software?

Software displays the National Instruments software components installed on a LabVIEW Real-Time target.

Software Selection

Select the recommended software set you want to install to the target. National Instruments recommends the following software sets for your target.

LabVIEW Real-Time 8.6.1
NI-RIO 3.1.0 (minimal) - January 2009 (current
NI-RIO 3.1.0 - January 2009
NI-RIO 3.1.0 with NI Scan Engine support - Jar
Custom software installation
Uninstall all software

Click Next to install the following recommended software set to the target:

NI-RIO 3.1.0 - January 2009

DataSocket for LabVIEW Real-Time 4.5.5
LabVIEW PID Control Toolkit 8.6.1
LabVIEW Real-Time 8.6.1
Modbus I/O Server 1.5.1
NI-RIO 3.1.0
NI-Serial RT 3.3.3
NI-VISA 4.4

Figure 8.21. Recommended software sets Being Installed on a CompactRIO Controller

Recommended software sets guarantee that an application has the same set of underlying drivers for every real-time system that has the same software set. In general, there is a minimal and full software set.

System Replication

LabVIEW Example Code
is provided for this section

After a LabVIEW Real-Time application has been deployed to a CompactRIO controller, it may become desired to deploy that image to other identical Real-Time targets. The LabVIEW Project and the LabVIEW Application Builder can be used to re-deploy an already built application using the procedure described above. This method of replication becomes inefficient when attempting to replicate and deploy more than a few systems.

Because of these in-efficiencies, National Instruments provides a set of system replication VIs for the replication of LabVIEW Real-Time targets. The tools provided can be used to replicate one Real-Time target into multiple copies, circumventing the use of MAX and an FTP client in favor of a simple utility or the ability to customize your own using LabVIEW.

The Real-Time System Replication Tools provide a LabVIEW API with functionality to create an image of a Real-Time target as well as replicating a target with a stored image. It also provides the ability to find target, get information about the target, and also format targets which support remote formatting. The System Replication toolkit can be downloaded by going to Support»Drivers and Updates»All Versions»LabVIEW»Real-Time Module. The Real-Time Target System Replication VIs require the LabVIEW Internet Toolkit to implement FTP functions.

A prebuilt replication executable is installed when the System Replication toolkit is installed. For many users, the prebuilt executable is sufficient for their replication needs. It can be run on a Windows desktop, or run from the command line or within a batch file as part of an installer. When the executable is run from the desktop, it provides a graphical user interface. Additionally, the Replication API is described for applications that require more functionality and flexibility than the prebuilt replication utility.

Use the Real-Time System Deployment Utility

After creating, testing, and deploying a built application, you are left with a working Real-Time target with the necessary set of drivers. At this point, you want preserve this working image of the real-time target for later replication to identical systems. The replication executable stores the image of these drivers and optionally the rest of the hard drive contents. To copy the contents of the remote hard drive to the local file system, only a few steps are necessary. First you must launch the RT System Deployment Utility found here: C:\Program Files\National Instruments\LabVIEW 8.6\user.lib\SystemReplication\Replication Examples\rtcp.exe

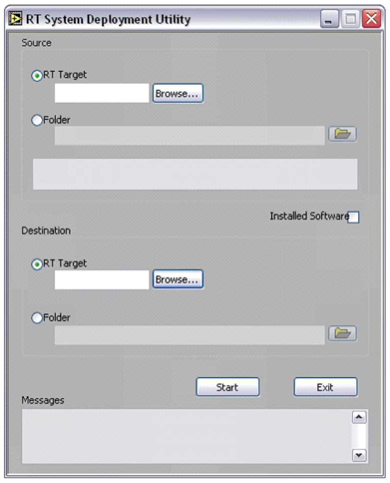
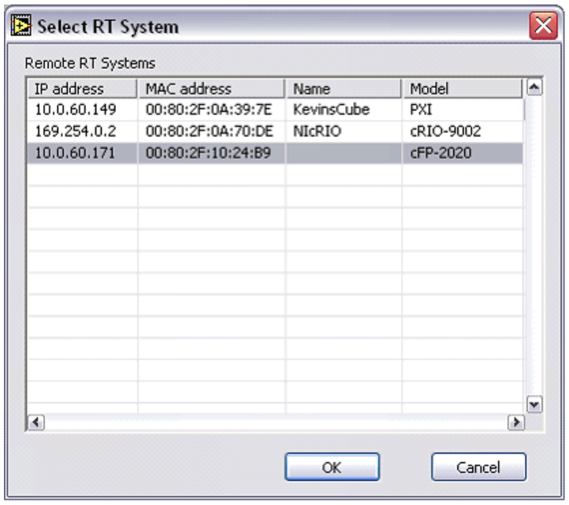


Figure 8.22. The RT System Deployment Utility GUI

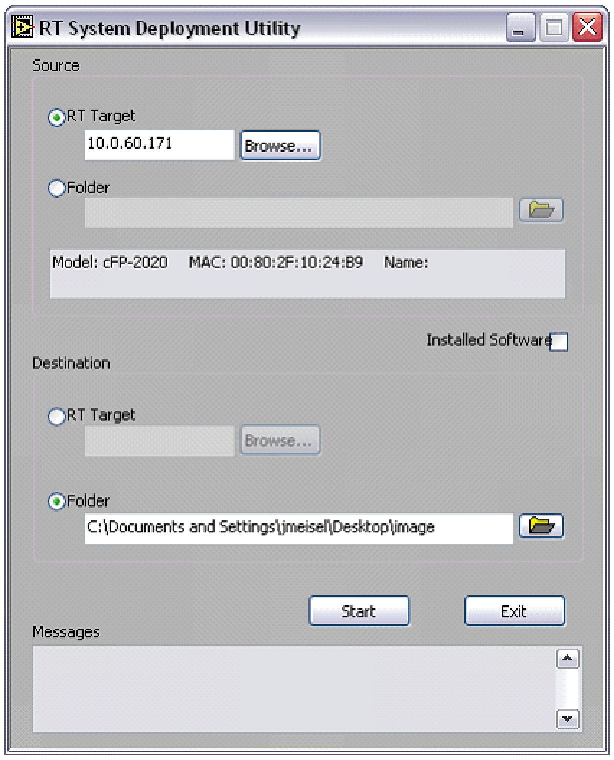
In the utility, set the RT Target to be the source as in the screen capture shown in Figure 8.22. Then, press the adjacent Browse… button to choose a real-time target to serve as the master.



IP address	MAC address	Name	Model
10.0.60.149	00:80:2F:0A:39:7E	KevinsCube	PXI
169.254.0.2	00:80:2F:0A:70:DE	NIcRIO	cRIO-9002
10.0.60.171	00:80:2F:10:24:B9		cFP-2020

Figure 8.23. The RT System Deployment Utility Remote RT Systems Dialog Box

Pressing the browse button creates a list of real-time targets available on the local subnet, as shown in Figure 8.23. IP address, MAC address (a unique identifier assigned to individual network interface cards), name, and model are given to aid identification of the right remote machine. Choose the machine you want to replicate and then press OK.



Source
RT Target: 10.0.60.171
Folder

Model: cFP-2020 MAC: 00:80:2F:10:24:B9 Name:

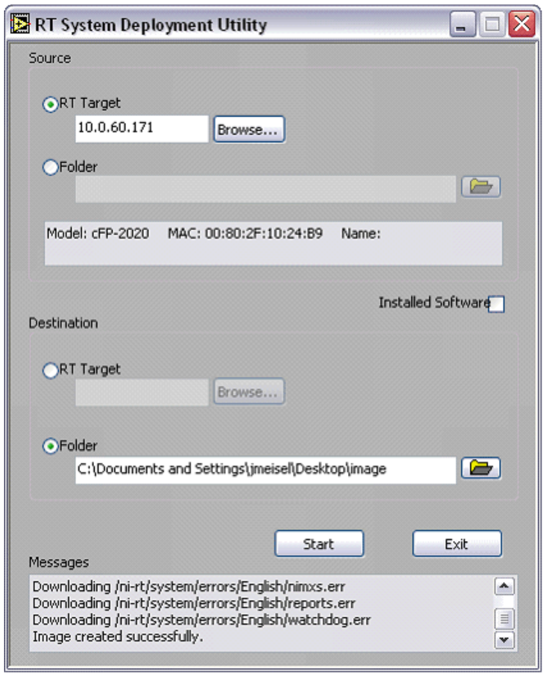
Installed Software

Destination
RT Target
Folder: C:\Documents and Settings\jmeisel\Desktop\image

Figure 8.24. Selecting the Real-Time Target Source and Destination

Next choose destination to be Folder and then select the directory in the local file system where you want to keep the master image.

Press start, and the backup process begins. At the end of the process, the Messages window should display "Image created successfully."



Source

(● RT Target) 10.0.60.171

Folder

Model: cFP-2020 MAC: 00:80:2F:10:24:B9 Name:

Installed Software ☐

Destination

RT Target

(● Folder) C:\Documents and Settings\jmeisel\Desktop\image

Messages

```
Downloading /ni-rt/system/errors/English/nimxs.err
Downloading /ni-rt/system/errors/English/reports.err
Downloading /ni-rt/system/errors/English/watchdog.err
Image created successfully.
```

Figure 8.25. Creating the Replication Image from a Real-Time Target

Deploying a Stored Image

The RT System Deployment Utility assumes the targets of interest are ready to accept file transfers. This means that they must have a valid IP address assigned. This is assigned from within MAX, and, for the purposes of this utility, by someone with a copy of MAX and knowledge of setting IP addresses for remote real-time targets. To deploy a created image, perform the following steps:

1. Select Folder as the source this time. Then browse to or type in the path to the image stored in the previous section.
2. Select RT Target as the destination. Either type in the IP address of the desired target or use the Browse… button to interactively select a target.
3. Now press start to begin the process of copying the previously stored image. Note that images can only be deployed to identical systems.

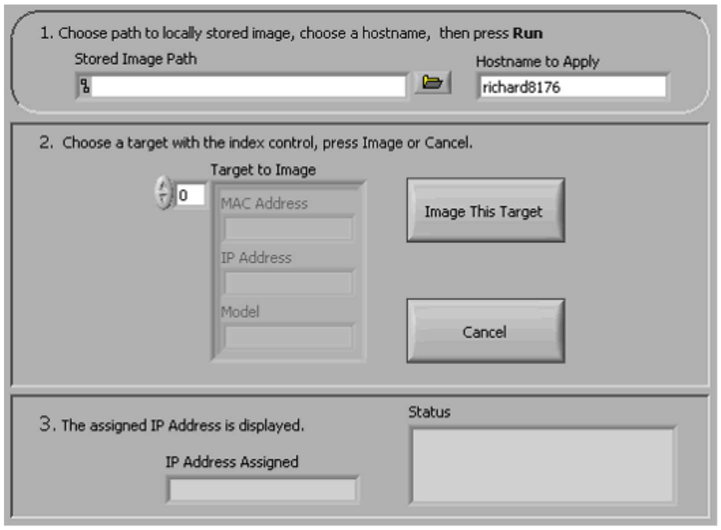
Create a Custom Replication Tool Using the System Replication VIs

With the System Replication VIs, you can create custom VIs and executables to perform replication and backup operations. As an example, a systems integrator can use such an executable to deploy upgrades of software to customers without requiring customers to install drivers or use MAX and an FTP client to execute the update.

The next section discusses a highly simplified example VI to install a stored image to a target which has no software installed and no configured IP address.

Apply Image to Unconfigured Target.vi

This simple example finds a target in a freshly formatted (what is noted in MAX as Unconfigured – without IP address applied) state and applies a selected image to it. It is intended as a starting point for understanding the usage of the VIs, and is not provided as a solution by National Instruments.



1. Choose path to locally stored image, choose a hostname, then press **Run**

Stored Image Path

Hostname to Apply: richard8176

2. Choose a target with the index control, press Image or Cancel.

Target to Image: 0

MAC Address

IP Address

Model

3. The assigned IP Address is displayed.

IP Address Assigned

Status

Figure 8.26. The Apply Image to Unconfigured Target.vi Front Panel

The front panel provides steps to follow in order to use the VI. First, choose the path to the stored image, and choose a hostname to apply to the target. This name should distinguish the target from other targets in some way. Next, press the Run button to begin the rest of the VI.

After a few seconds, the middle section of the front panel should display a MAC and IP address and a model name (such as cRIO-9014) for every un-configured target found. Use the index control to choose the target of interest and select the Image… button to proceed with installing the image.

After several seconds (time to completion varies based on network traffic and hardware), the assigned IP address (assigned by DHCP) is displayed, and the image has been successfully copied, including any startup executables and support files.

Now examine the block diagram to learn about the System Replication VIs.

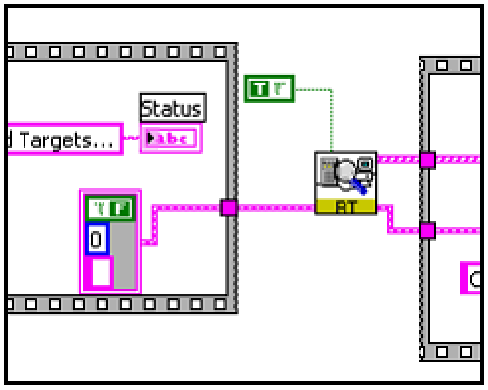


Figure 8.27 The Apply Image to Unconfigured Target.vi Block Diagram

First, the VI called Find All Targets.vi executes. The Boolean True constant wired into this VI specifies that only unconfigured targets are returned. This means only targets that do not have IP addresses assigned are returned. The returned array of clusters contains details about each target found: Name, MAC address, Model Code, IP address, and more. These can be used to filter the list for certain models of target. This is especially useful for automated imaging since an image should only be applied to an identical target. For example, the array of clusters can be parsed to find only cRIO-9014 controllers.

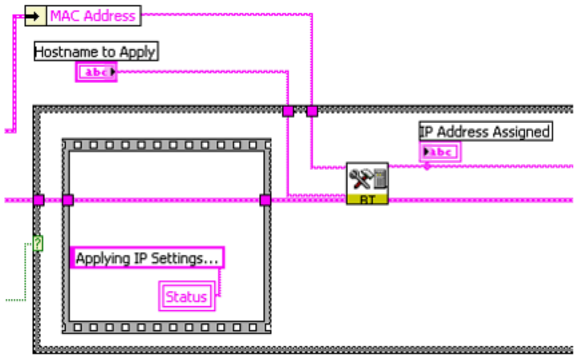


Figure 8.28. The Apply Image to Unconfigured Target.vi Block Diagram

Next, after the user selects the target of interest, that target's MAC address is unbundled by name, and passed on to Set IP Address.vi. This VI can enable or disable DHCP (automatic IP assignment), and set IP, Subnet, Gateway, and DNS addresses. It also applies a hostname to the target. The VI outputs the assigned IP address, which is especially useful in the case of DHCP, which automatically finds an available IP address and can therefore result in unpredictable addresses.

Set Target Image.vi

This IP address is passed along to the VI Set Target Image.vi. This VI performs the final task of copying the stored master image to the duplicate machine. The VI detects Windows installations and avoids overwriting Windows and related files. Therefore it is safe to perform Set Target Image on dual-boot targets, though as noted this may require additional tweaking of the stored image. The VI also preserves IP address settings rather than overwriting them with the master's settings. An ordinary FTP transfer overwrites the IP address settings. This VI can also restrict images to be applied only to the exact same target, as verified via MAC address.

Get Target Info (IP).vi and Get Target Info (MAC).vi

These two VIs search for a single target based on its IP or MAC address and returns the same information as Find All Targets.vi, but only for the single specified target. It is useful for verifying that the IP or MAC address corresponds to the right Model.

Format Target.vi

Format Target formats the disk of remote targets. For targets that may create several files during execution, or may need frequent upgrades or application changes, this may prove useful.

Get Target Image.vi

This VI saves the target image to the specified directory. A Boolean input determines if all files on the target are copied, or only those that are normally installed by MAX. If all files are copied, any executable that has been set as "startup" also runs at startup on the duplicate machine.

