

NI LabVIEW for CompactRIO Developer's Guide

Recommended LabVIEW Architectures and Development Practices
for Control and Monitoring Applications

This document provides an overview of recommended architectures and development practices when designing control and monitoring applications using LabVIEW Real-Time and LabVIEW FPGA software with NI CompactRIO controllers.

Contents

Overview and Background	1
CompactRIO Architecture.....	1
LabVIEW	1
Real-Time Controller.....	1
Reconfigurable I/O FPGA	3
I/O Modules.....	3
System Configurations	4
CHAPTER 1	
Designing a CompactRIO Software Architecture	6
CompactRIO Reference Architectures	6
Processes.....	7
Data Communication	7
Typical CompactRIO Architecture	9
Common Variant Architectures.....	9
Example—Turbine Testing System	11
CHAPTER 2	
Choosing a CompactRIO Programming Mode	14
When to Use LabVIEW FPGA	15
Using LabVIEW FPGA Interface Mode	17
When to Use CompactRIO Scan Mode	17
Using CompactRIO Scan Mode	18
When to Use Hybrid Mode.....	21
CHAPTER 3	
Designing a LabVIEW Real-Time Application	23
Designing a Top-Level RTOS VI	23
Deterministic and Nondeterministic Processes	24
Implementing Deterministic Processes	25
Setting the Priority of a Deterministic Task	25

Creating Modular Code With SubVIs	28
Interprocess Data Communication	29
Single-Process Shared Variables	29
Current Value Table (CVT)	30
Queues	32
RT FIFOs	32
RT FIFO-Enabled Shared Variables	34
LabVIEW Real-Time Design Patterns	35
Synchronization and Timing	36
Design Patterns	37
Working With Constrained Resources	39
Working With Limited Disk Space	39
Working With Limited RAM	40
Working With Limited CPU Resources	46
Ensuring Reliability With Watchdog Timers	48
LabVIEW Real-Time Watchdog	49
LabVIEW FPGA Watchdog and Fail-Safes	50
CHAPTER 4	
Best Practices for Network Communication	52
Network Communication	52
Selecting a Networking Protocol	52
Communication Model	52
Network Configuration	53
Interfacing to OSs and Third-Party Applications	53
LabVIEW Version	54
Security	54
Ease of Use Versus Performance and Reliability	54
Network-Published Shared Variables	55
Network Variable Nodes	56
Shared Variable Engine	56
Publish-Subscribe Protocol (PSP)	57
Network-Published Shared Variable Features	57
Hosting and Monitoring Network-Published Shared Variables	59
Tips for Using Network Shared Variables Effectively	61
Network Streams	62
Using Network Streams to Send Messages and Commands	65
Installing Network Stream Support to CompactRIO	66

Raw Ethernet (TCP/UDP).....	67
Simple TCP/IP Messaging (STM).....	68
Dealing With Orphaned Sockets When Using TCP/IP.....	71
CVT Client Communication (CCC).....	73
Instructions for Installing the CCC Library.....	73
Implementation	73
API and Example.....	74
Web Services	75
Security Options With Web Services.....	76
Adding Communication Mechanisms to Your Design Diagram	77
CHAPTER 5	
Customizing Hardware Through LabVIEW FPGA.....	78
FPGA Technology	78
Establishing a Design Flow.....	79
Best Practices for Implementing LabVIEW FPGA Code	80
Reading and Writing I/O	80
Synchronize Your Loops.....	82
Creating Modular, Reusable SubVIs.....	84
Avoiding Arbitration.....	86
Creating Counters and Timers.....	86
Data Communication	87
Interprocess Data Communication.....	87
Intertarget Communication	89
Test and Debug LabVIEW FPGA Code	98
Execute LabVIEW FPGA Code on Your Windows PC	99
Execute in Simulation Mode	99
CHAPTER 6	
Timing and Synchronization of I/O.....	111
LabVIEW FPGA Communication Nodes.....	111
I/O Nodes.....	112
Method Nodes	112
Property Nodes	112

Module Classifications	114
Direct FPGA Communication	114
SPI Bus Communication	115
Synchronizing Modules	122
Synchronizing Delta-Sigma Modules.....	122
Synchronizing Simultaneous On-Demand Modules	123
Synchronizing Multiplexed Modules	123
Synchronizing Delta-Sigma and Scanned (SAR) Modules.....	123
CHAPTER 7	
Adding I/O to the CompactRIO System.....	125
Adding I/O to CompactRIO	125
MXI-Express RIO	126
Ethernet RIO	127
EtherCAT RIO.....	128
Tutorial: Accessing I/O Using the EtherCAT RIO Chassis	129
CHAPTER 8	
Communicating With Third-Party Devices.....	135
Serial Communication From CompactRIO.....	135
Serial Communications From LabVIEW.....	137
Communicating With PLCs and Other Industrial Networked Devices	142
Industrial Communications Protocols	143
OPC.....	147
CHAPTER 9	
Designing a Touch Panel HMI	153
Building User Interfaces and HMIs Using LabVIEW	153
Basic HMI Architecture Background.....	153

CHAPTER 10	
Adding Vision and Motion	158
Machine Vision/Inspection	158
Machine Vision Using LabVIEW Real-Time	161
Machine Vision Using Vision Builder AI	165
Motion Control	168
Components of a Motion System	170
NI SoftMotion Architecture	173
NI SoftMotion Project Items and APIs	174
Real-Time-Based NI SoftMotion Components: The NI SoftMotion Engine	186
FPGA-Based NI SoftMotion Components	186
Hardware Interfaces to NI SoftMotion	194
NI SoftMotion and EtherCAT AKD Drives	194
NI SoftMotion and Drive Interface Modules (NI 951x C Series)	197
NI SoftMotion and Drive Modules (NI 950x)	199
Example Configurations	202
Ethernet RIO With C Series Motion Modules	202
EtherCAT-Based System	203
Mixed-Axis Systems	204
NI Drives and Motors	205
Determining System Requirements and Components	205
Stage Selection	206
Backlash	206
Motor Selection	206
CHAPTER 11	
Deploying and Replicating Systems	208
Application Deployment	208
Deploying Applications to CompactRIO	208
Deploying LabVIEW FPGA Applications	213
Method 2: Storing the Application in Nonvolatile Flash Memory on the FPGA Target	214
Deploying Applications That Use Network-Published Shared Variables	216
Recommended Software Stacks for CompactRIO	219
Imaging	220
Application Components	221
The Replication and Deployment Utility (RAD)	221

Deploying CompactRIO Application Updates Using a USB Memory Stick.....	225
APIs for Developing Custom Imaging Utilities	225
IP Protection.....	227
Locking Algorithms or Code to Prevent Copying or Modification	227
License Key.....	230
Deploying Applications to a Touch Panel.....	234
Porting to Other Platforms	240
NI Single-Board RIO.....	240
CHAPTER 12	
Security on NI RIO Systems.....	248
Security Concerns on RIO Systems	248
The Nature of Security on RIO Systems	248
Layers of Security	249
Security Vectors on RIO Systems.....	249
Best Practices for Security on RIO Systems.....	250
CHAPTER 13	
Using the LabVIEW for CompactRIO Sample Projects.....	251
Overview of Sample Projects.....	251
Walk-Through of the LabVIEW FPGA Control on CompactRIO Sample Project	256
Create a New Sample Project.....	256
FPGA Main VI	260
Control Loop.....	260
Watchdog Loop	261
RT Main VI.....	262
Watchdog Loop	266
System Health and FPGA Monitoring Loop	268
UI Main VI.....	270
Event Handling Loop	270
UI Message Loop.....	272
Monitoring Loop.....	273
Downloading the FPGA Bitfile to Flash Memory.....	275

Overview and Background

This guide provides best practices for designing NI CompactRIO control and monitoring applications using NI LabVIEW system design software along with the LabVIEW Real-Time and LabVIEW FPGA modules. Controllers based on these modules are used in applications ranging from the control of particle accelerators like the CERN Large Hadron Collider, 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 guide provides tips for developing CompactRIO applications that are readable, maintainable, scalable, and optimized for performance and reliability.

CompactRIO Architecture

CompactRIO is a rugged, reconfigurable embedded system containing three components: a processor running a real-time operating system (RTOS), a reconfigurable FPGA, and interchangeable industrial I/O modules. The real-time processor offers reliable, predictable behavior and excels at floating-point math and analysis, while the FPGA excels at smaller tasks that require high-speed logic and precise timing. Often CompactRIO applications incorporate a human machine interface (HMI), which provides the operator with a graphical user interface (GUI) for monitoring the system's state and setting operating parameters.