NI-RIO System Replication Tools

The NI-RIO System Replication Tools provide additional support to the Real-Time System Replication tools by providing a LabVIEW API with functionality to create an image of a Real-Time target that incorporates NI FPGA targets. It has the ability to erase and download an FPGA bitfile to the Flash, the ability to set how a VI is loaded from Flash Memory. The NI-RIO System Replication Tools require NI-RIO to be installed on the Host PC using the VIs.

Set RIO Device Settings.vi

Set RIO Device Settings is used to configure when the bitfile loads from flash memory.

Download Bitfile.vi

This VI downloads a specific bitfile to one or more FPGA targets or erases the existing bitfile. The input is an IP Address of a Host machine, the FPGA Target Resource (RIO0), the path to the bitfile, and the operation to perform (Download or Erase the bitfile).

IP Protection

Intellectual property (IP) in this context refers to any unique software or application algorithm(s) that you or your company has independently developed. This can be a specific control algorithm or a full scale deployed application. IP normally takes a lot of time to develop and provides companies with a way to differentiate from competition. Therefore, protecting this software IP is very important. LabVIEW development tools and CompactRIO provide you the ability to protect and lock your IP. In general there are two levels of IP protection you can implement:

Lock Algorithms or Code to Prevent IP from Being Copied or Modified

If you have created algorithms for specific functionality, such as performing advanced control functions, implementing custom filtering, and so on, you may want to be able to distribute the algorithm as a subVI but prevent someone from viewing or modifying that actual algorithm. This may be for IP protection or it may be to reduce a support burden by preventing other parties from modifying and breaking your algorithms.

Lock Code to Specific Hardware to Prevent IP from Being Replicated

If you want to ensure that a competitor can't replicate your system by running your code on another CompactRIO system or may want your customers to come back to you for service and support.

Locking Algorithms or Code to Prevent Copying or Modification

Protect Deployed Code

LabVIEW is designed to protect all deployed code and all code running as a start-up application on a CompactRIO controller is by default locked and cannot be opened. Unlike other off-the-shelf controllers or some PLCs where the raw source code is stored on the controller and only protected by a password, CompactRIO systems do not require the raw source code to be stored on the controller.

Code running on the real-time processor is compiled into an executable and cannot be "decompiled" back to LabVIEW code. Likewise, code running on the FPGA has been compiled into a bit file and cannot be "decompiled" back to LabVIEW code. To aid in future debugging and maintenance it is possible to store the LabVIEW project on the controller or to call raw VIs from running code, but by default any code deployed to a real-time controller is protected to prevent copying or modifying the algorithms.

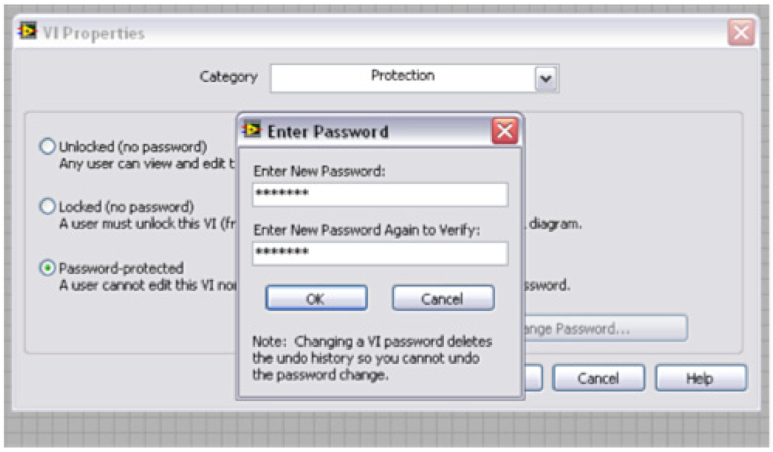
Protect Individual VIs

Sometimes you want to provide the raw LabVIEW code to enable end customers to perform customization or maintenance but still want to protect specific algorithms. LabVIEW provides a few mechanisms to provide usable subVIs while still protecting the IP in those VIs.

Method 1. Password Protecting Your LabVIEW Code

Password protecting a VI adds functionality that requires users to enter a password if they want to edit or view the block diagram of a particular VI. Because of this, you can give a VI to someone else and protect your source code. Password protecting a LabVIEW subVI prohibits others from editing the VI or viewing its block diagram without the password. However, if the password is lost, there is no way you can unlock a VI. Therefore, you should strongly consider keeping a backup of your files stored without passwords in another secure location.

To password protect a VI, go to **File»VI Properties**. Choose Protection for the category. This gives you three options: unlocked (the default state of a VI), locked (no password), and password-protected. When you click on password-protected, a window appears for you to enter your password. The password takes effect the next time you launch LabVIEW.



Unlocked (no password)
Any user can view and edit t

Locked (no password)
A user must unlock this VI (fr . diagram.

Password-protected
A user cannot edit this VI no ssword.

Enter New Password:
****

Enter New Password Again to Verify:
****

Note: Changing a VI password deletes the undo history so you cannot undo the password change.

Figure 8.29. Password Protecting LabVIEW Code

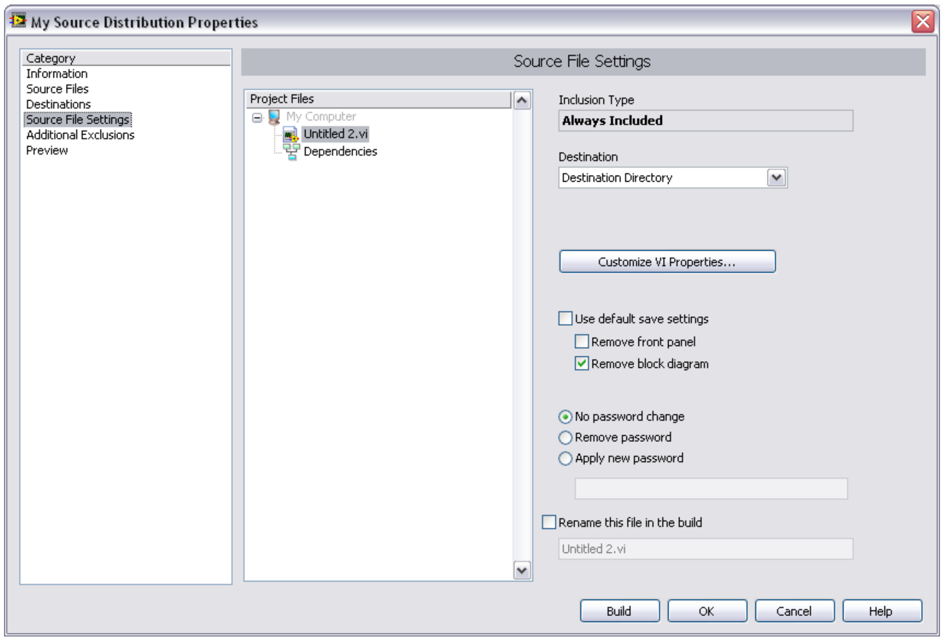
The LabVIEW password mechanism is quite difficult to defeat, but no password algorithm is 100% secure from attack. If you need total assurance that someone cannot gain access to your source code, you should consider removing the block diagrams.

Method 2. Removing the Block Diagram

To guarantee that a VI cannot be modified or opened you can remove the block diagram completely. Much like an executable, the code you distributed no longer contains the original editable code. Don't forget to make a backup of your files if you use this technique, as the block diagrams cannot be recreated. Removing the block diagram is an option you can select when creating a source distribution. A source distribution is a collection of files that you can package and send to other developers to use in LabVIEW. You can configure settings for specified VIs to add passwords, remove block diagrams, or apply other settings.

Complete the following steps to build a source distribution.

1. In the LabVIEW Project right-click Build Specifications and select New»Source Distribution from the shortcut menu to display the Source Distribution Properties dialog box. Add your VI(s) to the distribution.
2. On the Source File Settings page of the Source Distribution Properties dialog box, remove the checkmark from the Use default save settings checkbox and place a checkmark in the Remove block diagram checkbox to ensure that LabVIEW removes the block diagram.
3. Build the source distribution to create a copy of the VI without its block diagram.


My Source Distribution Properties

Category: Information, Source Files, Destinations, Source File Settings, Additional Exclusions, Preview

Source File Settings

Project Files: My Computer, Untitled 2.vi, Dependencies

Inclusion Type: Always Included

Destination: Destination Directory

Customize VI Properties...

☐ Use default save settings
☐ Remove front panel
☑ Remove block diagram

◉ No password change
○ Remove password
○ Apply new password

☐ Rename this file in the build: Untitled 2.vi

Build | OK | Cancel | Help

Figure 8.30. Removing the Block Diagram from LabVIEW VIs

CAUTION: If you save VIs without block diagrams, do not overwrite the original versions of the VIs. Save the VIs in different directories or use different names.

Lock Code to Hardware to Prevent IP Replication

Some OEMs and machine builders also want to protect their IP by locking the deployed code to a specific system. To make system replication easy, by default the deployed code on a CompactRIO controller is not locked to hardware and can be easily moved and executed on another controller. For designers who want to prevent customers or competitors from replicating their systems, one effective way to protect application code with CompactRIO is by locking your code to specific pieces of hardware in your system. This ensures that customers cannot take the code off of a system they have purchased from you and run the application on a different set of CompactRIO hardware. You can lock the application code to a variety of hardware components in a CompactRIO system including:

- The MAC address of a real-time controller
- The serial number of a real-time controller
- The serial number of the CompactRIO backplane
- The serial number of individual modules
- Third-party serial dongle

The following steps can be used as guidelines to programmatically lock any application to any of the above mentioned hardware parameters and thus prevent users from replicating application code:

1. Obtain the hardware information for the device. Refer to the procedures below for more information on programmatically obtaining this information.
2. Compare the values obtained to a predetermined set of values that the application code is designed for using the Equal? function from the Comparison palette.
3. Wire the results of the comparison to the selector input of a Case Structure.
4. Place the application code in the true case and leave the false case blank.

Performing these steps ensures that the application is not replicated or usable on any other piece of CompactRIO hardware.

License Key

When deploying many systems hard coding the hardware identification may not be ideal as it requires a manual change to the source code and recompile for each system deployed. This problem can be addressed by using a license key file which is stored separate from the application code on the CompactRIO controller. The license key file can be easily updated for each system without needing to change the application. In addition to reading the MAC or serial number the VI can open the license file and verify that the license is valid. For security, the license file should be specific for each deployed system and your code should perform a mathematical operation between the license key and hardware specific features such as the MAC address. Because you can find the MAC and serial numbers programmatically on Windows and real-time OSs, you can develop a LabVIEW application for system deployment that automatically queries the CompactRIO system and generates and deploys the appropriate license key file.

Acquire the MAC Address of Your CompactRIO System

Example LabVIEW Example Code
is provided for this section

You can acquire the MAC address of real-time controllers manually from MAX on the network settings tab of the controller or from the Windows command line. You can also find the MAC address programmatically.

Acquiring the MAC Address from Windows

To programmatically find the MAC address of a real-time system from Windows, perform the following steps:

1. Run the RT Ping Controllers VI found in the **Real-Time»Real-Time Utilities** palette. This VI returns the network information of all real-time controllers on the subnet.

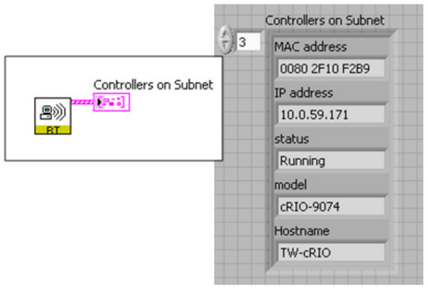


Figure 8.31. Obtain MAC address of networked real-time targets from Windows.

2. Search in the data that the RT Ping Controllers VI returned to find the appropriate real-time controller by using either the IP Address or Hostname.
3. This cluster returns the controllers' MAC addresses. The MAC addresses returned by the RT Ping Controllers VI are an array of strings in hexadecimal format.

Acquiring the MAC Address from the Real-Time Target

To find the MAC address programmatically from a real-time controller, perform the following steps:

1. Run the RT Ping Controllers VI found in the **Real-Time»Real-Time Utilities** palette while targeted to a real-time controller. Wire into the system location input a constant cluster with False and local host entries. This returns the information for the real-time controller where the code is executing.

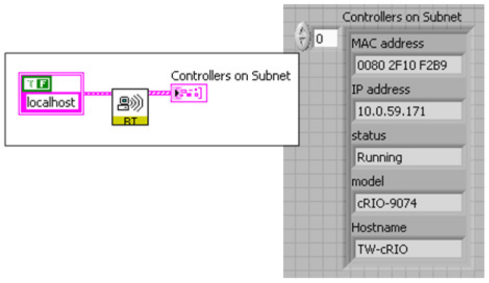


Figure 8.32. Obtain the MAC address on a real-time target.

2. Search in the data that the RT Ping Controllers VI returned for the controllers' MAC addresses. The MAC addresses returned by the RT Ping Controllers VI are an array of strings returned in a hexadecimal format.

CompactRIO Information Retrieval Tools

You can also get other information about CompactRIO system such as serial numbers using the CompactRIO Information Retrieval Tools. These free VIs can be run under Window or on the real-time controller to retrieve information about a local or remote CompactRIO controller, backplane, and modules including the type and serial number of each of these system components.

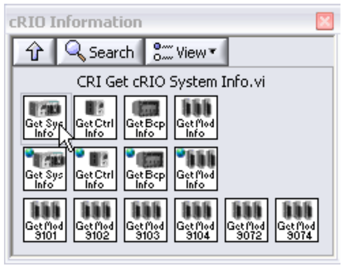


Figure 8.33. The CompactRIO Information Retrieval Tools return information such as serial numbers.

If you are using the LabVIEW Real-Time System Replication Tools, these also programmatically retrieve the real-time target's serial number.

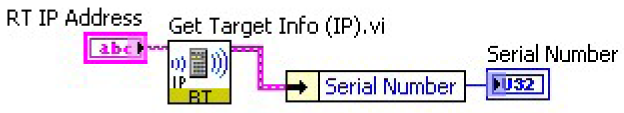


Figure 8.34. Obtain Serial Number Using Real-Time System Replication Tools

Porting to Other Platforms

This document has focused on architectures for building embedded control systems using CompactRIO systems. The same basic techniques and structures also work on other National Instruments control platforms including PXI and NI Single-Board RIO. Because of this, you can reuse your algorithms and your architecture for other projects that require different hardware or easily move your application between platforms. However, CompactRIO has several features to ease learning and speed development that are not available on all targets. This section covers the topics you need to consider when moving between platforms and shows you how to port an application to NI Single-Board RIO.

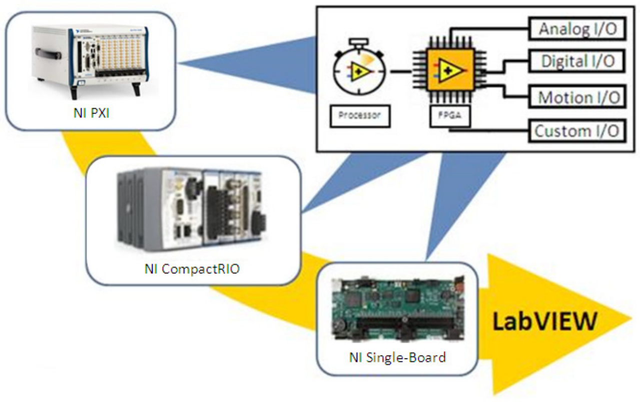


Figure 8.35. Using LabVIEW, you can use the same architecture for applications ranging from CompactRIO to high-performance PXI to board-level NI Single-Board RIO.

LabVIEW Code Portability

LabVIEW is a cross-platform programming language capable of compiling for multiple processor architectures and operating systems. In most cases, algorithms written in LabVIEW are portable between all LabVIEW targets. In fact you can even take LabVIEW code and compile it for any arbitrary 32 bit processor allowing you to port your LabVIEW code to custom hardware. When porting code between platforms, the most commonly needed changes are related to the physical I/O changes of the hardware.

When porting code between CompactRIO targets, all I/O is directly compatible because C Series modules are supported on all CompactRIO targets. If you need to port an application to NI Single-Board RIO, all C Series modules are supported, but depending on your application, you may need to adjust the software I/O interface.

NI Single-Board RIO

The NI Single-Board RIO is a board-only version of CompactRIO designed for applications where a bare board form factor is required. While it is physically a different design, it uses the processor and FPGA and accepts up to three C Series modules. NI Single-Board RIO differs from CompactRIO because it includes I/O built directly onto the board. NI Single-Board RIO features 110 3.3 V bidirectional digital I/O lines, and up to 32 analog inputs, 4 analog outputs, and 32 24 V digital input and output lines, depending on the model used.

LabVIEW FPGA Programming

NI Single-Board RIO does not currently support scan mode. Instead of using the scan mode to read I/O you need to write a LabVIEW program to read the I/O from the FPGA and insert it into a memory table. This section examines an effective FPGA architecture for single-point I/O communication similar to scan mode later in this section and shows how to covert an application using scan mode.

Built-In I/O and I/O Modules
Depending on the I/O requirements of your application, you may be able to create your entire application to use only the NI Single-Board RIO onboard I/O, or you may need to add modules. When possible, design your application to use the I/O modules available onboard NI Single-Board RIO. The I/O available on NI Single-Board RIO and the module equivalents are listed below:

- 110 general purpose, 3.3 V (5 V tolerant, TTL compatible) digital I/O (no module equivalent)
- 32 single-ended/16 differential channels, 16-bit analog input, 250 kS/s aggregate (NI 9205)
- 4-channel, 16-bit analog output, 100 kS/s simultaneous (NI 9263)
- 32-channel, 24 V sinking digital input (NI 9425)
- 32 channel, 24 V sourcing digital output (NI 9476)

NI Single-Board RIO accepts up to three additional C Series modules. Applications that need more than three additional I/O modules are not good candidates for NI Single-Board RIO, and you should consider CompactRIO integrated systems as a deployment target.

FPGA Size
The largest FPGA available on NI Single-Board RIO is the Xilinx 2M system gate Spartan-3 FPGA. CompactRIO targets offer versions using both the Spartan-3 FPGAs and larger, faster Virtex-5 FPGAs. To test if code fits on hardware you do not own, you can add a target to your LabVIEW project and, as you develop your FPGA application, you can periodically benchmark the application by compiling the FPGA code for a simulated RIO target. This gives you a good understanding of how much of your FPGA application will fit on the Spartan-3 FPGA.

Port CompactRIO Applications to NI Single-Board RIO or R Series Devices
There are four main steps to port a CompactRIO application to NI Single-Board RIO or PXI/PCI R Series FPGA I/O devices.

1. Build an NI Single-Board RIO or R Series project with equivalent I/O channels
2. If using CompactRIO Scan Mode, build a LabVIEW FPGA-based scan API
   a. Build LabVIEW FPGA I/O scan (analog in, analog out, digital I/O, specialty digital I/O)
   b. Convert I/O variable aliases to single-process shared variables with real-time FIFO enabled
   c. Build a real-time I/O scan with scaling and shared variable-based current value table
3. Compile LabVIEW FPGA VI for new target
4. Test and validate updated real-time and FPGA code

The first step in porting an application from CompactRIO to NI Single-Board RIO or R Series FPGA device is finding the equivalent I/O types on your target platform. For I/O that cannot be ported to the onboard I/O built into NI Single-Board RIO or R Series targets, you can add C Series modules. All C Series modules for CompactRIO are compatible with both NI Single-Board RIO and R Series. You must use the NI 9151 R Series expansion chassis to add C Series I/O to an R Series data acquisition (DAQ) device.

Step 2 is necessary only if the application being ported was originally written using scan mode. If you need to replace the scan mode portion of an application with an I/O method supported on NI Single-Board RIO and R Series, an example is included below to guide you through the process.

If the application you are migrating to NI Single-Board RIO did not use scan mode, the porting process is nearly complete. Skip step 2 and add your real-time and FPGA source code to your new NI Single-Board RIO project, recompile the FPGA VI, and you are now ready to run and verify application functionality. Because CompactRIO and NI Single-Board RIO are both based upon the RIO architecture and reusable modular C Series I/O modules, porting applications between these two targets is very simple.

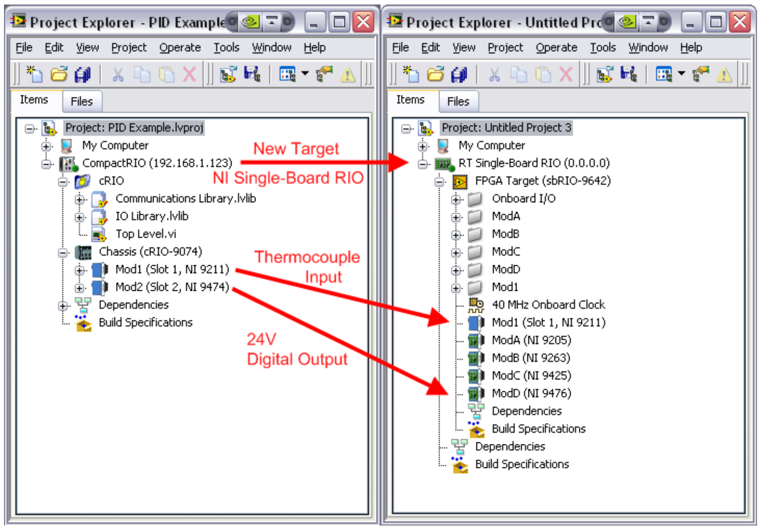


Figure 8.36. The first step in porting an application from CompactRIO to an alternate target is finding replacement I/O on the future target.

Example of Porting a CompactRIO Scan Mode-Based Application to NI Single-Board RIO

LabVIEW Example Code
is provided for this section

If you used scan mode in your original application, you need to create a simplified FPGA version of the scan mode because NI Single-Board RIO and R Series DAQ devices do not support scan mode. Building a scan engine in FPGA is very similar to the method for inserting single-point data from FPGA into the real-time scan discussed in the "Programming with LabVIEW FPGA" section. There are three steps to replace scan mode with a similar FPGA-based scan engine and current value table:

1. Build a LabVIEW FPGA I/O scan engine
2. Replace IOVs with single-process shared variables
3. Insert FPGA data into the shared variable based current value table in LabVIEW Real-Time

First, create a LabVIEW FPGA VI that samples and updates all analog input and output channels at the rate specified in your scan engine configuration. You can use IP blocks to recreate specialty digital functionality such as counters, PWM, and quadrature encoder.

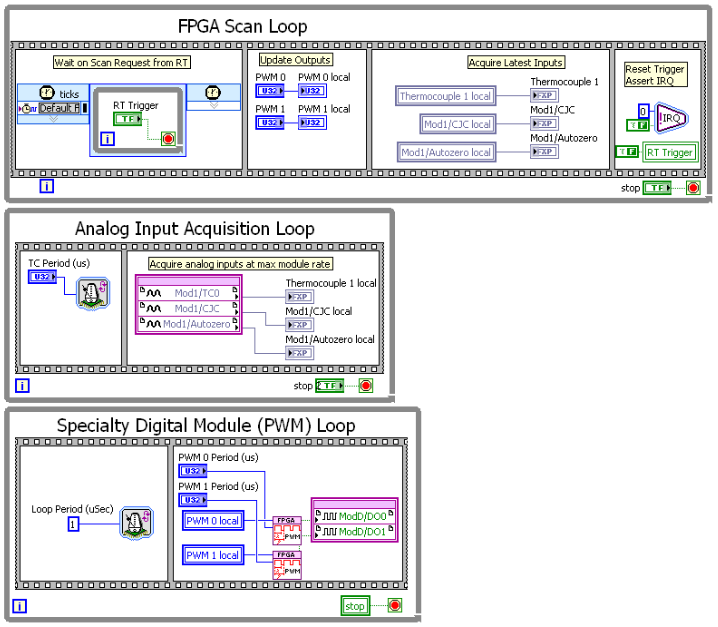
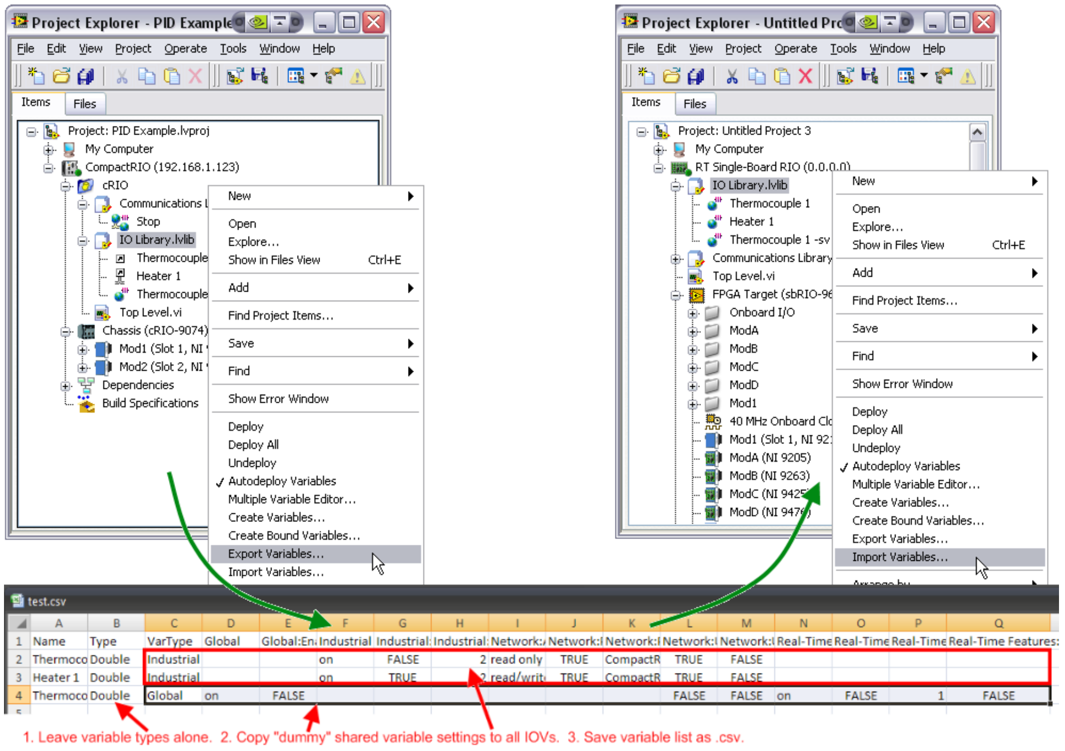


Figure 8.37. Develop a simple FPGA application to act as an FPGA scan engine.

After you have implemented a simple scan engine in the FPGA, it is time to port the real-time portion of the application to communicate with the custom FPGA scan rather than the current value table built into scan mode. To accomplish this, you need to first convert all I/O variable aliases to single-process shared variables with the real-time FIFO enabled. The main difference between the two variables is while I/O variables are automatically updated by a driver to reflect the state of the input or output channel, a single-process shared variables are not updated by a driver. You can change the type by going to the properties page for each IOV Alias and changing to single-process.

Tip: If you have numerous variables to convert, by exporting to a text editor and changing the properties you can easily convert a library of IOV aliases to shared variables. To make sure you get the properties correct it is easiest if you first create one "dummy" single-process shared variable with single element real-time FIFO enabled in the library then export the library to a spreadsheet editor. While in the spreadsheet editor, delete the columns exclusive to IOVs and copy the data exclusive to the share variables to the IOV rows. Then import the modified library into your new project. The IOV Aliases are imported as single-process shared variables. Because LabVIEW references shared variables and IOV Aliases by the name of the library and the name of the variable, all instances of IOV Aliases in your VI are automatically updated. Finally, delete the dummy shared variable that was created before the migration process.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1	Name	Type	VarType	Global	Global:En	Industrial	Industrial:	Industrial:	Network:/	Network:I	Network:I	Network:I	Network:I	Real-Time	Real-Time	Real-Time	Real-Time Features:
2	Thermoco	Double	Industrial			on	FALSE	2	read only	TRUE	CompactR	TRUE	FALSE				
3	Heater 1	Double	Industrial			on	TRUE	2	read/writ	TRUE	CompactR	TRUE	FALSE				
4	Thermoco	Double	Global	on	FALSE							FALSE	FALSE	on	FALSE	1	FALSE

1. Leave variable types alone. 2. Copy "dummy" shared variable settings to all IOVs. 3. Save variable list as .csv.

Figure 8.38. You can easily convert an IOV Alias Library to shared variables by exporting the variables to a spreadsheet, modifying the parameters, and importing into your new target.

The final step for implementing an FPGA based scan engine and shared variable current value table is building the real-time task to read data from the FPGA and constantly update the current value table. The FPGA I/O you are adding to the current value table is deterministic, so you again use the method described in the "Programming with LabVIEW FPGA" section, except for now you create the real-time portion of that code.

To read data from the FPGA based scan engine, create a timed loop task set to the desired scan rate in your top-level RT VI. This timed loop is the deterministic I/O loop, so it should be set to the highest priority. To match the control loop speed of your previous scan mode application, set the period of this loop to match the period previously set for scan mode. Any other task loops in your application that were previously synchronized to the scan also need to change their timing source to the 1 kHz clock and set to the same rate as the I/O loop.

The I/O scan loop pushes new data to the FPGA and then pulls updated input values. The specific write and read VIs are also responsible for scaling and calibration of analog and specialty digital I/O.

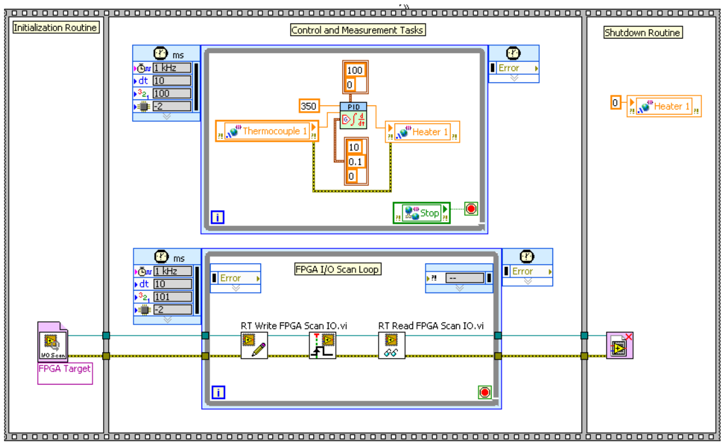


Figure 8.39. The FPGA I/O Scan Loop mimics the CompactRIO Scan Mode feature by deterministically communicating the most recent input and output values to and from the FPGA I/O and inserting the data into a memory table.

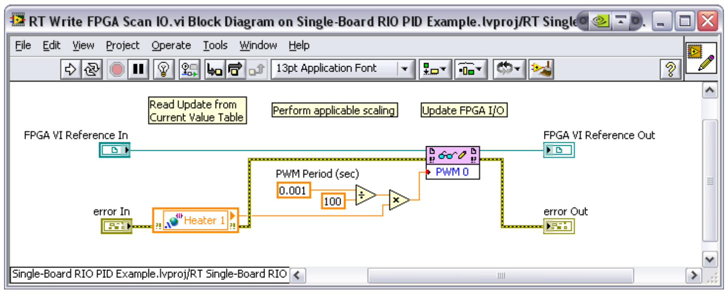

RT Write FPGA Scan IO.vi Block Diagram on Single-Board RIO PID Example.lvproj/RT Single
File Edit View Project Operate Tools Window Help
13pt Application Font
Read Update from Current Value Table
Perform applicable scaling
Update FPGA I/O
FPGA VI Reference In
FPGA VI Reference Out
PWM Period (sec)
0.001
100
PWM 0
error In
Heater 1
error Out
Single-Board RIO PID Example.lvproj/RT Single-Board RIO

Figure 2. The RT Write FPGA Scan IO VI pulls data from the memory table using a real-time FIFO single-process shared variable, scales values with appropriate conversion for the FPGA Scan VI, and pushes values to the FPGA VI.