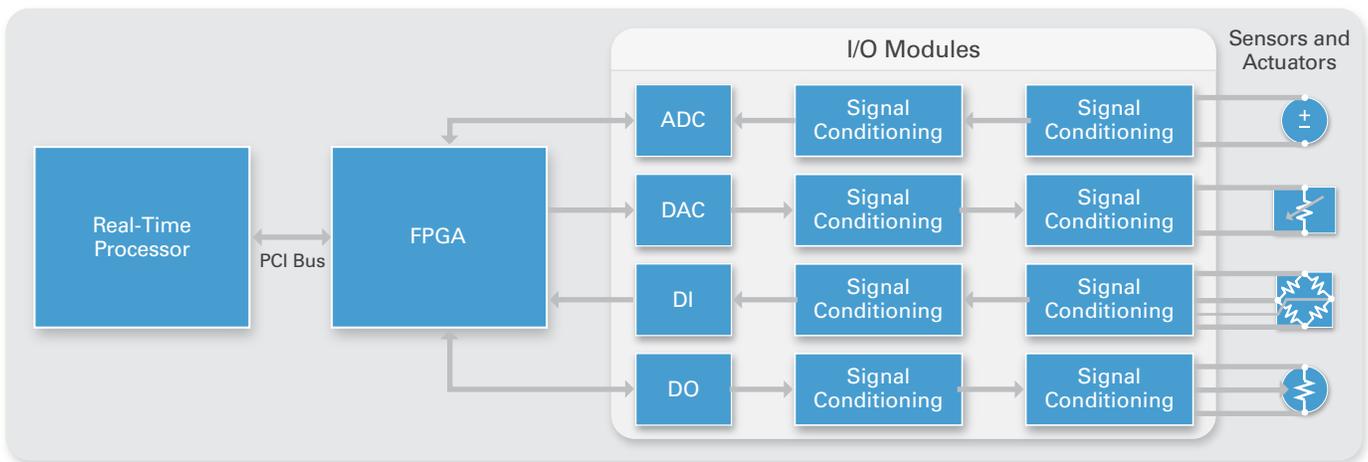


Figure 1. Reconfigurable Embedded System Architecture

LabVIEW

LabVIEW is a graphical programming environment used by thousands of engineers and scientists to develop sophisticated control systems using graphical icons and wires that resemble a flowchart. It offers integration with thousands of hardware devices and provides hundreds of built-in libraries for advanced control, analysis, and data visualization—all for creating user-defined systems more quickly. The LabVIEW platform is scalable across multiple targets and OSs, and, in the case of CompactRIO, LabVIEW can be used to access and integrate all of the components of the LabVIEW reconfigurable I/O (RIO) architecture.

Real-Time Controller

The real-time controller contains a 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 VDC 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 support.



Figure 2. NI cRIO-9024 Real-Time Controller

Real-Time Operating System (RTOS)

An RTOS is able to reliably execute programs with specific timing requirements, which is important for many science and engineering projects. The key component needed to build a real-time system is a special RTOS; other hardware and software pieces that make up an entire real-time system are discussed in the next section.

Precise timing

For many engineers and scientists, running a measurement or control program on a standard PC with a general-purpose OS installed (such as Windows) is unacceptable. At any time, the OS might delay execution of a user program for many reasons: running a virus scan, updating graphics, performing system background tasks, and so on. For programs that need to run at a certain rate without interruption (for example, a cruise control system), this delay can cause system failure.

Note that this behavior is by design—general-purpose OSs are optimized to run many processes and applications at the same time and provide other features like rich user interface graphics. In contrast, real-time OSs are designed to run a single program with precise timing. Specifically, real-time OSs help you implement the following:

- Perform tasks within a guaranteed worst-case timeframe
- Carefully prioritize different sections of your program
- Run loops with nearly the same timing on each iteration (typically within microseconds)
- Detect if a loop missed its timing goal

Reliability

In addition to offering precise timing, RTOSs can be set up to run reliably for days, months, or years without stopping. This is important not only for engineers building systems that need 24/7 operation but also for any application where downtime is costly. A “watchdog” feature is also typically included in real-time systems to automatically restart an entire computer if the user program stops running. Furthermore, hardware used in a real-time system is often designed to be rugged to sustain harsh conditions for long periods.

RTOSs on NI hardware

Each series of NI real-time targets runs on one of three different RTOSs. LabVIEW Real-Time provides a level of abstraction that helps programmers largely ignore the details of the underlying RTOS. However, to offer context to more advanced users, note that RTOSs on NI real-time targets include Phar Lap/ETS, VxWorks, and NI Linux Real-Time. Learn more about the new Linux-based RTOS at ni.com/linux.

Reconfigurable I/O FPGA

The reconfigurable I/O FPGA chassis is the center of the embedded system architecture. It is directly connected to the I/O for high-performance access to the I/O circuitry of each module and timing, triggering, and synchronization. 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 controller architectures. 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 further customize the system. Because of the FPGA speed, this chassis is frequently used to create controller systems that incorporate high-speed buffered I/O, 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 highest reliability and determinism, which is ideal for hardware-based interlocks, custom timing and triggering, or the elimination of custom circuitry normally required with nonstandard sensors and buses. For an overview of FPGA technology, see [Chapter 5: Customizing Hardware Through LabVIEW FPGA](#).



Figure 3. Reconfigurable FPGA Chassis

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 70 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, you can build your own modules or purchase modules from third-party 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 4. You can choose from more than 50 NI C Series I/O modules for CompactRIO to connect to almost any sensor or actuator.

System Configurations

The simplest embedded system consists of a single controller running in a “headless” configuration. This configuration is used in applications that do not need an HMI except for maintenance or diagnostic purposes. However, most control and monitoring applications require an HMI to display data to the operator or allow the operator to send commands to the embedded system. A common configuration is 1:1, or 1 host to 1 target, as shown in Figure 5. The HMI communicates to the CompactRIO hardware over Ethernet through either a direct connection, hub, or wireless router.

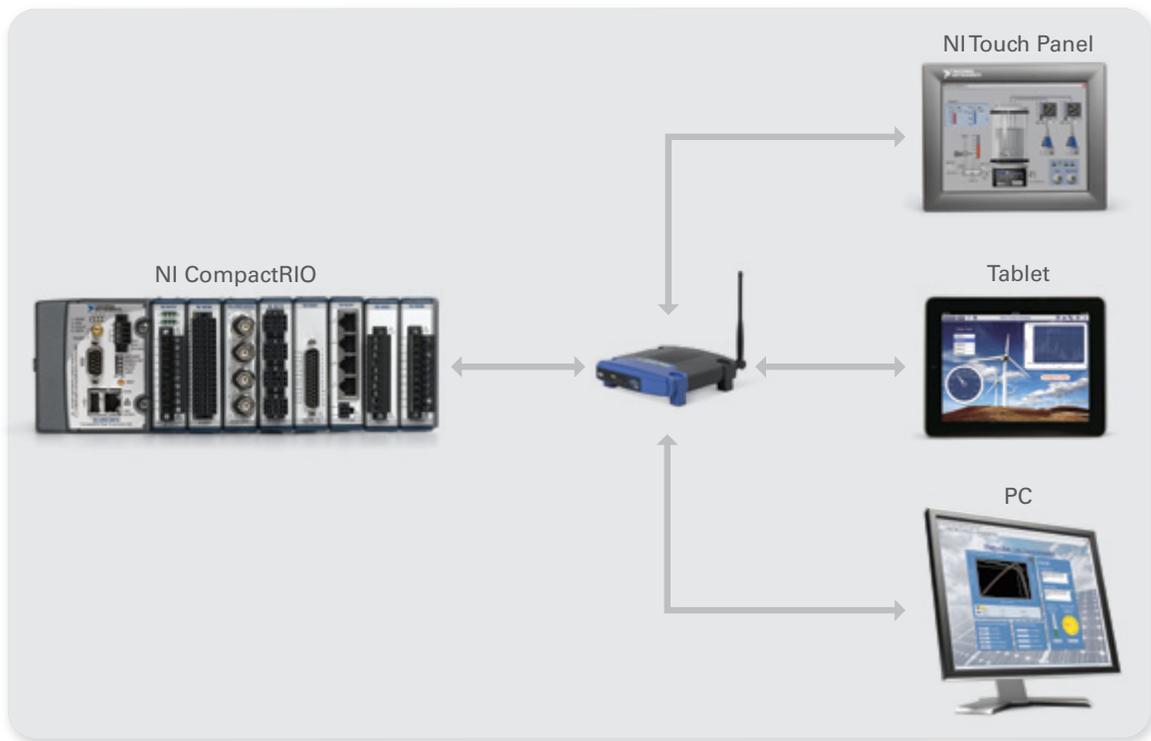


Figure 5. A 1:1 (1 Host to 1 Target) Configuration

The next level of system capability and complexity is either a 1:N (1 host to N targets) or N:1 (N hosts to 1 target) configuration. The host is typically either a desktop PC or an industrial touch panel device. The 1:N configuration, shown in Figure 6, is typical for systems controlled by a local operator. The N:1 configuration is common when multiple operators use an HMI to check on the system state from different locations.