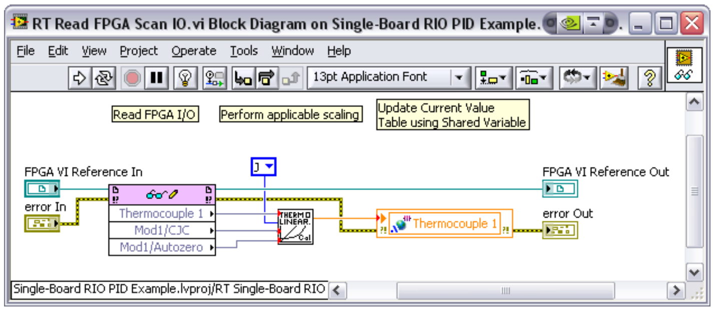

RT Read FPGA Scan IO.vi Block Diagram on Single-Board RIO PID Example.
File Edit View Project Operate Tools Window Help
13pt Application Font
Read FPGA I/O
Perform applicable scaling
Update Current Value Table using Shared Variable
FPGA VI Reference In
FPGA VI Reference Out
J
error In
Thermocouple 1
Mod1/CJC
Mod1/Autozero
THERMO LINEAR
Cal
Thermocouple 1
error Out
Single-Board RIO PID Example.lvproj/RT Single-Board RIO

Figure 8.41. The RT Read FPGA Scan IO VI pulls all updates from the FPGA scan, performs applicable conversions and scaling, and publishes data to the memory table using a real-time FIFO single-process shared variable.

After building the host interface portion of a custom FPGA I/O scan to replace scan mode, you are ready to test and validate your ported application on the new target. Ensure the FPGA VI is compiled and the real-time and FPGA targets in the project are configured correctly with a valid IP address and RIO resource name. After the FPGA VI is compiled, connect to the real-time target and run the application.

Because the RIO architecture is common across NI Single-Board RIO, CompactRIO, and R Series FPGA I/O devices, LabVIEW code written on each of these targets is easily portable to the others. As demonstrated in this section, with proper planning, you can migrate applications between all targets with no code changes at all. When you use specialized features of one platform, such as the CompactRIO Scan Mode, the porting process is more involved, but, in that case, only the I/O portions of the code require change for migration. In both situations, all the LabVIEW processing and control algorithms are completely portable and reusable across RIO hardware platforms.

APPENDIX A
Getting Started with CompactRIO

Getting Started with CompactRIO Tutorial

This tutorial shows you how to get your CompactRIO system out of the box and create your first project. It explains the hardware configuration, necessary software component installation, and development of a simple project to deploy to the CompactRIO system.

CompactRIO and Its Components

A CompactRIO system consists of the following five main components:

1. Chassis
2. Controller (may be integrated with the controller in devices such as cRIO-907x)
3. Modules
4. System accessories
5. Software

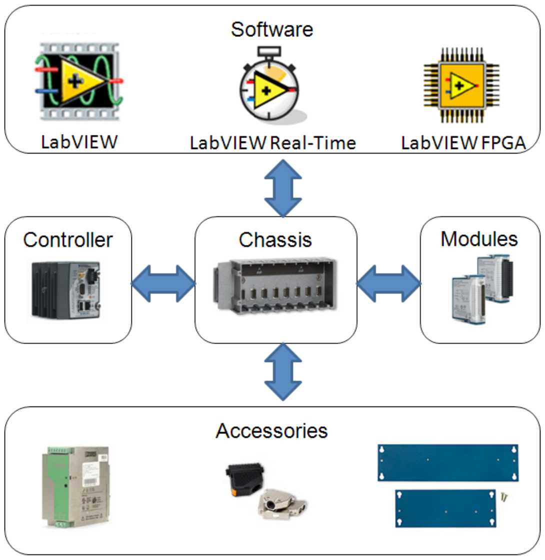


Figure 9.1. Components of a Complete CompactRIO System

Assembling and Configuring the Hardware

After removing the CompactRIO system components from the box, follow these steps to prepare your system for software installation and programming.

Connect the Controller, Chassis, and Modules

If you have a modular controller, attach it to the reconfigurable FPGA chassis. The cRIO-907x series consists of an integrated chassis and controller so no assembly is necessary. To connect the controller to the chassis, follow these steps:

- Make sure there is no power connected to the controller and align it with the chassis.

- Slide the controller over the controller slot in the chassis, as shown in Figure 9.2, and press firmly to make sure they are connected. Use a No. 2 Philips screwdriver to tighten the two captive screws in front of the controller.

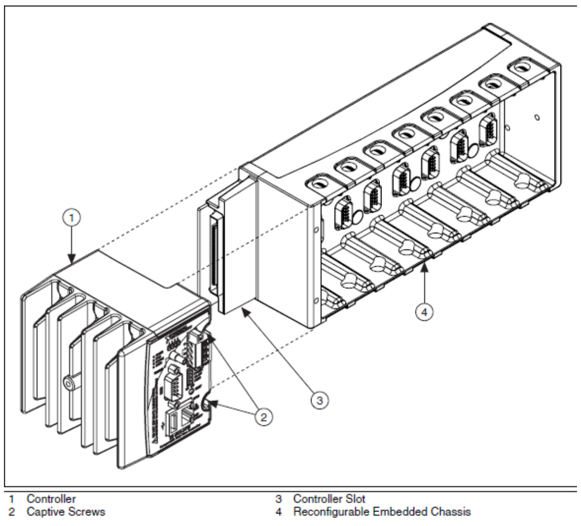


Figure 9.2. Connecting the Controller to the Chassis

Connect Power to the Controller

To connect power, connect an appropriate DC power supply, such as a 24 V supply, to the system. Connect the positive lead of the power supply to the V1 terminal and the negative lead to either of the C terminals. Some controller feature redundant power supply inputs where you can connect a second supply to the V2 terminal.

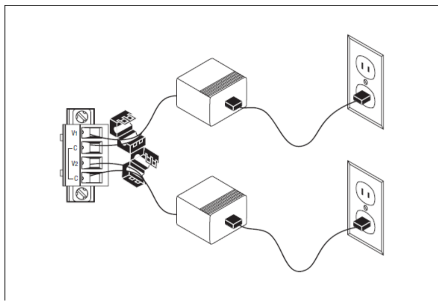


Figure 9.3. Connecting CompactRIO to the Power Supply

Configure Dip Switches

The CompactRIO controller has five dip switches – Safe Mode, Console Out, IP Reset, No App, and User1. All of these switches are set to OFF when National Instruments ships the controller. The switches should remain in the OFF position while configuring the controller.

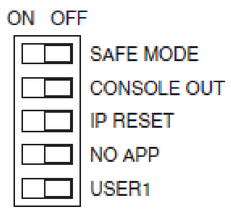


Figure 9.4. Keep the switches in the OFF position while configuring your CompactRIO controller.

The individual dip switches and their uses are discussed in the CompactRIO operating instructions and specifications manuals.

Connect the CompactRIO System to Your PC

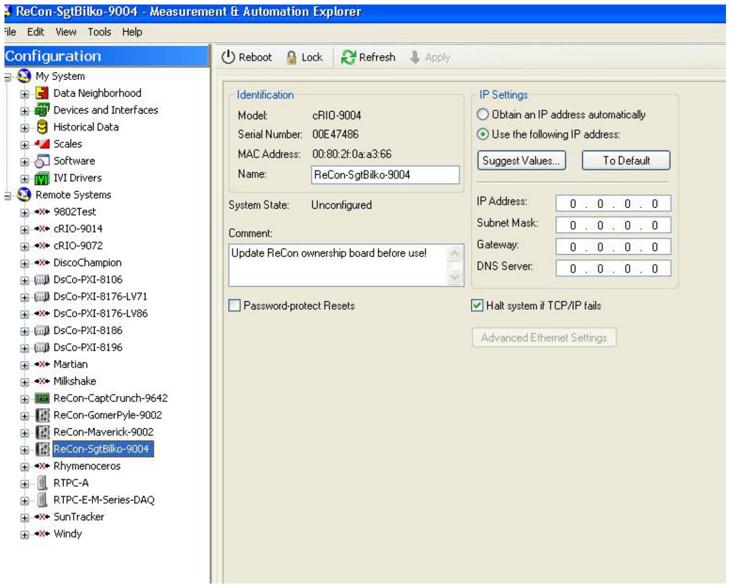
You have two ways to connect the CompactRIO controller to your PC:

1. Place it in the same subnet as your PC by connecting it to a router or hub
2. Connect the CompactRIO system directly to your PC using a crossover cable

Configuring Your CompactRIO System in MAX

Once you have connected all the hardware, you can configure your CompactRIO system with Measurement & Automation Explorer (**Start»Programs»National Instruments»Measurement & Automation Explorer**) software so it can be targeted and programmed with LabVIEW software. Before configuring CompactRIO in MAX, you need to install the LabVIEW Real-Time Module and NI-VISA and NI-RIO software on your host computer. Make sure that you install NI-RIO last.

When you power on your CompactRIO system, you can expand Remote Systems in MAX so that the CompactRIO system initially appears with the cRIO-[controller number] default name. You can change this name in the Identification settings. Once you select your system in the navigation tree, the details pane shows the IP settings for your CompactRIO system. IP settings default to 0.0.0.0.



ReCon-SgtBilko-9004 - Measurement & Automation Explorer
File Edit View Tools Help
Configuration
Reboot
Lock
Refresh
Apply
Suggest Values...
To Default
Advanced Ethernet Settings

- My System
  - Data Neighborhood
  - Devices and Interfaces
  - Historical Data
  - Scales
  - Software
  - IVI Drivers
- Remote Systems
  - 9802Test
  - cRIO-9014
  - cRIO-9072
  - DiscoChampion
  - DsCo-PXI-8106
  - DsCo-PXI-8176-LV71
  - DsCo-PXI-8176-LV86
  - DsCo-PXI-8186
  - DsCo-PXI-8196
  - Martian
  - Milkshake
  - ReCon-CaptCrunch-9642
  - ReCon-GomerPyle-9002
  - ReCon-Maverick-9002
  - ReCon-SgtBilko-9004
  - Rhymenoceros
  - RTPC-A
  - RTPC-E-M-Series-DAQ
  - SunTracker
  - Windy

Identification

Model:	cRIO-9004
Serial Number:	00E47486
MAC Address:	00:80:2f:0a:a3:66
Name:	ReCon-SgtBilko-9004

System State: Unconfigured

Comment: Update ReCon ownership board before use!

☐ Password-protect Resets

IP Settings

○ Obtain an IP address automatically
◉ Use the following IP address:

IP Address:	0 . 0 . 0 . 0
Subnet Mask:	0 . 0 . 0 . 0
Gateway:	0 . 0 . 0 . 0
DNS Server:	0 . 0 . 0 . 0

☑ Halt system if TCP/IP fails

Figure 9.5. IP settings default to 0.0.0.0.

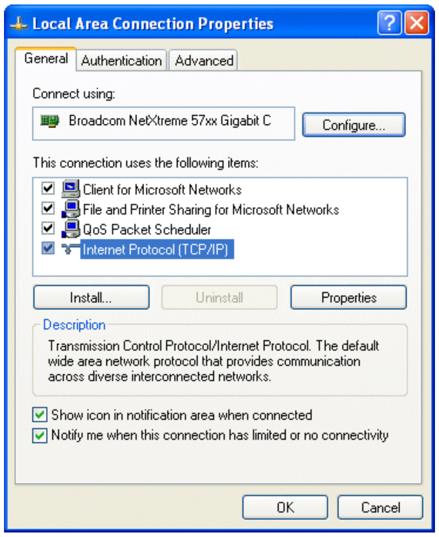
If you are connecting to your CompactRIO system using a network router that supports DHCP, you can set your system to obtain an IP address automatically. When using a crossover cable, however, you must set up the host computer with a static IP. The Internet Assigned Numbers Authority (IANA) has reserved the following three blocks of IP address space for private internets:

- 10.0.0.0 - 10.255.255.255
- 172.16.0.0 - 172.31.255.255
- 192.168.0.0 - 192.168.255.255
- 169.254.0.0 - 169.254.255.255 (link-local addresses for use with crossover cables)

You must assign a unique IP number to each machine on the private network. For example, you can assign 192.168.0.1 to the host machine and 192.168.0.3 to the remote system. You must initially place the host and CompactRIO systems on the same subnet until you configure them with the subnet mask. Find the subnet mask for your host machine by typing "**ipconfig**" in a command prompt: **Start»Run»cmd**.

To configure TCP/IP settings for the host system, complete the following steps:

1. Open the network connections (accessible through the control panel).
2. Right-click the appropriate Local Area Connection and select Properties. On the General tab, under **This connection uses the following items**, click **Internet Protocol (TCP/IP)** and then click **Properties**.
3. To specify an IP address, click Use the following IP address and enter the IP address and subnet mask.



Connect using:
Broadcom NetXtreme 57xx Gigabit C

This connection uses the following items:
☑ Client for Microsoft Networks
☑ File and Printer Sharing for Microsoft Networks
☑ QoS Packet Scheduler
☑ Internet Protocol (TCP/IP)

Description
Transmission Control Protocol/Internet Protocol. The default wide area network protocol that provides communication across diverse interconnected networks.

☑ Show icon in notification area when connected
☑ Notify me when this connection has limited or no connectivity

Figure 9.6. Configuring TCP/IP Settings for the Host System

Configuration Steps in MAX

Follow these steps in MAX to complete the configuration of your CompactRIO system:

1. In the **IP Settings** window set the IP address:
   - If you are connected using a router that supports DHCP, select **Obtain an IP Address automatically**.
   - If you are using a crossover cable, select **Use the following IP Address** and assign an IP address to the CompactRIO system with the same subnet and IP settings as those for your PC.
2. Enter a new name in the **Name** field of the Identification window to change the CompactRIO name, if desired. Click Apply to apply the changes. You may receive a warning stating that the CompactRIO system and the gateway are on different subnets if you did not enter an IP address in the Gateway field. Click **Yes** to safely ignore this warning. If you entered an address in the Gateway field, make sure that it corresponds to a valid Default Gateway value and is not 0.0.0.0. The latter results in a loss of communication with MAX. To find the Default Gateway for your network, type **ipconfig/all** in the command prompt.
3. Install software on the CompactRIO system. Expanding the CompactRIO under Remote Systems brings up Devices and Interfaces as well as Software. Right-click on Software and select Add/Remove Software.

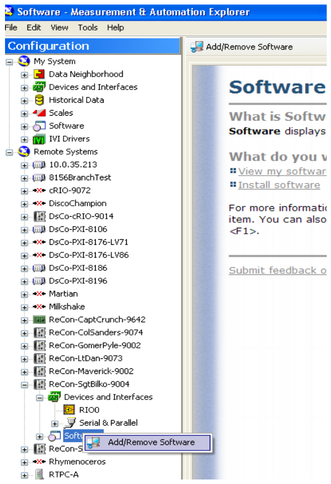


Figure 9.7. Installing Software on the CompactRIO System

4. You then see a wizard with the recommended set of software. Click **Next** and follow the directions to install the software. The CompactRIO system needs the same software as your host computer along with a few additional components. The version of software on both the host computer and the CompactRIO system has to be the same. The CompactRIO reboots automatically after it has installed the software.

You are now ready to add your CompactRIO system to a LabVIEW Project.

Adding a CompactRIO System to the LabVIEW Project

Follow these steps to create a LabVIEW Project using the RIO Scan Interface.

1. Open an empty project from the LabVIEW **Getting Started** window.

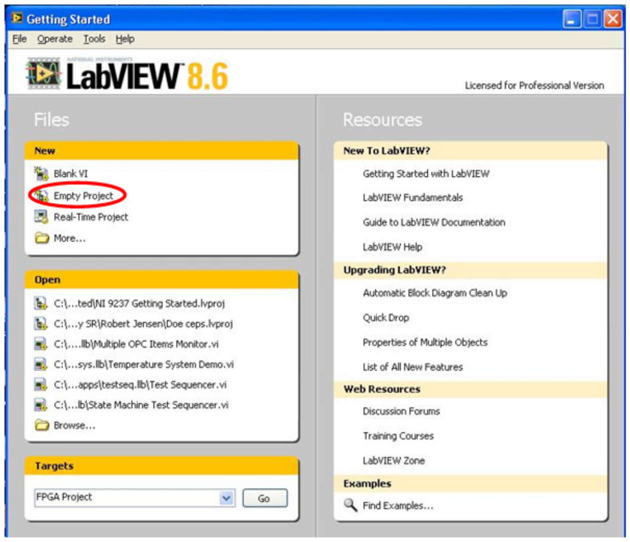

Getting Started
File Operate Tools Help
LabVIEW 8.6
Licensed for Professional Version
Files
New
Blank VI
Empty Project
Real-Time Project
More...
Open
C:\...ted\NI 9237 Getting Started.lvproj
C:\...y SR\Robert Jensen\Doe ceps.lvproj
C:\....lib\Multiple OPC Items Monitor.vi
C:\...sys.llb\Temperature System Demo.vi
C:\...apps\testseq.llb\Test Sequencer.vi
C:\...lb\State Machine Test Sequencer.vi
Browse...
Targets
FPGA Project
Go
Resources
New To LabVIEW?
Getting Started with LabVIEW
LabVIEW Fundamentals
Guide to LabVIEW Documentation
LabVIEW Help
Upgrading LabVIEW?
Automatic Block Diagram Clean Up
Quick Drop
Properties of Multiple Objects
List of All New Features
Web Resources
Discussion Forums
Training Courses
LabVIEW Zone
Examples
Find Examples...

Figure 9.8. Opening an Empty Project from the LabVIEW Getting Started Window

2. Name the project and save it in a folder. Now right-click on the project name and select **New »Targets and Devices**. You see a pop-up window, where you can expand Real-Time CompactRIO to select your remote target and add it to the project. If you do not have the hardware yet you can choose "New target or device" radio button and manually select the type of hardware.

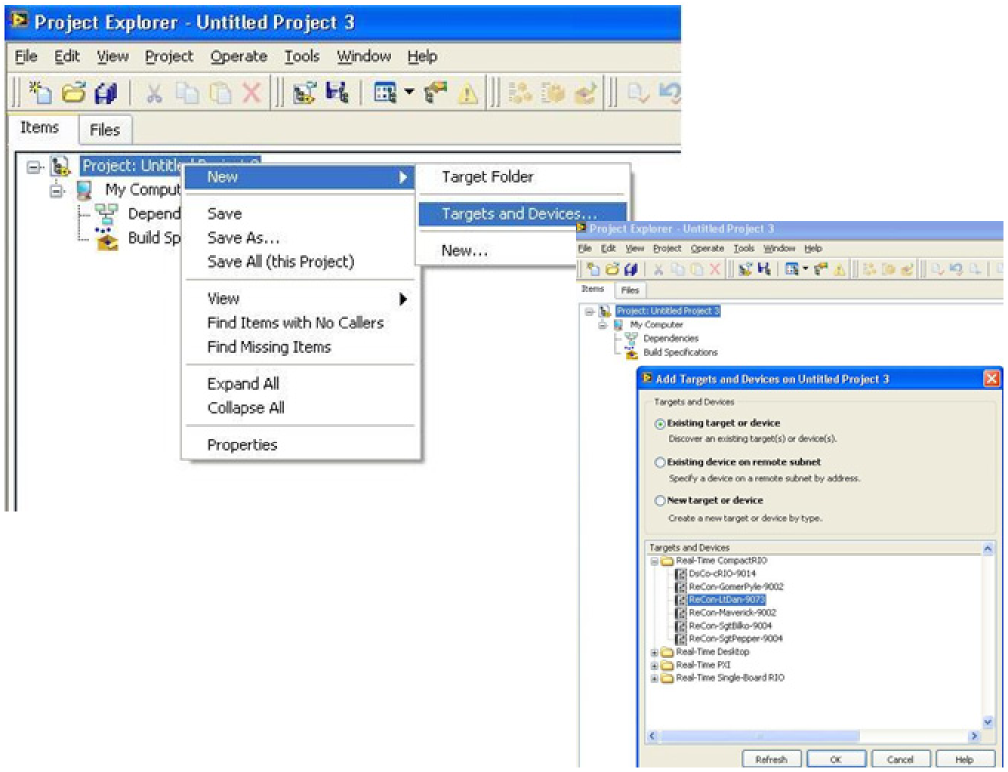


Figure 9.9. Selecting Your Remote Target and Adding It to Your Project

3. If your CompactRIO controller contains modules, you see a dialog box with a prompt asking whether you want to add the modules using the CompactRIO **Scan Mode** or the FPGA Interface. If you choose Scan Mode, the chassis and modules are automatically added to your project.

You are now ready to program your real-time control application or you can open an example program to get started.

Modifying an Existing LabVIEW Project

If you choose to customize an example program to work with your remote target you can right-click on the real-time target in the examples LabVIEW Project Explorer and select Properties. In the General tab, change the IP address in the IP Address/DNS Name to match that of your CompactRIO. Always save a copy of the example by selecting **File »Save As»Duplicate.lvproj file and contents** and saving all the project contents in a separate folder.

APPENDIX B
LabVIEW Debugging Tools

Debugging LabVIEW Applications during Development

LabVIEW has numerous tools to help understand the execution of user code while editing and developing the code. These tools are designed to guide you through the execution and may affect the determinism of the application during debugging. The following tools can be found in the toolbar section of your block diagram and are a good starting place when debugging applications.

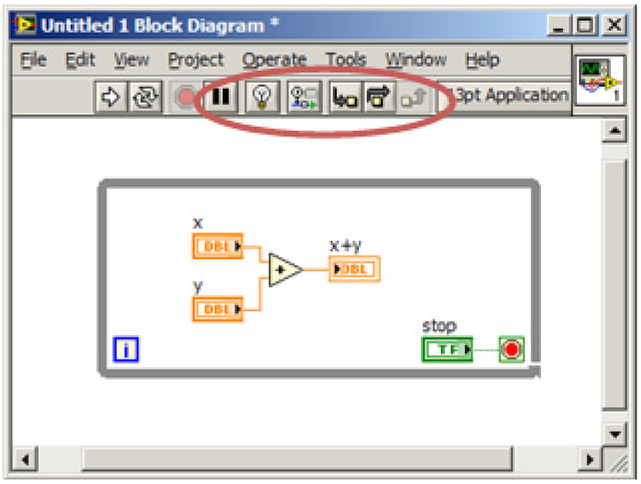
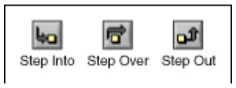

Untitled 1 Block Diagram *
File Edit View Project Operate Tools Window Help
13pt Application
x
y
x+y
stop

Figure 10.1. LabVIEW has multiple debugging tools.

Execution Highlighting

Execution highlighting animates the diagram and traces the flow of data, allowing you to view intermediate values. To view an animation of the execution of the block diagram clicking the **Highlight Execution** button on the toolbar. Next, when clicking the white run arrow, Execution highlighting shows the movement of data on the block diagram from one node to another using bubble that move along the wires. You can use execution highlighting in conjunction with single-stepping to see how data values move from node to node through a VI.

Single-Stepping



Single-step through a VI to view each action of the VI on the block diagram as the VI runs. The single-stepping buttons affect execution only in a VI or subVI in single-step mode.

Enter single-step mode by clicking the **Step Into** or **Step Over** button on the block diagram toolbar. Move the cursor over the **Step Into**, **Step Over**, or **Step Out** button to view a tip strip that describes the next step if you click that button. You can single-step through subVIs or run them normally.

When you single-step through a VI, nodes blink to indicate they are ready to execute. If you single-step through a VI with execution highlighting on, an execution glyph, shown as follows, appears on the icons of the subVIs that are currently running.

Breakpoints

Use the Breakpoint tool, to place a breakpoint on a VI, node, wire, and pause execution at that location. When you set a breakpoint on a wire, execution pauses after data passes through the wire and the Pause button appears red. Place a breakpoint on the block diagram to pause execution after all nodes on the block diagram execute. The block diagram border appears red and flashes to reflect the placement of a breakpoint.

When a VI pauses at a breakpoint, LabVIEW brings the block diagram to the front and highlights the node, wire, or line of script that contains the breakpoint. When you move the cursor over an existing breakpoint, the black area of the Breakpoint tool cursor appears white.

When you reach a breakpoint during execution, the VI pauses and the Pause button appears red. Also, the VI background and border begin flashing. You can take the following actions:

- Single-step through execution using the single-stepping buttons.
- Check intermediate values on probes that you placed on wires prior to running the VI.
- Check intermediate values on probes that you place after running the VI if you have enabled the Retain Wire Values option.
- Change values of front panel controls.
- Click the Pause button to continue running to the next breakpoint or until the VI finishes running.

You can use the Breakpoint Manager window to disable, enable, clear, or locate existing breakpoints. Open the Breakpoint Manager window by selecting **View»Breakpoint Manager** or by right-clicking an object on the block diagram and selecting **Breakpoint»Breakpoint Manager** from the shortcut menu. You can remove breakpoints individually or throughout the entire VI hierarchy.

Probe Tool

Use the Probe tool, to check intermediate values on a wire as a VI runs. Use the Probe tool if you have a complicated block diagram with a series of operations, any one of which might return incorrect data. Use the Probe tool with execution highlighting, single-stepping, and breakpoints to determine if and where data is incorrect. If data is available, the probe immediately updates and displays the data during execution highlighting, single-stepping, or when you pause at a breakpoint. When execution pauses at a node because of single-stepping or a breakpoint, you also can probe the wire that just executed to see the value that flowed through that wire.

Disabling Code

Developers can also disable entire sections of graphical code in a method analogous to commenting out code in a text-based programming language. Disabling parts of the code without rendering the entire program inoperative is a valuable troubleshooting technique for dealing with nonfunctional or faulty code. LabVIEW provides this functionality through the Diagram Disable Structure (Figure 10.2). Each Diagram Disable Structure contains two or more frames – at least one frame contains disabled code, which is wholly ignored by the LabVIEW compiler and not executed when the VI runs. A single frame can hold code that executes when the VI runs.

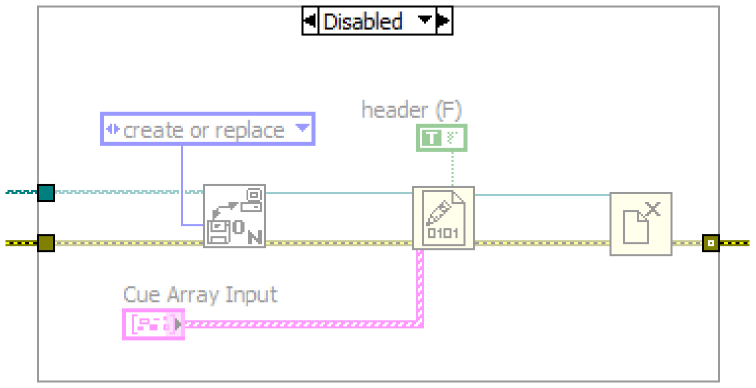
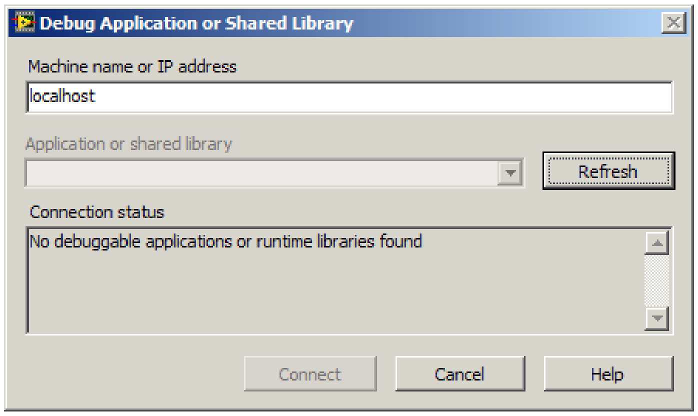


Figure 10.2. LabVIEW Diagram Disable Structure

Debugging Deployed Applications

Debug Application Dialog

With LabVIEW you can also debug applications that are deployed and running embedded. For this you can use the Debug Application or Shared Library dialog box to debug stand-alone real-time applications running on a real-time target. With the Debug Application dialog box, you also can view the block diagram of the application, run the application in "highlight execution" mode, probe inputs and outputs, and allow the use of other debugging tools. You can get to the Debug Application or shared library from the Project Explorer window by selecting **Operate»Debug Application** or **Shared Library** to display the Debug Application or Shared Library dialog box.


Debug Application or Shared Library
Machine name or IP address
Application or shared library
Refresh
Connection status
Connect
Cancel
Help

localhost

No debuggable applications or runtime libraries found

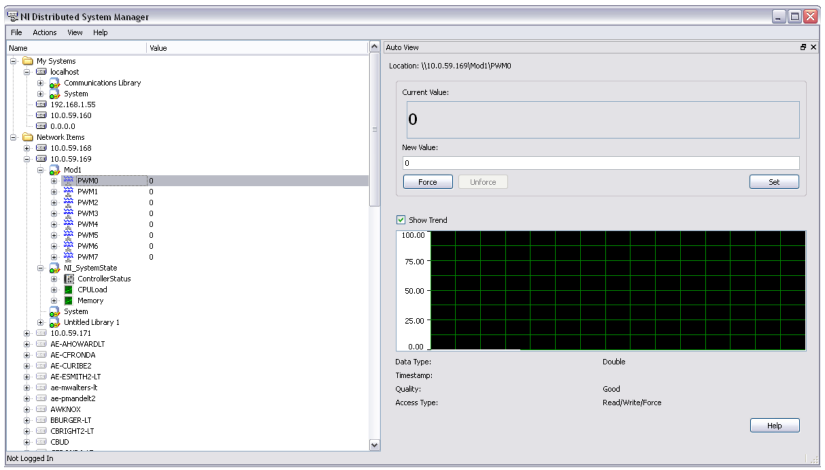
Figure 10.3. Debug Application or Shared Library Dialog

To enable an application for debugging, you must modify the build specification inside the LabVIEW Application Builder. Complete the following steps to debug a stand-alone real-time application.

1. Enable debugging in the build specification for the stand-alone real-time application by placing a checkmark in the Enable debugging checkbox on the Advanced page of the Real-Time Application Properties dialog box.
2. Right-click the build specification for the stand-alone real-time application and select Build from the shortcut menu to build the application.
3. Reboot the real-time target to run the stand-alone real-time application.
4. From the Project Explorer window, select **Operate»Debug Application** or **Shared Library** to display the Debug Application or Shared Library dialog box.
5. Enter the IP address of the real-time target in the Machine name or IP address text box. Click the Refresh button to view the list of stand-alone real-time applications that you can debug on the real-time target.
6. Select the stand-alone real-time application you want to debug.
7. Click the Connect button to open the front panel of the startup VI for debugging. Use the block diagram of the startup VI to debug the application using probes, breakpoints, and other debugging techniques.
8. After you finish debugging, close the startup VI, which also closes the stand-alone real-time application on the real-time target and then reboot the real-time target to restart the stand-alone real-time application. If you want to disconnect without closing the startup VI, right-click the front panel of the startup VI and select Remote Debugging»Quit Debug Session from the shortcut menu.

NI Distributed System Manager

The NI Distributed System Manager provides a central location for monitoring systems on the network, viewing and clearing faults, and managing published data. The NI Distributed System Manager is a stand-alone tool and does not require LabVIEW to be installed on the same computer. You can open the NI Distributed System Manager from the Windows Start menu or by right-clicking on the real-time target in the Project Explorer Window and selecting **Tools»Distributed System Manager**.



```
Name                         Value
My Systems
  localhost
    Communications Library
    System
  192.168.1.55
  10.0.59.160
  0.0.0.0
Network Items
  10.0.59.168
  10.0.59.169
    Mod1
      PWM0                   0
      PWM1                   0
      PWM2                   0
      PWM3                   0
      PWM4                   0
      PWM5                   0
      PWM6                   0
      PWM7                   0
    NI_SystemState
      ControllerStatus
      CPULoad
      Memory
    System
    Untitled Library 1
  10.0.59.171
  AE-AHOWARDLT
  AE-CFRONDA
  AE-CURIBE2
  AE-ESMITH2-LT
  ae-mwalters-lt
  ae-pmandelt2
  AWKNOX
  BBURGER-LT
  CBRIGHT2-LT
  CBUD
Not Logged In
```

```
Location: \\10.0.59.169\Mod1\PWM0
Current Value: 0
New Value: 0
Show Trend
100.00
75.00
50.00
25.00
0.00
Data Type:      Double
Timestamp:
Quality:        Good
Access Type:    Read/Write/Force
```

Figure 10.4. The NI Distributed System Manager provides network viewing of I/O and controller states.