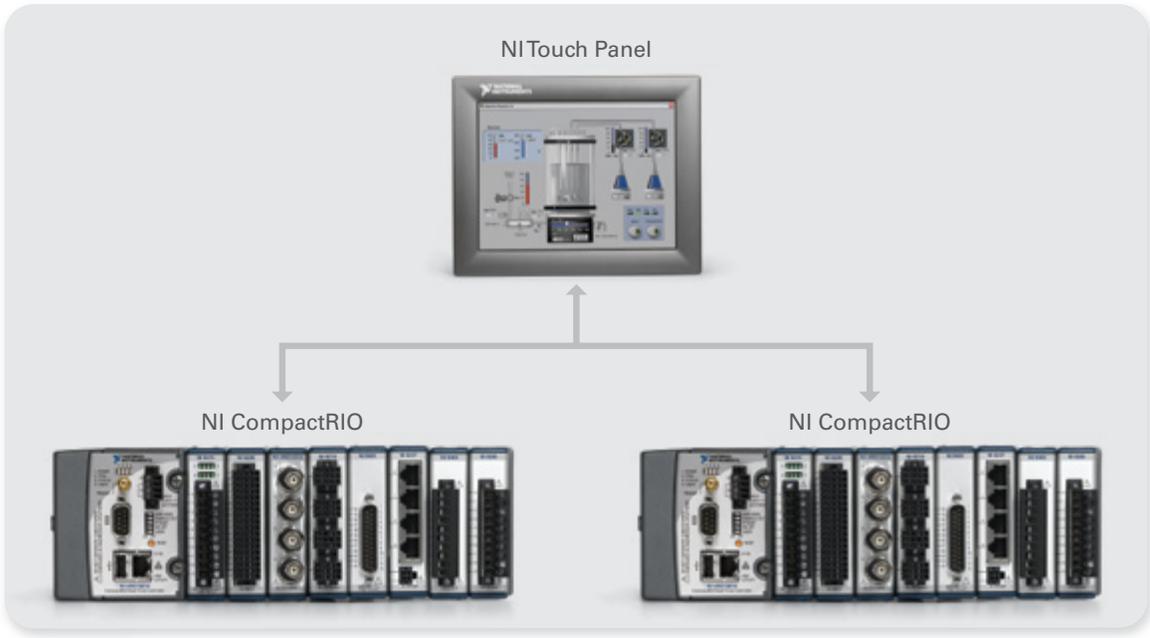


Figure 6. A 1:N (1 host to N targets) configuration is common for systems controlled by a local operator.

Complex machine control applications may have an N:N configuration with many controllers and HMIs (Figure 7). They often involve a high-end server that acts as a data-logging and forwarding engine. This system configuration works for physically large or complex machines. Using it, you can interact with the machine from various locations or distribute specific monitoring and control responsibilities among a group of operators.

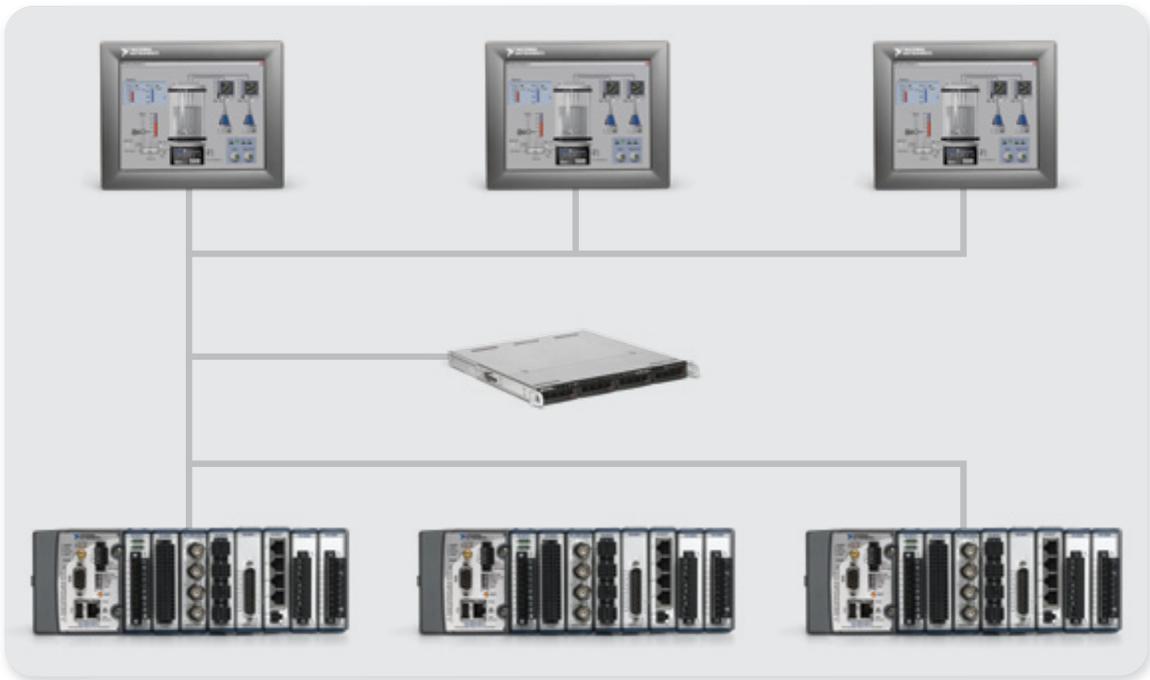


Figure 7. A Distributed Machine Control System

This guide walks through recommended implementations for a variety of FPGA, controller, and HMI architecture components. It also offers example code and, in some cases, alternative implementations and the trade-offs between implementations.

CHAPTER 1

Designing a CompactRIO Software Architecture

Almost all CompactRIO systems have a minimum of three top-level VIs executing asynchronously across three different targets: the FPGA, the real-time operating system (RTOS), and a host PC. If you begin your software development without having some sort of architecture or flowchart to refer to, you may find keeping track of all of the software components and communication paths to be challenging. Having a diagram that describes all of the system pieces at a high level helps guide development and communication about the design with stakeholders. This section describes several common CompactRIO architectures that you can use as a starting point for many applications. You also can use them as a guide for creating your own architecture.

The example architectures in the following sections were created using a vector-based drawing program called the yEd Graph Editor, which is a free design tool downloadable from yworks.com.

CompactRIO Reference Architectures

A CompactRIO system architecture or diagram should provide a basic overview of the system. A basic diagram includes processes and data communication paths. A process, as discussed in this guide, is an independent, asynchronously executing segment of code—basically a loop. The architectures referenced in this guide include processes represented by yellow boxes, the hardware targets that the processes execute on represented by blue boxes, and data communication paths represented by black arrows.

The optional RIO Scan Interface process is used when programming the CompactRIO device with Scan Mode or Hybrid Mode. The RIO Scan Interface is included in the LabVIEW Real-Time Engine so you can access your I/O channels as variables within the LabVIEW Real-Time development environment. You can find more information on when to use the Scan Mode in [Chapter 2: Choosing a CompactRIO Programming Mode](#).

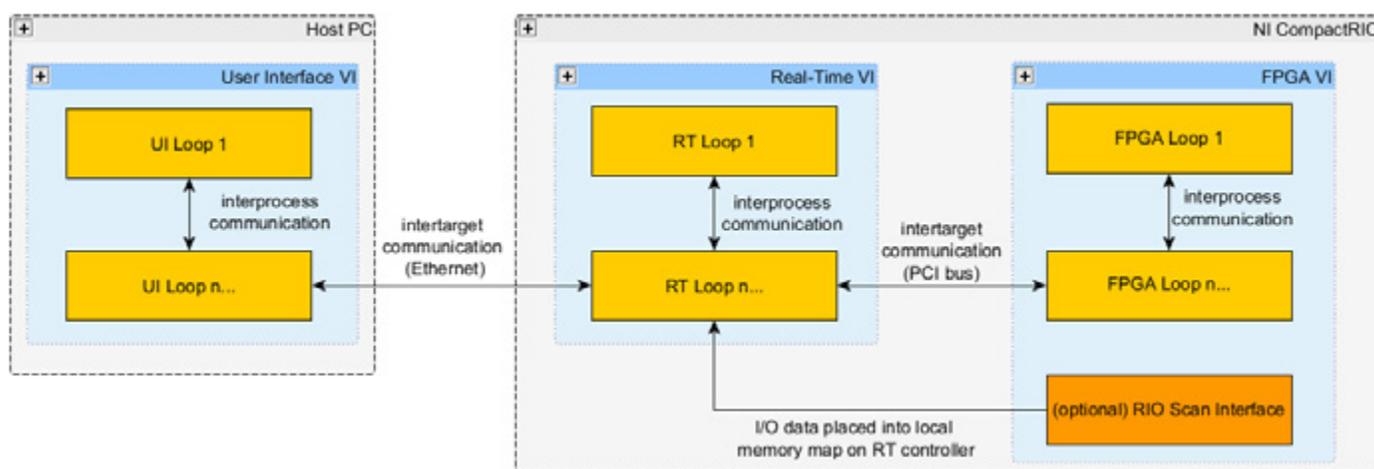


Figure 1.1. Basic Components of a CompactRIO Architecture

Processes

The first step in identifying the processes that your application will contain is to create a list of tasks that your application needs to accomplish. Common tasks include PID control, data logging, communication to an HMI, communication to I/O, and safety logic or fault handling. The next step is deciding how the tasks are divided into processes. An application with a large number of processes requires more time to be spent on interprocess data communication. At the same time, breaking your tasks into individual processes makes your program more scalable. For example, you might have a real-time application during the first phase of a project that is communicating across the network to a user interface (UI) built with LabVIEW, but later on the application needs to communicate to a web browser. If you separate the network communication task from other tasks within the application, you can redesign your network communication process without impacting your control process. More information on designing processes in LabVIEW Real-Time can be found in [Chapter 3: Designing a LabVIEW Real-Time Application](#).

Data Communication

Once you've diagrammed your processes, the next step is to identify data communication paths. Data communication is one of the most important factors to consider when designing an embedded system. LabVIEW, LabVIEW Real-Time, and LabVIEW FPGA include many different mechanisms for transferring data between processes on a single target as well as processes that communicate across targets.

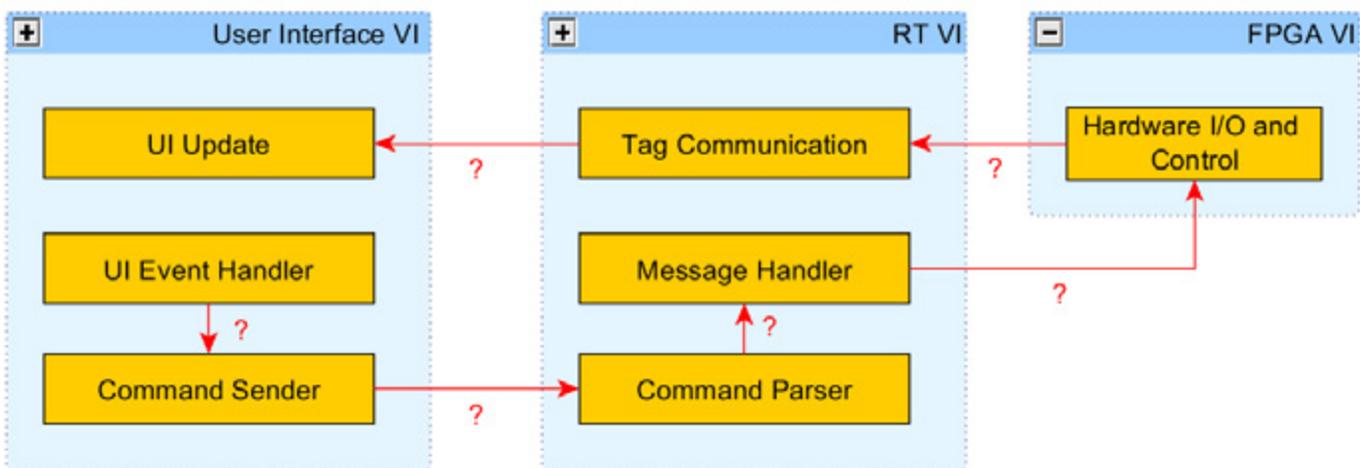


Figure 1.2. Data communication is one of the most important factors when designing an embedded system.

Choosing a data communication mechanism depends on several factors, with the primary factor being the data communication model. Most CompactRIO applications can contain at least two or more of the data communication models listed below:

- Message/Command
- Current Value (tag)
- Stream

Each model has different data transfer requirements. These models apply to both interprocess communication and intertarget communication. Later chapters refer to these data communication types when recommending LabVIEW mechanisms for data transfer between processes on a Windows target, real-time target, or FPGA target in addition to communication between the CompactRIO device and a host PC.

Message/Command

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. In message-based communication, it is important to guarantee delivery of the message in a timely manner or with low latency. In the previous example, when the operator pushes the stop button, the operator expects an immediate response (a human's perception of "immediate" is on the order of tenths of a second). It is often desirable to synchronize a process to the incoming message or command to conserve processor resources. This involves blocking until a message is received or a timeout occurs.

Current Value (tag)

Tags are used to communicate current values periodically between processes, or from a controller to an HMI. Tags are commonly used to represent process variables, such as a setpoint for a control system or current temperature data. Often tags are bound to I/O channels within a high-channel-count system such as a machine controller. With current value data or tags, each data transmission does not need to be guaranteed because the controller or HMI is always interested in the latest value, not in historical values. It is often desirable to have features that you can use to look up tags dynamically and manage different groups since tags are usually associated with high-channel counts.

Stream

Streaming communication is buffered and involves the high-throughput transfer of every data point, where throughput is more important than latency. Typically, one process writes data to another process that reads, displays, or processes that data. An example is an in-vehicle data logger that sends data from the FPGA to the RTOS for data logging. Streaming often requires multilayer buffering, during which each end (the sender and the receiver) has its own uniquely sized buffer. This is ideal when you have a processor loop that wants to read large chunks of data at a time and a DAQ loop that writes smaller chunks of data at shorter intervals.

A summary of the different types of data communication models and their features is shown in Table 1.1. Keep these features in mind when selecting a data communication model.

Communication Type	Fundamental Features	Optional Features	Performance
Message/Command	Buffering, blocking (timeout)	Acknowledgement of data transfer	Low latency
Current Value (Tag)	Current value only, read periodically	Dynamic lookup, group management	Low latency, high-channel count
Stream	Buffering, blocking (timeout)	Multilayer buffering	High throughput

Table 1.1. Summary of Data Communication Models

Typical CompactRIO Architecture

Figure 1.3 shows a common architecture that you can use as a starting point for most control and monitoring applications. The host VI provides an event-based user interface so an operator can interact with the embedded system. The RTOS executes high-level control and the FPGA executes low-level control.

There are two network communication paths: one path for sending commands from the user interface to the CompactRIO hardware and a second path for sending tags (current value data) from the CompactRIO hardware to the user interface for display. To build a scalable application, you should design your system so that it has a single command parser task that you can use to interpret and redistribute the command as needed. This ensures 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.

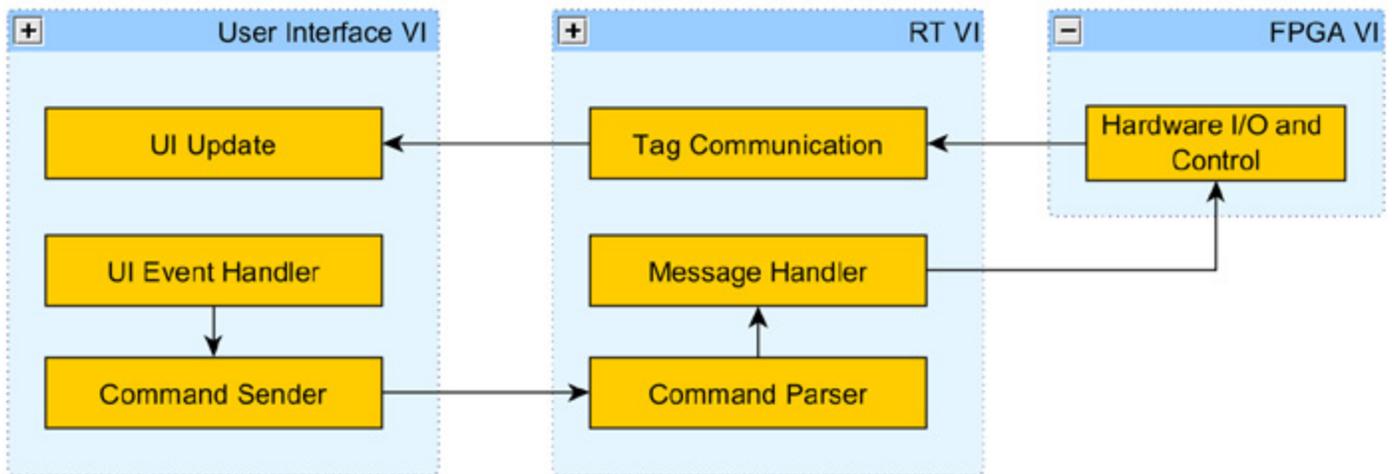


Figure 1.3. Common Software Architecture Control and Monitoring Applications

On the RTOS_Main.vi, the communication task and control task are designed as two separate loops. This is generally a good practice since the network communication could impact the determinism of the control algorithm if the tasks are executed within a single loop. Also, the modularity makes it easier to simulate parts of the system. For example, you could easily swap out the actual control loop with a simulated control loop to test the network communication.