The system manager offers test panels for CompactRIO modules running in scan mode. As soon as your system is available on the network, you have access to real-time and historical-trend I/O values, so you can quickly verify your connections and signal integrity. In addition to test panels, the system manager also gives visibility into memory usage and processor load for CompactRIO controllers. You can also use the NI Distributed System Manager to monitor and manage scan engine faults and modes.

User-Controlled LEDs

Most real-time controllers have one or more LEDs that can be controlled from a user application. A designer can use these LEDs to indicate the state of a running application. A common practice is to blink an LED at different rates to indicate the application is running and at what state or function the application is currently in or performing. LabVIEW Real-Time provides an RT LEDs VI for communicating with user LEDs on a Real-Time target.

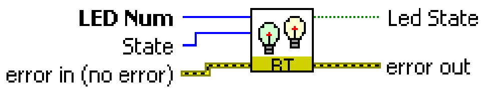


Figure 10.5. LEDS can provide a simple view of program status.

A user can change the state of an LED by executing the RT LEDs VI on a block diagram and setting the state input to the following values:

0	Turns on the LED
1	Sets the LED to default color1
2	Sets the LED to default color2
3	Toggles between off and default color1

Table 10.1. LED State Inputs

Additionally, some real-time controllers provide access to LEDs from the FPGA application. In this case, the LabVIEW FPGA application can be written to using a LabVIEW FPGA I/O node that is by default named FPGA LED which when written can change the state of the LED from off based on the Boolean control value.

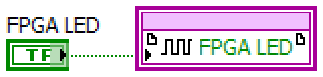


Figure 10.6. You also can access LEDs from the FPGA.

CompactRIO Console Out Switch

CompactRIO controllers also provide a Console Out switch provides useful diagnostic information from the controller's COM1 port when it is connected to a computer using serial port. This functionality is particularly useful for troubleshooting systems in the following scenarios:

- Displaying printf style information to a text console from the controller.
- Displaying the controller's current firmware version and IP address.
- Diagnosing an unresponsive controller or a controller with errors indicated by the LEDs.
- Troubleshooting a controller that is not showing up in MAX.
- Assisting an NI Applications Engineer with troubleshooting the controller.

To view the output from the Console Out on a CompactRIO controller, follow the following procedure:

1. Shut down the controller in question.
2. Connect the controller to a PC using a null modem cable (NI controllers are DTE devices).
3. Apply the following serial port settings on the PC:
   a. Bits per second: 9600
   b. Data bits: 8
   c. Parity: None
   d. Stop bits: 1
   e. Flow control: None
4. On the controller, set the Console Out DIP switch to the ON position.
5. Power on the controller, and observe the output on the PC's terminal window from the controller.

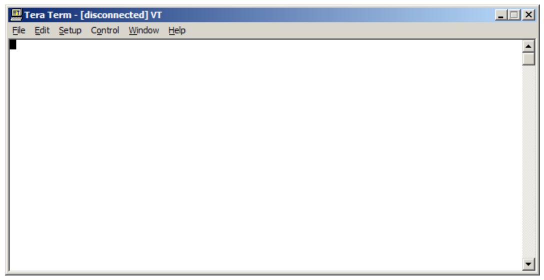

Tera Term - [disconnected] VT
File Edit Setup Control Window Help

Figure 10.7. Console Out provides a simple serial interface to the controller.

6. Set the Console Out switch back to OFF when done troubleshooting.

To send your own debugging information through the console out, LabVIEW Real-Time provides an RT Debug String VI for communicating textual information from the controller to the terminal window on a host PC.

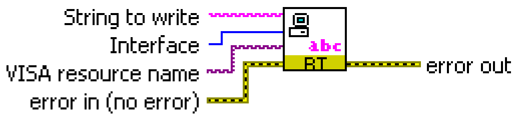


Figure 10.8. You can customize your application to send additional data to the PC via the Console Out feature.

Error Logging

Since CompactRIO applications often do not have a user interface there may not be an indication of when an error occurs. Therefore, it is important to store these errors in a log file on the controller that can later be retrieved. You can use error log files to store information about LabVIEW errors and/or unexpected data. You can set-up a separate lower priority loop that logs errors or log errors as part of the shut-down routine. A simple error logging implementation is shown below.

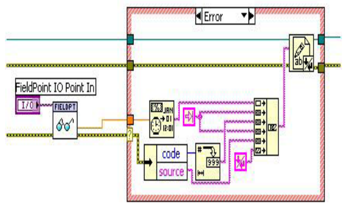


Figure 10.9. Simple Error-Logging Implementation

In addition, real-time controllers automatically generate an error log if the controller crashes. You can access this error log by right-clicking on the controller in MAX under Remote Systems and selecting View Error Log. The error log is stored in the following directory on the controller: /ni-rt/system/errlog.txt.

LabVIEW Analysis Tools

When building your application it is recommended you analyze your VIs to calculate execution time and to check for unwanted behavior such as memory allocation or thread switching.

Analyzing VIs and Sections of Code

Real-Time Benchmarking VIs

Benchmarking an operation involves measuring the time at the start of the operation, performing the operation, and measuring the time at the end of the operation. Since timing is crucial in a deterministic application, LabVIEW Real-Time ships with a Real-Time Benchmarking VI to help verify correct timing behavior.

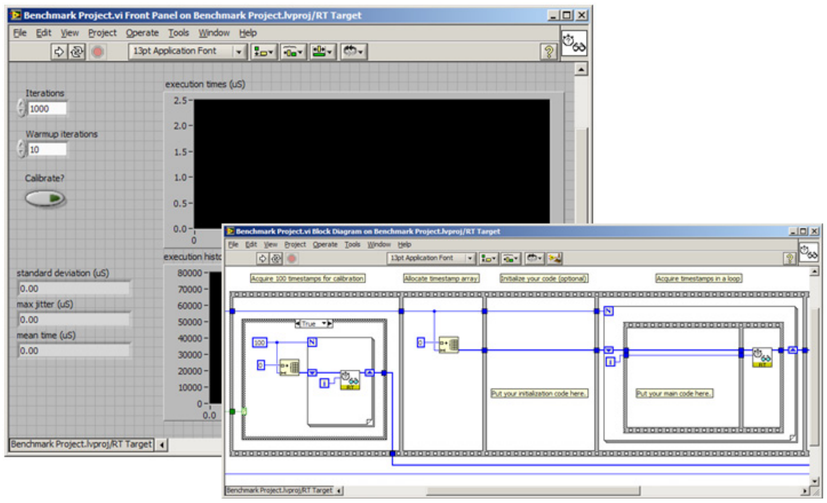
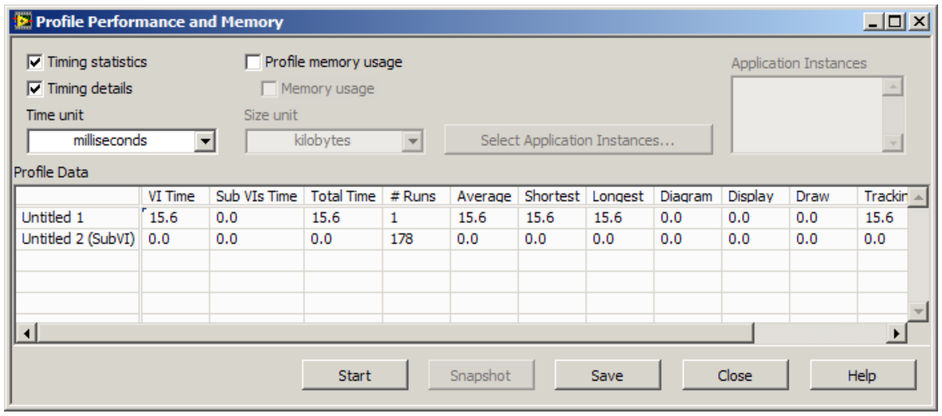

Benchmark Project.vi Front Panel on Benchmark Project.lvproj/RT Target
Iterations
1000
Warmup iterations
10
Calibrate?
execution times (uS)
standard deviation (uS)
0.00
max jitter (uS)
0.00
mean time (uS)
0.00
Benchmark Project.vi Block Diagram on Benchmark Project.lvproj/RT Target
Acquire 100 timestamps for calibration
Allocate timestamp array
Initialize your code (optional)
Acquire timestamps in a loop
Put your initialization code here.
Put your main code here.

Figure 10.10. The Real-Time Benchmark VI can help you understand execution time for subVIs.

The Benchmark Project VI uses the RT Get Timestamp VI and the RT Timestamp Analysis VI to benchmark the performance of VIs and sections of VIs running on a real-time target. You can use the benchmark information to optimize the design of real-time target VIs. See the NI Example Finder to locate the Real-Time Benchmarking VIs.

Profile Performance and Memory Window

The Profile Performance and Memory window is a tool for statistically analyzing how an application uses execution time and memory. You can use the Profile Performance and Memory window to display performance information for all VIs and subVIs in memory. This information can help you optimize the performance of your VIs by identifying potential bottlenecks. For example, if you notice that a particular subVI takes a long time to execute, you can improve the performance of that VI.


Profile Performance and Memory
Application Instances
Select Application Instances...
Start
Snapshot
Save
Close
Help

☑ Timing statistics
☑ Timing details
☐ Profile memory usage
☐ Memory usage
Time unit: milliseconds
Size unit: kilobytes

Profile Data

	VI Time	Sub VIs Time	Total Time	# Runs	Average	Shortest	Longest	Diagram	Display	Draw	Trackir
Untitled 1	15.6	0.0	15.6	1	15.6	15.6	15.6	0.0	0.0	0.0	15.6
Untitled 2 (SubVI)	0.0	0.0	0.0	178	0.0	0.0	0.0	0.0	0.0	0.0	0.0

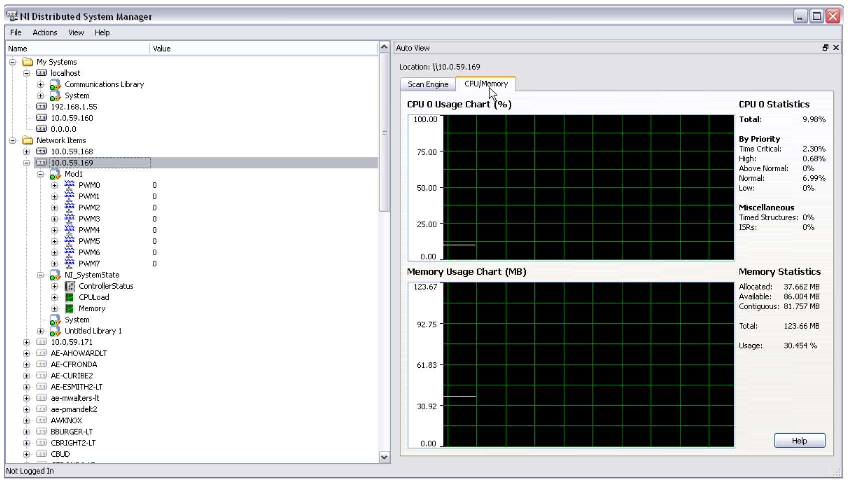
Figure 10.11. The Profile Performance and Memory Window provides timing and memory details for applications and subVIs.

To launch the Profile Performance and Memory application, select **Tools»Profile»Performance and Memory**.

Monitoring Target Resources in Real Time

NI Distributed System Manager and Real-Time System Manager

It is helpful to view the memory and CPU resources on a controller executing deployed code. In some cases, application timing failures are caused by insufficient memory or CPU resources on the real-time target. By monitoring the target resources, you can determine if a memory leak is occurring over time (normally caused by performing memory allocations in a loop) or how much CPU time different loops are using. The NI Distributed System Manager displays controller CPU and memory usage as shown in Figure 10.12.

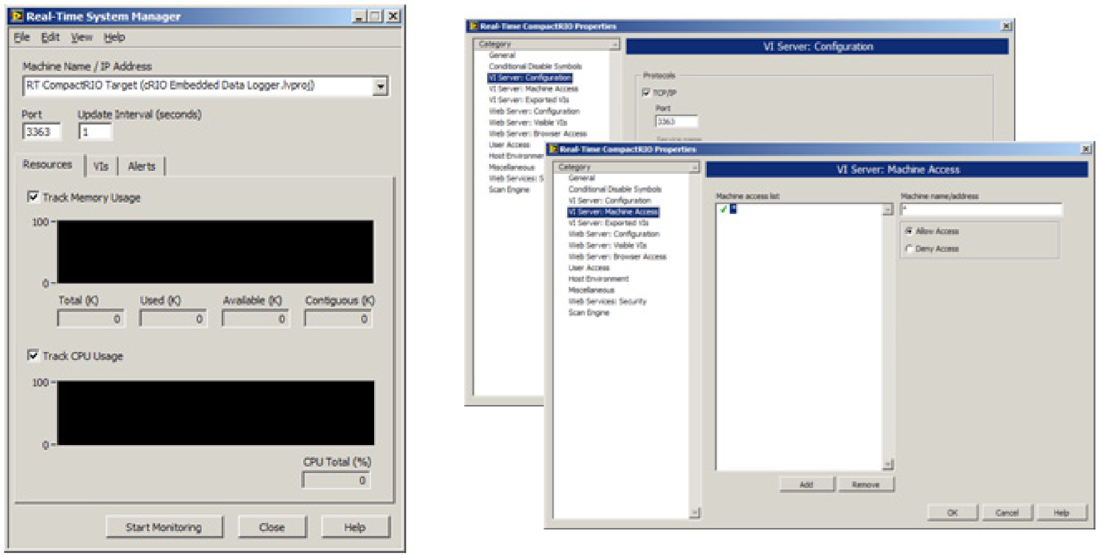


CPU 0 Statistics
Total: 9.98%
By Priority
Time Critical: 2.30%
High: 0.68%
Above Normal: 0%
Normal: 6.99%
Low: 0%
Miscellaneous
Timed Structures: 0%
ISRs: 0%

Memory Statistics
Allocated: 37.662 MB
Available: 86.004 MB
Contiguous: 81.757 MB
Total: 123.66 MB
Usage: 30.454 %

Figure 10.12. The NI Distributed System Manager exposes CPU and memory usage for controllers on the network.

You can also use a tool called the NI Real-Time System Manager to view CPU and memory usage.



Machine Name / IP Address: RT CompactRIO Target (cRIO Embedded Data Logger.lvproj)
Port: 3363 Update Interval (seconds): 1
Track Memory Usage
Total (K): 0 Used (K): 0 Available (K): 0 Contiguous (K): 0
Track CPU Usage
CPU Total (%): 0

Protocols: TCP/IP
Port: 3363

Machine access list: ✓ *
Machine name/address: *
Allow Access
Deny Access

Figure 10.13. The NI Real-Time System Manager also shows CPU and memory usage.

You can launch the NI Real-Time System Manager (RTSM) from the **Tools»Real-Time Module** menu. To configure the RTSM for use, you must first configure the VI Server on the target. To configure the VI Server on the target, first right-click on the real-time target and select properties to bring up the Real-Time Properties page. Highlight the VI Server: Configuration page and ensure the TCP/IP checkbox is checked and the checkboxes in the Accessible Server Resources are checked. On the VI Server Machine Access category, ensure that the IP address of your host PC or an * is present.

Real-Time Execution Trace Toolkit
One of the best tools is the Execution Trace Tool. This is a real-time event and execution tracing tool that you can use to capture and display the timing and event data of VI and thread events for LabVIEW Real-Time applications. You can use the Execution Trace Tool VIs to capture the timing and execution data of VI and thread events for applications running on a real-time target.

With minimal modifications to your embedded code, these tools graphically display multithreaded code executions while highlighting thread swaps, mutexes, and memory allocation. Using this information, you can optimize your real-time code for faster control loops and more deterministic performance.

To use the tool, place Execution Trace Tool VIs in the application running on the real-time target. Once a trace has completed, the VIs send information back to the Execution Trace Tool user interface that is running on the host or can save the information to disk for later view. Below is an example of a trace where the Trace Tool Start Trace VI is placed before the code of interest and the Trace Tool Stop Trace and Send VI sends information to the host after the code is complete.