Common Variant Architectures

The control and monitoring architecture is a great starting point for most applications, but if you want to design an embedded data logger, an embedded monitoring system, or a supervisory control and data acquisition (SCADA) system, you can leverage the more application-specific architectures in the following sections.

Headless Data Logger

A headless CompactRIO system does not require a user interface running on a remote machine. In the case of a headless data logger, the remote machine might be used for retrieving files that contain logged data from the FTP server. In this architecture, the FPGA is performing low-level control while acquiring data from NI C Series I/O modules at a high speed. This data is streamed from the FPGA to the RTOS, where it can be logged to the onboard memory. The FPGA communication process is separate from the logging engine processes to maximize the rate at which data is read.

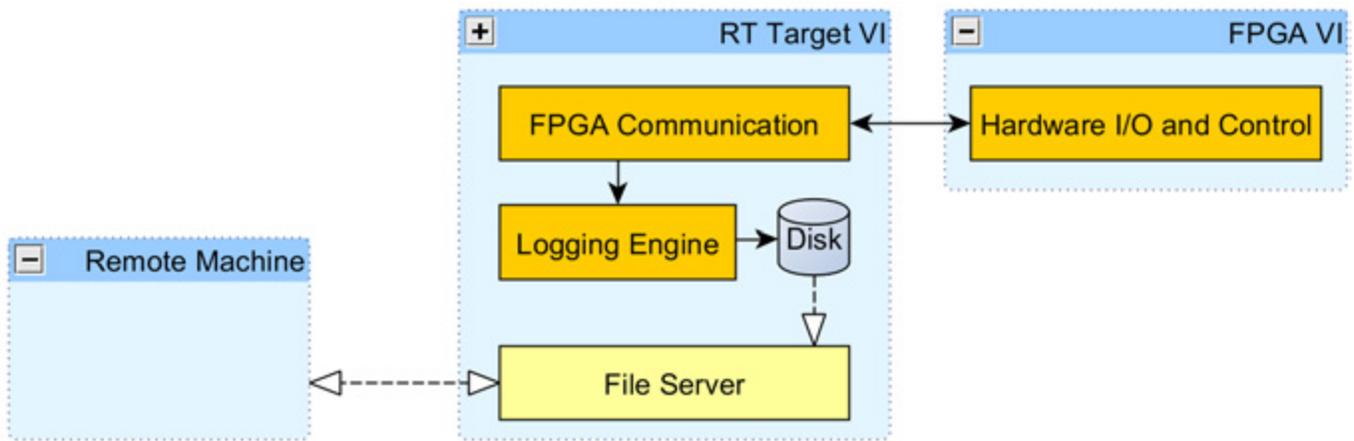


Figure 1.4. Headless Data Logger Architecture

Embedded Monitoring

Embedded monitoring applications typically require data to be streamed at high rates from the C Series I/O modules up to a host user interface where the data is displayed to an operator. The user interface also allows the operator to configure the hardware, which requires a mechanism for sending commands from the host down to the RTOS.

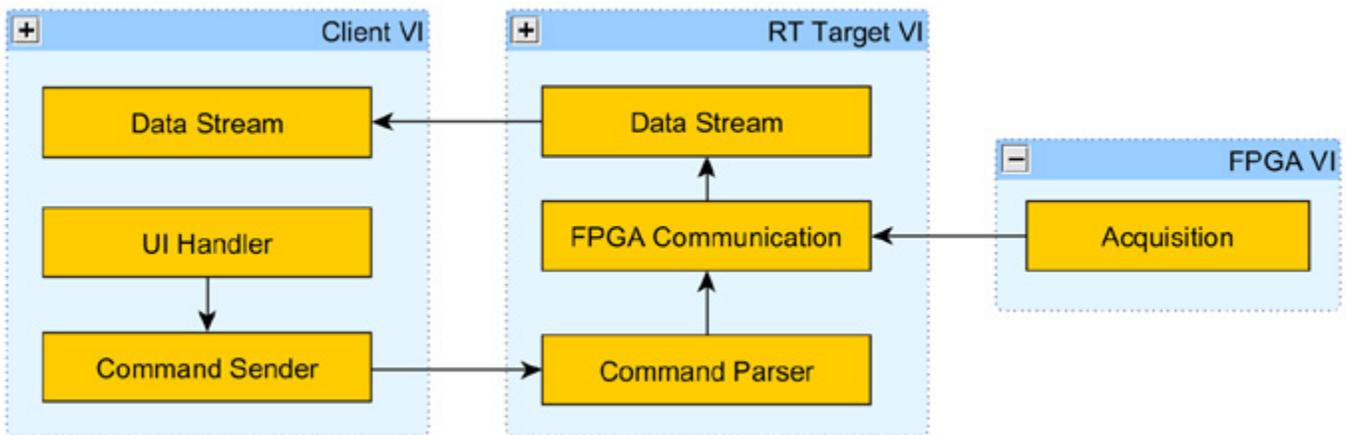


Figure 1.5. Embedded Monitoring Architecture

Supervisory Control and Data Acquisition (SCADA) Architecture

In a typical SCADA application, the CompactRIO device communicates I/O channels as tags to a supervisor. Some control might be implemented on either the FPGA or RTOS. Often you can use the RIO Scan Interface to handle I/O for these types of applications since they do not require high-speed DAQ rates or customized hardware. The NI Scan Engine is discussed in [Chapter 2: Choosing a CompactRIO Programming Mode](#). With the LabVIEW Datalogging and Supervisory Control (DSC) Module, you can extend the software capabilities of a SCADA system to include the ability to log data to a historical database, set alarms, and manage events. LabVIEW DSC also features support for commonly used industrial protocols including OLE for process control (OPC), so LabVIEW can communicate to third-party programmable logic controllers (PLCs).

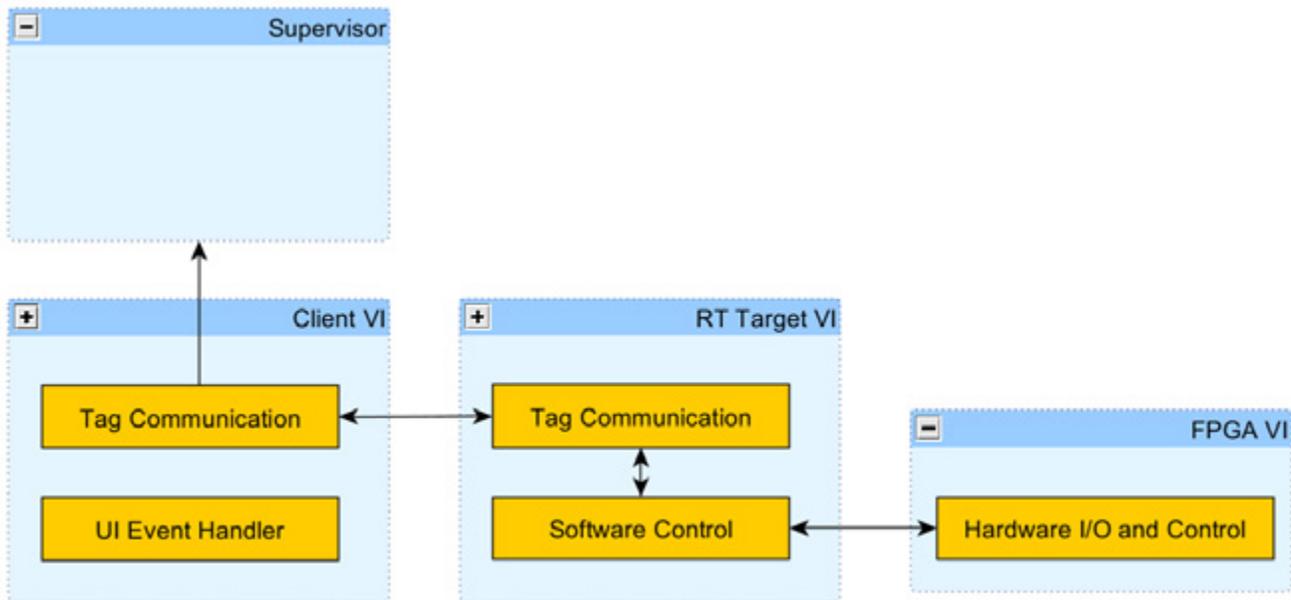


Figure 1.6. SCADA Architecture

Example—Turbine Testing System



LabVIEW example code is provided for this section.

An example of a turbine testing system demonstrates how the control and monitoring architecture in Figure 1.3 could apply to an actual application. A turbine is a rotary engine in which the kinetic energy of wind, water, or steam power is converted into mechanical energy by causing a bladed rotor to rotate. The blades turn a shaft that is connected to a generator, as shown in Figure 1.7. The generator creates electricity as it turns.

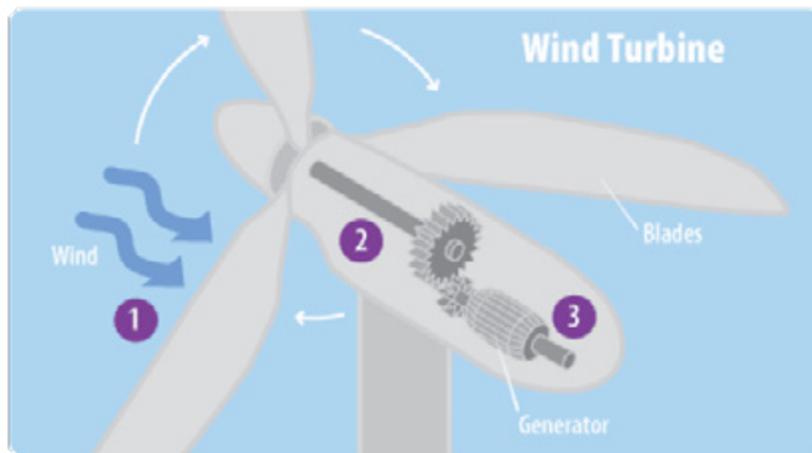


Figure 1.7. One common type of turbine is a wind turbine.

The turbine testing system used in this example has the following software requirements:

- Control the turbine speed using an analog PWM signal
- Monitor the revolutions per minute (rpm) of the turbine using a tachometer
- Trigger finite acquisitions that can be used for analysis

- Calculate and display the frequency of the accelerometer data
- Monitor temperature to prevent overheating of the turbine during testing
- Display the temperature, tachometer velocity (rpm), accelerometer data, and frequency of the accelerometer data to the UI

To create an architecture for the turbine testing system, start with the control and monitoring architecture shown in Figure 1.8.

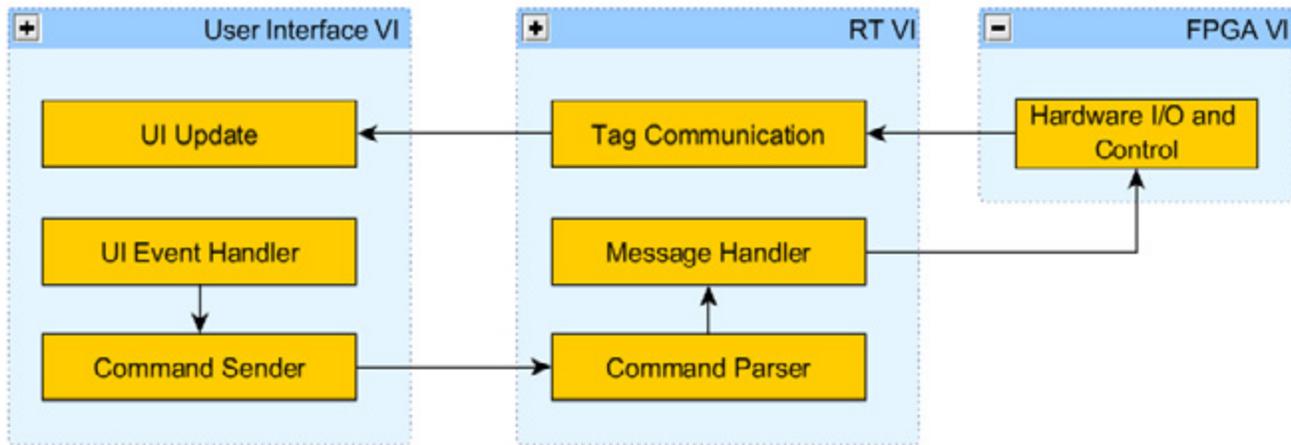


Figure 1.8. Common Software Architecture for Control and Monitoring Applications

This application was implemented in Hybrid Mode, and uses both the RIO Scan Interface and LabVIEW FPGA Module. LabVIEW FPGA is used for the following tasks:

- **Turbine Speed Control**—This loop generates a PWM signal to control the turbine speed based on the duty cycle (%) and period (mS) communicated from the LabVIEW Real-Time VI.
- **Accelerometer and Tachometer Acquisition**—This loop takes in the start trigger and acquires data from the tachometer and accelerometer until it reaches the specified number of samples. It also streams the sensor data to the LabVIEW Real-Time VI using a DMA first-in-first-out (FIFO) memory buffer.

The RIO Scan Interface is used to monitor the temperature at a rate of 10 Hz

The LabVIEW Real-Time VI has four tasks running in parallel. The data acquisition and analysis task uses a Timed Loop, and the other three tasks use While Loops. In LabVIEW Real-Time, Timed Loops are used to implement high-priority tasks, and While Loops are used to implement normal priority tasks. For more information on when to use a Timed Loop versus a While Loop, see [Chapter 3: Designing a LabVIEW Real-Time Application](#). The following is an overview of the LabVIEW Real-Time loops:

- **Data Acquisition and Analysis (High Priority)**—This loop communicates with the Accelerometer & Tachometer Acquisition Loop in the LabVIEW FPGA VI. It sends a trigger to begin the finite acquisition, retrieves accelerometer data from the DMA FIFO, and sends the collected and analyzed data to the user interface for display. It also retrieves tachometer data and converts it to revolutions per minute.
- **Turbine PWM Characteristics**—This loop monitors the temperature, and if it rises above the maximum limit, it shuts down the turbine.
- **Message Handler**—This loop receives any UI commands and distributes them to the appropriate real-time or FPGA process.

- **Command Parser**—This loop uses a Network Stream to receive commands from the user interface over Ethernet. The commands are placed into a Queue and sent to the Message Handler loop for distribution.

The User Interface VI runs on a Windows machine and allows the user to view data and interact with the CompactRIO system. It has the following tasks:

- **UI Event Handler**—This process handles user events with Event Structures such as “Stop” and “Trigger Acquisition.”
- **Command Sender**—This process sends any commands received from the UI Event Handler to the CompactRIO controller using a Network Stream.
- **UI Update**—This process receives the latest temperature value, accelerometer value, accelerometer frequency, and tachometer velocity and displays them to the user interface.

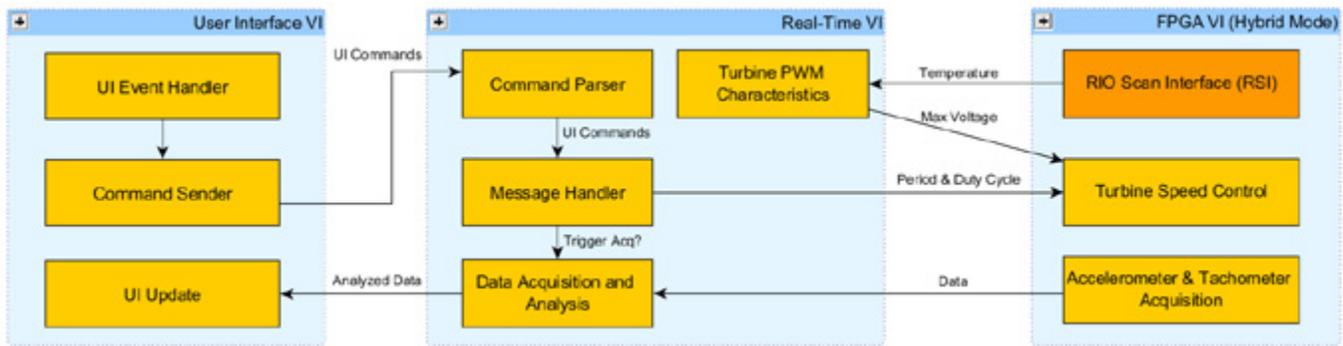


Figure 1.9. CompactRIO Architecture of a Turbine Testing Application

To view the actual turbine testing application, download the Turbine Testing Example from Section 1 Downloads. [Chapter 3: Designing a LabVIEW Real-Time Application](#) describes the real-time VI within the application in more detail.

In addition to the turbine testing example, LabVIEW 2012 and later includes several architectures similar to the ones discussed in this guide that you can use as starting points for your application. These architectures are called CompactRIO sample projects and they are listed below.