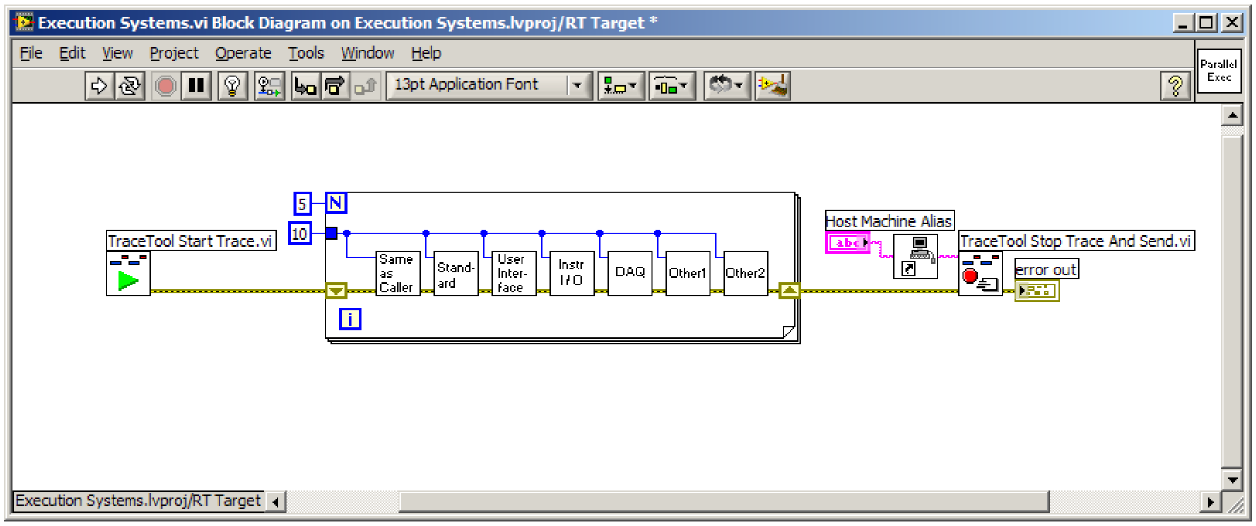

Execution Systems.vi Block Diagram on Execution Systems.lvproj/RT Target *
File Edit View Project Operate Tools Window Help
13pt Application Font
Parallel Exec
TraceTool Start Trace.vi
5
N
10
Same as Caller
Stand-ard
User Inter-face
Instr I/O
DAQ
Other1
Other2
i
Host Machine Alias
TraceTool Stop Trace And Send.vi
error out
Execution Systems.lvproj/RT Target

Figure 10.14. By modifying the program to start the trace log and send it to the PC, you can view low-level details of program execution.

Once the Execution Trace Tool user interface receives the information, you see two simultaneous views of the trace session. The thread view shows when each thread is active. The VI view lists each VI in memory and shows exactly when it executed.

The thread view shows all the threads that are running in the system. This includes threads outside LabVIEW, such as real-time timing and communication threads. The LabVIEW threads are listed as LabVIEW threads, are color-coded to match VI priorities in the VI view, and are labeled according to the LabVIEW execution system in which they reside. The thread view may also show OS, timed loop, and user flags you may have configured.

As mentioned before, the VI view lists each VI in memory and shows exactly when it executed. However, two VIs can overlap in the VI view but cannot execute at exactly the same time because on VI preempts the other. One VI has a higher priority, so its stops the lower priority VI so that it can instantly use the processor. This switch happens so quickly that the lower priority VI cannot log that it is stopping. You can examine the thread view to determine which thread is active, and hence which VI is active.

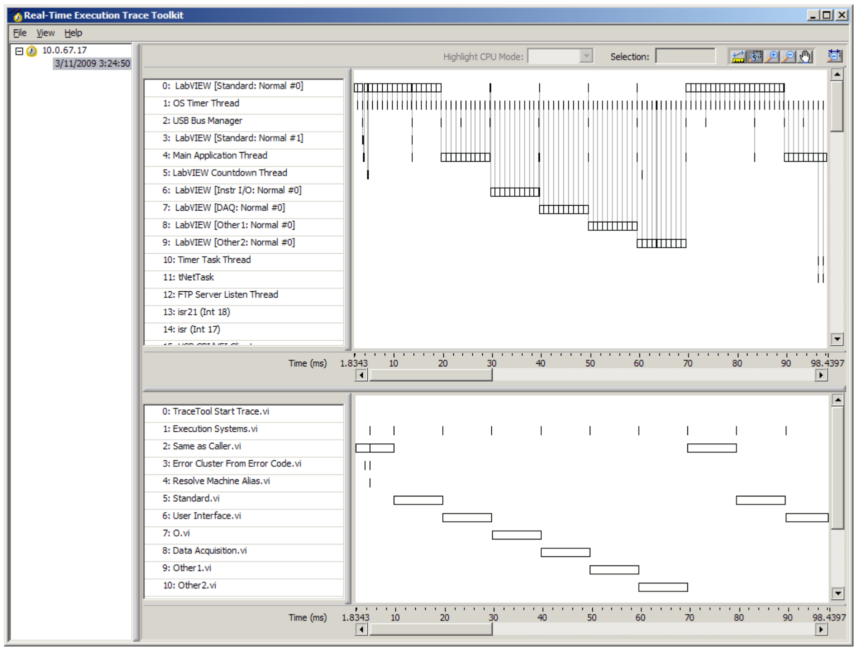

Real-Time Execution Trace Toolkit
File View Help
10.0.67.17
3/11/2009 3:24:50
Highlight CPU Mode:
Selection:
0: LabVIEW [Standard: Normal #0]
1: OS Timer Thread
2: USB Bus Manager
3: LabVIEW [Standard: Normal #1]
4: Main Application Thread
5: LabVIEW Countdown Thread
6: LabVIEW [Instr I/O: Normal #0]
7: LabVIEW [DAQ: Normal #0]
8: LabVIEW [Other1: Normal #0]
9: LabVIEW [Other2: Normal #0]
10: Timer Task Thread
11: tNetTask
12: FTP Server Listen Thread
13: isr21 (Int 18)
14: isr (Int 17)
Time (ms) 1.8343 10 20 30 40 50 60 70 80 90 98.4397
0: TraceTool Start Trace.vi
1: Execution Systems.vi
2: Same as Caller.vi
3: Error Cluster From Error Code.vi
4: Resolve Machine Alias.vi
5: Standard.vi
6: User Interface.vi
7: O.vi
8: Data Acquisition.vi
9: Other1.vi
10: Other2.vi
Time (ms) 1.8343 10 20 30 40 50 60 70 80 90 98.4397

Figure 10.15. The Execution Trace Tool provides low-level detail into application execution, showing thread execution, and preemption and memory allocation.

Often you want to view other events besides when a VI starts and stops. These events may include accessing the memory manager, sleep, iterations of a timed loop, etc. For this, you can use the Real-Time Execution Trace Toolkit to configure and event flag where you can configure flags based on the type of event by going to **View»Configure Flags** in the Execution Trace Tool. Refer to the LabVIEW help for more detailed information.

Debugging LabVIEW FPGA

Development with LabVIEW FPGA is very similar to development with LabVIEW Real-Time. The same graphical programming constructs apply to all platforms and the LabVIEW FPGA functions are for the most part a subset of the functions found in Real-Time. However, to optimize your LabVIEW FPGA application you can employ different techniques leveraging hardware concepts like registers, pipelining, clock cycles, and communication between asynchronous modules.

How FPGA Development Is Different from LabVIEW Real-Time

There are three main reasons that make the development techniques different between FPGA hardware and computer software.

Compilation Time

Creating the executable code for an FPGA requires a complex compilation that performs optimization top fit all the logic on the chip. Depending on the sophistication of the application, the compilation takes varying amounts of time.

Absence of Typical LabVIEW Debugging Features in Hardware

Once the code is running in hardware, traditional software debugging features such as the ability to probe, single-step, process highlight, set breakpoints, are no longer available. You can use simulation techniques to more easily debug and develop code for your FPGA application.

FPGAs Execute Fast and Deterministically

FPGAs are chosen for applications requiring high speed and determinism. Subsequently, designers are often concerned with exactly what is happening at every clock cycle of the FPGA, either for pure performance reasons or to characterize synchronization between parallel tasks.

Because it can take anywhere from five minutes to multiple hours for a program to be compiled and run, it is impractical to employ a "code and fix" method of programming. To become a more proficient FPGA programmer it is important to understand how to increase your development efficiency. The sections below examine how to use behavioral simulation techniques as well as post-compile debugging to create a better-tested and quicker FPGA system.

LabVIEW FPGA Post-Compile Debugging Techniques

Use Controls/Indicators for Debugging

Controls and indicators provide a good mechanism for debugging code while the program is running in the FPGA. Since you cannot use LabVIEW probes, use an indicator to get the instantaneous value of a wire of interest. Additionally, use a control instead of a constant during this process because changing a constant value requires a recompile. Use a control, find the right value, and change to a constant when you are sure it is correct. However, controls and indicator take a sizable amount of resources on the FPGA, because they create a communication interface with the host VI. Stripping away unnecessary controls and indicators is the first step in resource optimization. Use the additional controls and indicators while debugging, but plan to remove them as you verify your design.

Latch Error Conditions

Many times there are error conditions that only happen for a short time. The worst case is that the error occurs in only one cycle of the clock. While debugging, these error conditions happen much too quickly to catch them through the communication interface. The best way to catch the error is to latch the condition when it happen and then allow a reset if needed. After you get things working correctly you might be able to take out the debugging latch and handle the error in a different manor for the deployed system. The classic example of this is debugging a periodic timeout from a FIFO.

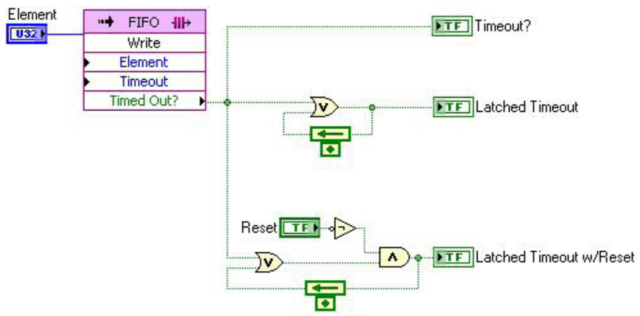


Figure 10.16. This VI shows latching a timeout condition with and without reset.

Data Log a Test Point with DMA

Sometimes you want to probe something in the FPGA but the instantaneous value from a control is not enough to debug. In this case, you can use a DMA channel for debugging purposes to actually capture the entire waveform of data from a test point.

Route a Test Point to an I/O

Whether it is an analog or digital test point, you can always route the LabVIEW wire to an I/O node. Once the signal is routed from the FPGA, you can debug the signal from any other NI test hardware such as a scope, digital card, DAQ board, or even another FPGA programmed for testing.

Test Multiple Candidate Implementations with a Case Structure

In some cases, you want to test a couple of ways to implement a task in the FPGA. You can compile each of them to see which one performs better. However, in order to avoid the extra compiles, put the test implementations into cases of a case structure and select between them on the fly. You can get instant feedback on the comparison between the cases and avoid extra compilation time.

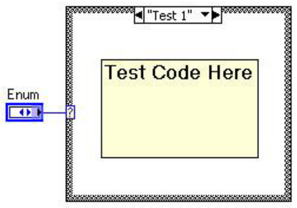


Figure 10.17. Place a case structure with the different code you want to try out on the FPGA.

FPGA Behavioral Simulation on the Development Computer

Because you are just using LabVIEW code when making FPGA logic, it is always possible to execute your VIs on the host computer. This means you can use all the debugging features of LabVIEW and you do not have to wait for it to compile every time you need to test some logic. Additionally, you can create a Testbench VI to assert the inputs that normally are connected to the outside world with custom user-defined I/O and capture the outputs for analysis and verification. Finally, you can run the host program simultaneously with the FPGA code including simulated registers and DMA FIFO memory buffers.

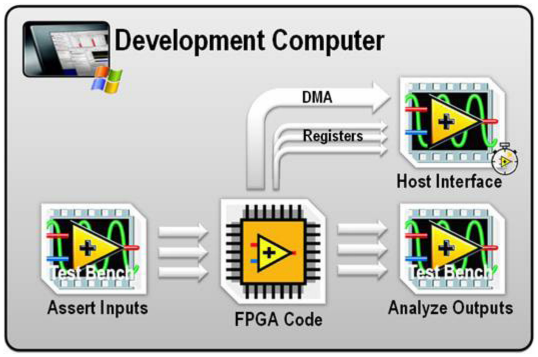
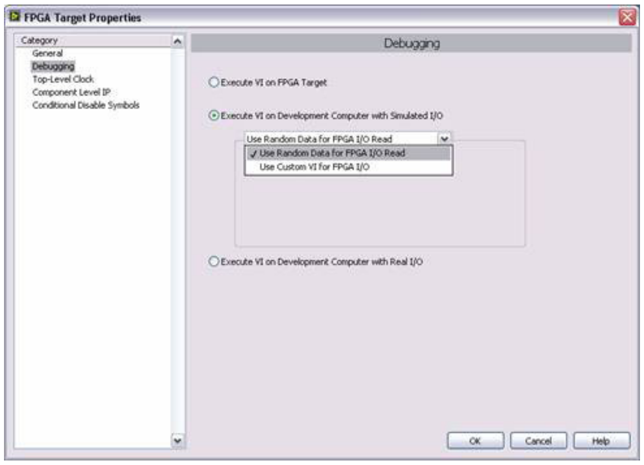


Figure 10.18. Conceptual Diagram of the Simulation System with a Testbench VI to Supply Custom Input and Capture Outputs with the Host Interface Running at the Same Time

To set the FPGA to run on the development computer, right-click the FPGA in the project and select "Properties…" In the debugging section, you have three possible options.

1. **Execute VI on FPGA target** – Leaving this option selected begins the compile process when you press the "Run" arrow for an FPGA VI.
2. **Execute VI on the development computer with simulated I/O** – This option sets the FPGA VI to execute on the PC when you press the run arrow. In the drop-down menu, you can select to use either random data (which was the main behavior in LabVIEW 8.5 and earlier) or custom I/O, so you can write a Testbench VI to assert the inputs and capture outputs.
3. **Execute VI on the development computer with real I/O** – At this time, this feature is available for only R Series plug-in devices. This option runs the VI on the PC and downloads a fixed personality to the R Series device to sample the I/O when the program executes the I/O Node. This is useful for early testing and prototyping, but keep in mind that the I/O is software-timed sampling and most likely does not represent the timing you really want your VI to use.


FPGA Target Properties

Category: General, Debugging, Top-Level Clock, Component Level IP, Conditional Disable Symbols

Debugging

○ Execute VI on FPGA Target
◉ Execute VI on Development Computer with Simulated I/O
Use Random Data for FPGA I/O Read
✓ Use Random Data for FPGA I/O Read
Use Custom VI for FPGA I/O
○ Execute VI on Development Computer with Real I/O

Figure 10.19. Properties Windows for the FPGA Showing the Three Options for Running the FPGA Code and I/O Type

APPENDIX C
Large Application Development Best Practices

Large Application Development Guidelines

Proper software engineering practices are critical to the successful development of a CompactRIO control application. Any project should start with clearly defined and documented requirements. These requirements lead to an architectural selection and then to development. The code is then validated against the requirements and finally deployed and maintained.

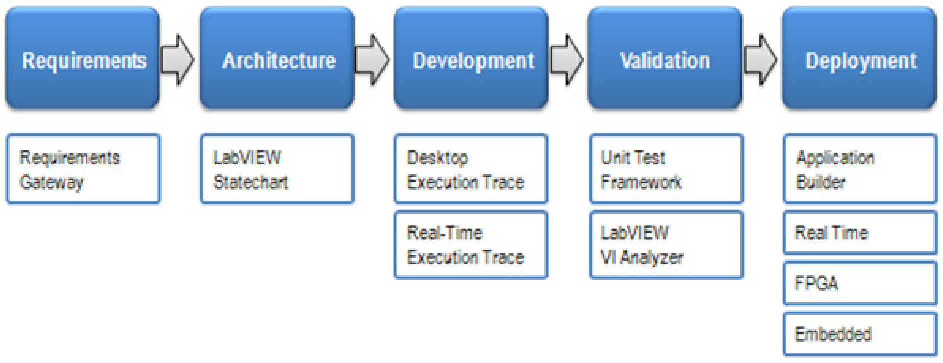


Figure 11.1. Proper software engineering practices are critical to successful project completion. National Instruments has toolkits that can help at each step.

Because creating applications in LabVIEW is so easy, many people focus on just the development step and begin to develop VIs immediately with relatively little planning. For simple applications, such as quick lab tests or monitoring applications, this approach can be appropriate. However, for larger development projects, good project planning is vital. This section presents fundamental techniques for successful software development in LabVIEW. For more detailed information, read the LabVIEW Development Guidelines Manual.

Applying a Structured Software Engineering Approach

Programmers creating mission-critical applications – embedded control applications, industrial monitoring applications, and high-performance test systems – cannot afford to introduce errors or uncertainty into their systems. For these kinds of applications, engineers must follow a very structured, and sometimes externally certified, programming process to ensure code quality and repeatability. Generally speaking, the situations when engineers need a structured programming process include the following:

- Building an especially large application
- Working as a team on the code, requiring a structured approach to ensure team members can safely and efficiently work on different areas of the project without conflicts
- Working in an industry or application area that requires coding processes to be certified by a government agency (FDA, FAA, and so on) or by a customer (automotive manufacturer working with suppliers)

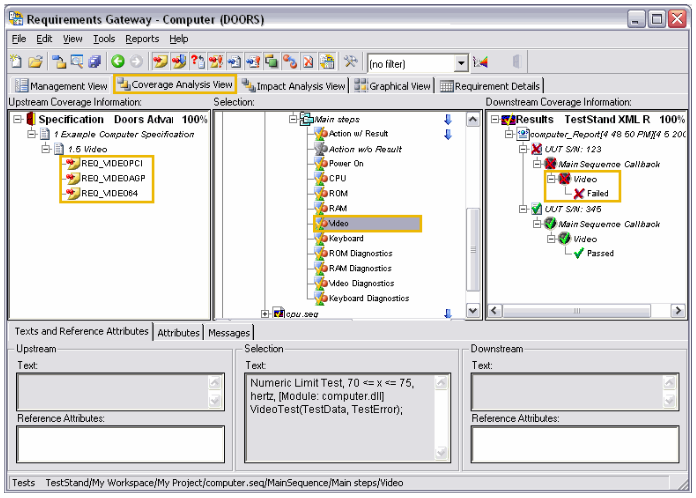
Many engineers may think that these situations require a traditional, text-based programming language to develop code. In reality, a structured process for developing code – in any language – is independent of the language or tool used. This section outlines common practices, built-in tools, and integration with external products that show how engineers can build large-scale or mission-critical systems requiring structured programming approaches.

Requirements Collection and Management

A key step in developing any large system is capturing the system requirements up front, before any serious development begins. System developers use many different approaches for capturing and documenting system requirements, from simple Microsoft Word or Excel documents to dedicated requirements management tools such as Telelogic DOORS or IBM Rational RequisitePro. The key to managing requirements is to clearly document what they are and associate them with the actual code that implements each.

To help with this process National Instruments created a product, the NI Requirements Gateway, where engineers can associate their requirements with specific LabVIEW VIs to track their implementation. With the gateway, engineers can do the following:

- Document requirements using text files; Microsoft Word, Excel, or Project; Telelogic DOORS; IBM RequisitePro; and other tools. Engineers also can use the out-of-the-box requirements manager that comes with the toolkit.
- Link each requirement to specific LabVIEW VIs or sections of code to easily track how each requirement is being implemented or tested.
- Trace requirements as they change. With NI Requirements Gateway, engineers can quickly see when a requirement changes and be alerted as to which VIs are associated with that requirement and must be verified to make sure they still meet the requirement.



```
Numeric Limit Test, 70 <= x <= 75,
hertz, [Module: computer.dll]
VideoTest(TestData, TestError);
```

```
Tests   TestStand/My Workspace/My Project/computer.seq/MainSequence/Main steps/Video
```

Figure 11.2. NI Requirements Gateway

Architecture Planning and Design

This document presents a scalable architecture that is appropriate for many control applications. However to successfully implement the architecture you must plan your software framework to meet your specified requirements. Your plan should define the system components and the interaction of those components. You should spend adequate time planning how to optimally convert your system requirements into code.

Designing large applications in LabVIEW, which is no different than software design in any other language, entails dividing an application into logical pieces of manageable size. LabVIEW graphical programming can make top-down design of an application straightforward and intuitive. In addition to graphical programming, LabVIEW provides methods you can use to create solutions. These include:

- Statecharts
- Simulation diagrams

Development

Organizing Files on Disk

File organization on disk should not be an afterthought. Poor planning for large applications leads to additional time spent moving and renaming files during development. Because LabVIEW VIs are linked by their name and path, when you move or rename subVIs you can break the links which then must be manually reestablished. Proper management of files on disk mitigates the risk of moving large sets of files at a later date and can help developers easily find files and determine where to save new files.

Many software developers already have practices and systems in place to determine where files should be stored. There are many legitimate practices and structures, but the following are established common practices that have proven to scale well for large applications.

- Store all project files within a single directory
- Create folders within that directory for logical groupings of files
- Group or "bucket" files according to predetermined criteria
- Divide your application into logical pieces of manageable size
- Use logical and descriptive naming conventions
- Separate the top-level VIs from other source code

Folders on the hard drive are commonly used to group or categorize files and thereby separate subVIs from callers. The criteria typically used for grouping these files are a combination of the file functionality, type, and hierarchical layer in the application. In fact, organization on disk should be a physical manifestation of the relationship of files and code in the application.

Using the LabVIEW Project to Manage Files

The LabVIEW Project offers tools to help developers manage their files. As an application grows larger, developers need to manage numerous files associated with the application, such as VIs, documentation, third-party libraries, data files, and hardware configuration settings. Engineers can use the **LabVIEW Project Explorer** to manage such files.

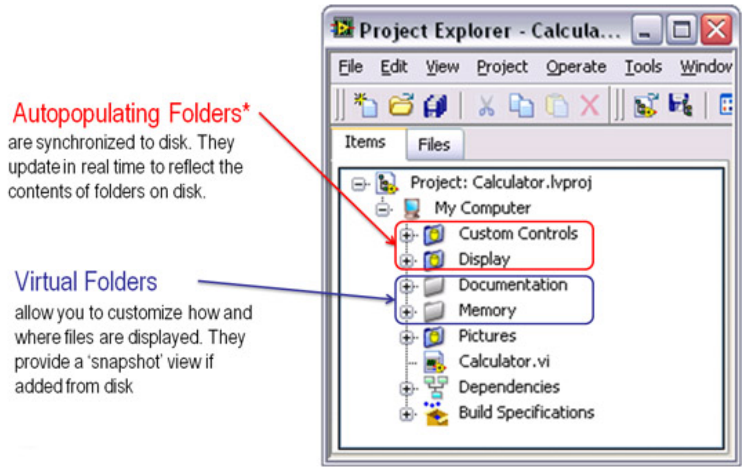

Autopopulating Folders*
are synchronized to disk. They update in real time to reflect the contents of folders on disk.
Virtual Folders
allow you to customize how and where files are displayed. They provide a 'snapshot' view if added from disk
Project Explorer - Calcula...
File Edit View Project Operate Tools Windov
Items Files
Project: Calculator.lvproj
My Computer
Custom Controls
Display
Documentation
Memory
Pictures
Calculator.vi
Dependencies
Build Specifications

Figure 11.3. The LabVIEW Project

Within a LabVIEW Project, developers can use folders to organize all the files that compose their applications. Project folders are virtual by default, but developers can synchronize them with physical directories in the file system. Once developers add a folder to a LabVIEW Project, they can change it from virtual to autopopulating (or vice versa) to maximize their file management and organization flexibility. Autopopulating folders are synchronized to a physical folder, which means that they always display any changes or modifications made to this location outside of the LabVIEW development environment. Autopopulating folders enforce a linkage between your organization of the files on disk and the logical grouping in the Project. Use autopopulating folders whenever possible to preserve the disk hierarchy within the LabVIEW Project Explorer.

Team-Based Development

1. Often multiple developers use LabVIEW when working together on the same project. In these cases, it is important to clearly define the interfaces to the code and identify coding standards to ensure the different components are easily understood and work well together. Developers can achieve this by:Making code modular and providing standard interfaces
2. Storing master copies of the project files, including VIs, on a single server and instituting a source code control policy
3. Implementing coding standards within a group or company

Code Modularity in LabVIEW

Large LabVIEW applications require a modular development approach in which the overall system is divided into logical components. Multiple developers working together on a project need this modular approach and clearly defined programming responsibilities. LabVIEW is, by definition, a modular programming language. Each LabVIEW VI (or unit of code) can be run as a stand-alone application or be called by another VI as a subVI. A subVI corresponds to a subroutine call in text-based programming languages or function/function block in 1131 programming. Using modular subVIs as part of a larger application simplifies the high-level VI block diagram and helps in the management of changes and system debugging.

Sharing Code with Source Code Control Software

At the beginning of a software development project, engineers must implement a process to deal with changes and share work. This process is critical for projects with multiple developers working collaboratively. Source code control tools are the best solution to the problem of sharing code and controlling access to avoid accidental loss of data. Source code control software tracks changes to code modules and facilitates file sharing among multiple users and software projects. In addition to maintaining source code like VIs, the source code control software can manage other aspects of a software project such as feature specifications and other documents.

LabVIEW provides integration with many industry-standard source code control providers such as **Microsoft Visual SourceSafe**, Perforce, **IBM Rational ClearCase**, PVCS Version Manager, MKS Source Integrity, and free, open-source software such as **CVS**. Through this integration, developers can check files in and out of source code control from within the LabVIEW environment, as well as obtain the revision history of the files. To check a VI in or out in LabVIEW once the source code control provider has been configured, a developer simply right-clicks on the file in the LabVIEW Project Explorer and selects the appropriate action from the right-click menu.

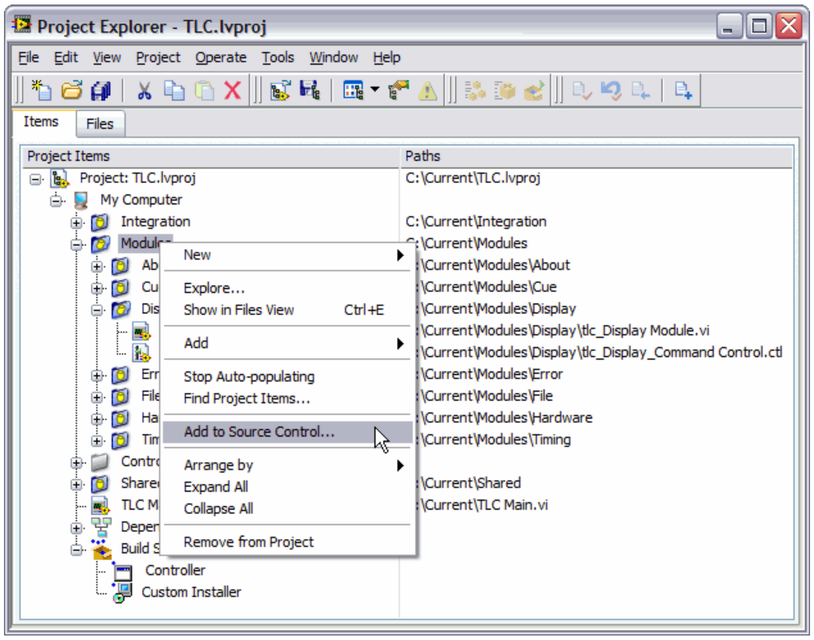


Figure 11.4. Source Code Control Options in the LabVIEW Project

Debugging

Debugging capabilities are key to a development environment, and LabVIEW has numerous debugging tools including:

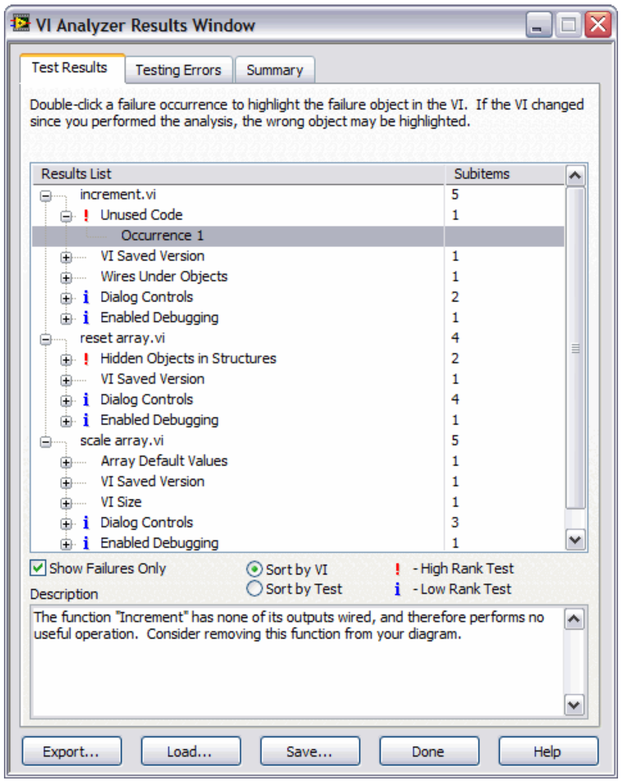
- Execution highlight, which shows the movement of data on the block diagram from one node to another using bubbles that move along the wires
- Single-stepping, which steps through a VI and shows each action of the VI on the block diagram as the VI runs
- Breakpoints, which can be placed on a VI, node, or wire on the block diagram to pause execution at that location
- Probes, which display intermediate values on a wire as a VI runs

A separate section of this document goes into more detail on tools and techniques for LabVIEW Real-Time and FPGA application debugging.

Validation and Reviewing Code

Developers using LabVIEW to create large applications are encouraged to submit their code to peer reviews. When a developer sits down for a code review, the developer walks the reviewer through the main path of the code and answers any questions. Some of the possible discussion topics include how easy the code architecture makes adding a new feature or implementing a change, how errors are reported and handled, and whether the code is modular enough.

To prepare for a code review, developers need to take advantage of tools that can help automate code inspection and identify improvements. One example is the **LabVIEW VI Analyzer** which analyzes LabVIEW code and steps the user through the test failures. The VI Analyzer contains more than 60 tests that address a wide range of performance and style issues. With the VI Analyzer, developers can enhance VI performance, usability, and maintainability. The VI Analyzer also generates reports so that developers can track code improvements over time.



Double-click a failure occurrence to highlight the failure object in the VI. If the VI changed since you performed the analysis, the wrong object may be highlighted.

Results List	Subitems
increment.vi	5
! Unused Code	1
Occurrence 1	
VI Saved Version	1
Wires Under Objects	1
i Dialog Controls	2
i Enabled Debugging	1
reset array.vi	4
! Hidden Objects in Structures	2
VI Saved Version	1
i Dialog Controls	4
i Enabled Debugging	1
scale array.vi	5
Array Default Values	1
VI Saved Version	1
VI Size	1
i Dialog Controls	3
i Enabled Debugging	1

[✓] Show Failures Only
[● Sort by VI] ○ Sort by Test
! - High Rank Test
i - Low Rank Test

Description

The function "Increment" has none of its outputs wired, and therefore performs no useful operation. Consider removing this function from your diagram.

Figure 11.5. LabVIEW VI Analyzer

Deployment

The final step in a control application is deploying the application to an embedded system and setting it to start automatically. To set a LabVIEW VI to run automatically on startup, create an exe for the real-time target. You do this in the LabVIEW Project by creating a build specification. This includes all the information on the executable as well as any supporting files you want to export. The build specification makes it easy to recreate or customize a build. LabVIEW also has tools for system replication that you can use to copy the embedded code on a controller and replicate the code on another system.