- LabVIEW FPGA Control Sample Project
- LabVIEW Real-Time Control Sample Project
- LabVIEW FPGA Waveform Acquisition and Data Logging Sample Project
- LabVIEW Real-Time Sequencer Sample Project (LabVIEW 2013 and later)
- LabVIEW Supervisory Control and Data Acquisition Sample Project (LabVIEW 2013 and later)

For more information on using these sample projects, see [Chapter 13: Using the LabVIEW for CompactRIO Sample Projects](#).

CHAPTER 2

Choosing a CompactRIO Programming Mode

The CompactRIO architectures presented in Chapter 1 provide you with the option to implement your I/O by customizing the FPGA hardware through LabVIEW FPGA or by using NI CompactRIO Scan Mode. If you have LabVIEW Real-Time and LabVIEW FPGA on your development computer, you are prompted to select which programming mode you would like to use when you add a CompactRIO target to your LabVIEW project. Upon selecting a mode, you can also go into Hybrid Mode if you want to use a hybrid of LabVIEW FPGA and CompactRIO Scan Mode for your application.

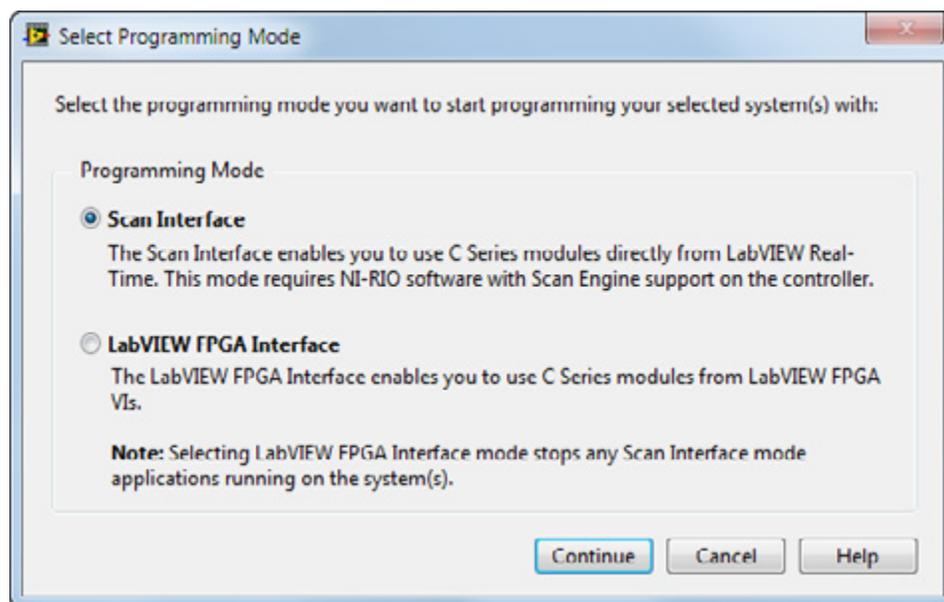


Figure 2.1. If you have LabVIEW FPGA installed, you are prompted to select a programming mode when adding a CompactRIO system to your LabVIEW project.

An overview of each programming mode follows.

LabVIEW FPGA Interface Mode

With the LabVIEW FPGA Interface mode, you can unlock the real power of CompactRIO by customizing the FPGA personality in addition to programming the real-time processor. This helps you achieve performance that typically requires custom hardware. Using LabVIEW FPGA, you can implement custom timing and triggering, offload signal processing and analysis, create custom protocols, and access I/O at its maximum rate.

When communicating data between an FPGA VI and Real-Time VI, you have the option to stream data at very high rates using DMA FIFOs, or to communicate single-point data using controls and indicators.

CompactRIO Scan Mode

Using CompactRIO Scan Mode, you can program the real-time processor of your CompactRIO system but not the FPGA. In this mode, NI provides a predefined personality for the FPGA that periodically scans the I/O and places it in a memory map, making it available to LabVIEW Real-Time. CompactRIO Scan Mode is sufficient for applications that require single-point access to I/O at rates of a few hundred hertz. It does not support streaming data at high rates.

Hybrid Mode

Using the CompactRIO Scan Mode and LabVIEW FPGA at the same time on a target is known as Hybrid Mode. With this approach, the modules you select to program directly with LabVIEW FPGA are removed from the I/O scan, and the remaining modules communicate with the RIO Scan Interface. Note that the RIO Scan Interface uses two of the three DMA channels that are normally available for use in LabVIEW FPGA.

This section offers tips on choosing a programming mode for your CompactRIO application. You should choose a programming mode based on your application requirements for performance, reliability, customization, and I/O. The NI LabVIEW for CompactRIO Developer's Guide discusses programming techniques and best practices for both LabVIEW FPGA programming and the RIO Scan Interface use.

When to Use LabVIEW FPGA

Like processor-based control systems, FPGAs have been used to implement all types of industrial control systems, including analog process control, discrete logic, and batch or state-machine-based control systems. However, FPGA-based control systems differ from processor-based systems in significant ways. If your application has any of the requirements listed below, you should program your I/O and other low-level tasks using LabVIEW FPGA. You can find more information on programming with LabVIEW FPGA in [Chapter 5: Customizing Hardware Through LabVIEW FPGA](#).

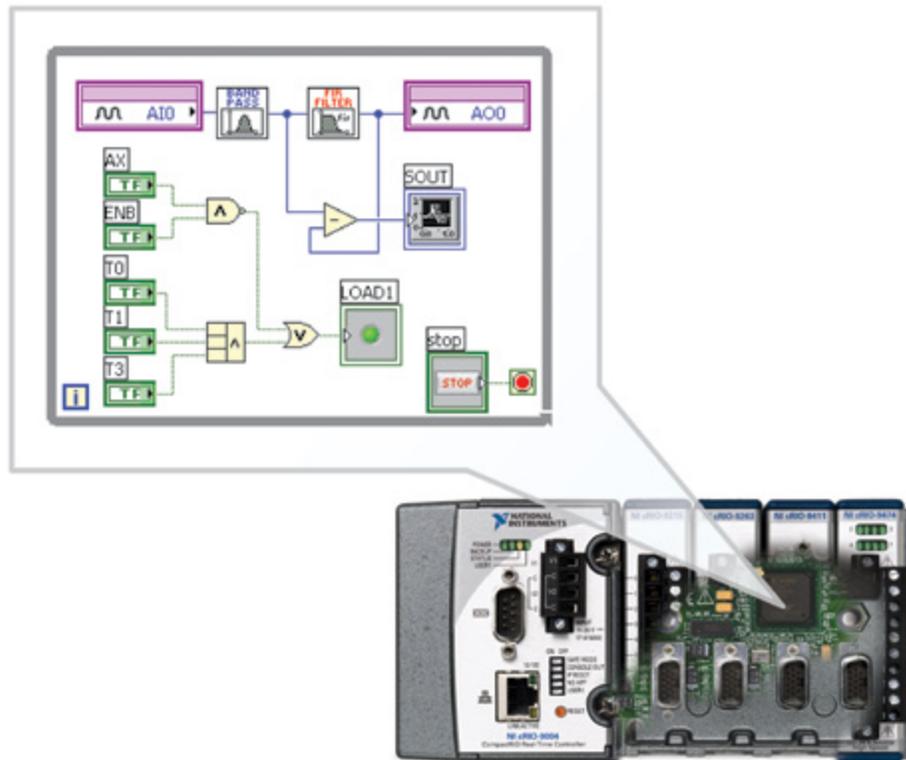


Figure 2.2. With LabVIEW FPGA, you can create custom FPGA VIs for high-speed data acquisition, control loops, or custom timing and triggering.

Maximum Performance and Reliability

When you compile your control application for an FPGA device, the result is a highly optimized silicon implementation that provides true parallel processing with the performance and reliability benefits of dedicated

hardware circuitry. Because there is no OS on the FPGA chip, the code is implemented in a way that ensures maximum performance and reliability.

High-Speed Waveform Acquisition/Generation (>500 Hz)

The RIO Scan Interface is optimized for control loops running at less than 500 Hz, 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 500 Hz, 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-variable 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 beyond 1 MHz. For example, the PID control algorithm that is included with the LabVIEW FPGA Module executes in just 300 ns (0.000000300 seconds).

Using LabVIEW FPGA Interface Mode

When you discover your CompactRIO controller from the LabVIEW project, select LabVIEW FPGA Interface as your programming mode. LabVIEW FPGA Interface mode automatically detects your I/O modules and adds them to the LabVIEW project. You can find more information and best practices on using LabVIEW FPGA Interface mode in [Chapter 5: Customizing Hardware Through LabVIEW FPGA](#).

When to Use CompactRIO Scan Mode

Some industrial control and monitoring applications are based on single-point I/O data. The data used in these processes represents the current value of a physical I/O channel. The processes are not concerned with tracking the time history of the data, comparing the current value to any of the previous values, or measuring the change rate of a value. Often they do not require loop rates faster than 500 Hz. You can simplify these types of applications by using CompactRIO Scan Mode.

RIO Scan Interface technology allows single-point I/O access up to rates of a few hundred hertz without the need to write FPGA code or an RT to FPGA 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 current value table on the CompactRIO controller.

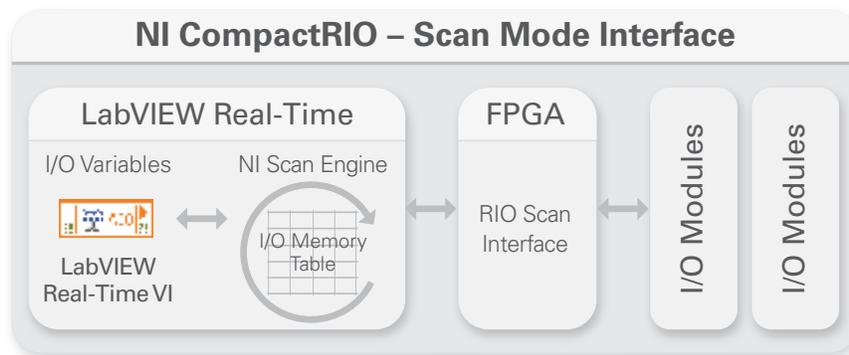


Figure 2.3. Block Diagram Description of the RIO Scan Interface Software Components

When deciding to use Scan Mode for a CompactRIO application, you should consider the required performance or loop rates and channel count. The graph in Figure 2.4 shows benchmarking that was completed using Scan Mode for a PID control loop, including one analog input and one analog output. The data shows that PID loop rates higher than 100 Hz combined with a high-channel count have a significant impact on CPU usage. Generally, you should not use Scan Mode when you require loop rates faster than 500 Hz.

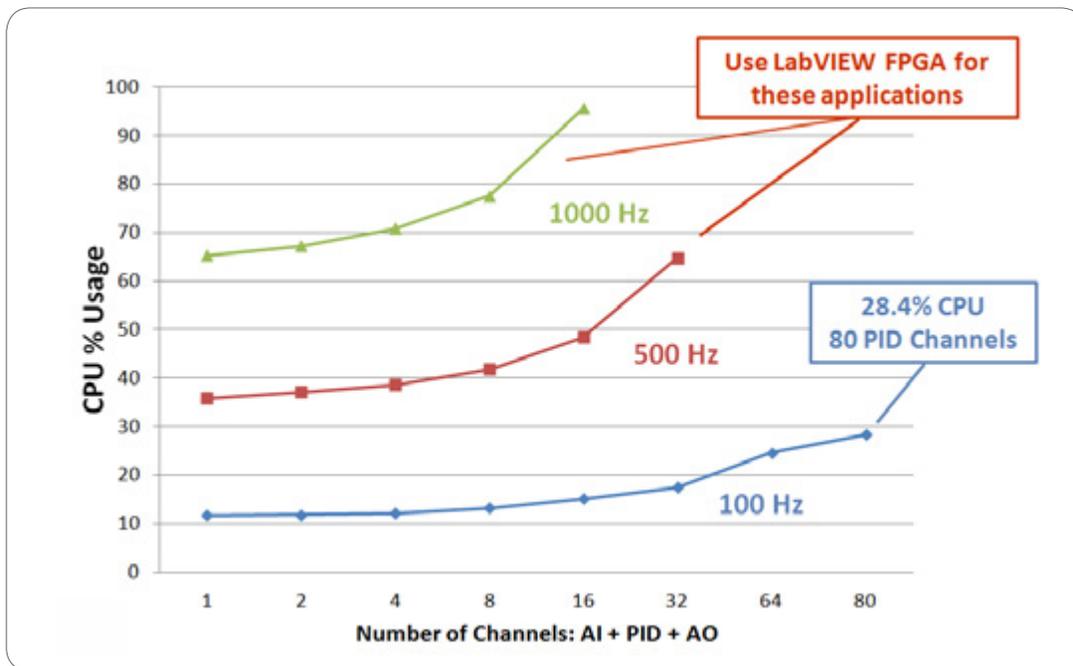


Figure 2.4. When using the RIO Scan Interface, high-channel counts combined with high-loop rates have a significant impact on CPU.

Although you should not use Scan Mode for high-speed data acquisition or control loops, it does offer several benefits:

- **Ease of Programming**—Drag and drop your I/O variables directly into your LabVIEW Real-Time VI during development.
- **Ability to Dynamically Detect I/O Modules**—Slots not configured through the project automatically detect inserted modules. You can then configure these modules through the variable API.
- **Fault Engine**—The NI Scan Engine features the built-in NI Fault Engine that throws errors deterministically.
- **Diagnostics and Debugging**—With the Distributed System Manager, you can view current values and faults as well as override current I/O values while your program is running.

Not all CompactRIO hardware works with NI Scan Mode. For a list of C Series I/O modules that feature Scan Mode support, see [C Series Modules Supported by CompactRIO Scan Mode](#). CompactRIO targets with 1M gate FPGAs cannot fully support the scan interface. You can implement some scan interface features on unsupported targets, but you must use LabVIEW FPGA.

Using CompactRIO Scan Mode

When you discover your CompactRIO controller from the LabVIEW project, select **Scan Interface** as your programming mode. The RIO Scan Interface automatically detects your I/O modules and adds them to the LabVIEW project. You can then drag and drop the I/O variables onto your LabVIEW Real-Time and host VI block diagrams and instantly read and write scaled, calibrated I/O data without any FPGA programming or compiling.

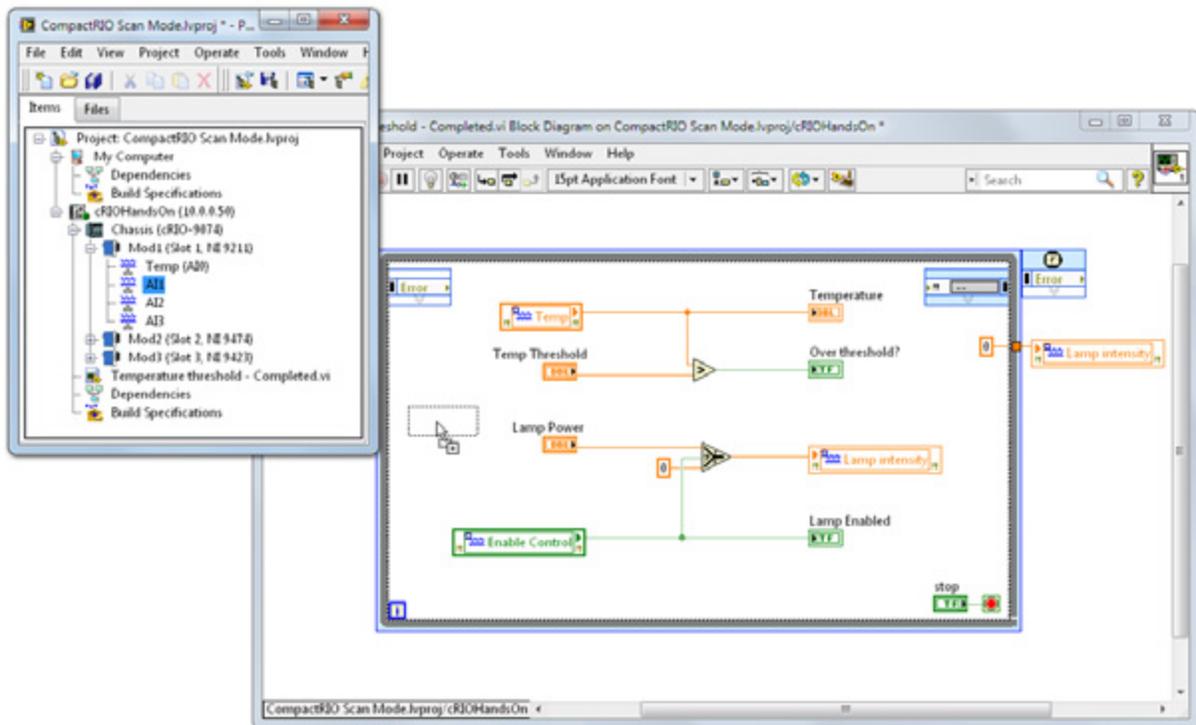


Figure 2.5. Drag and drop I/O variables onto your Real-Time VI block diagram.

The Scan Engine also provides a Timed-Loop timing source, so you can synchronize code with I/O updates for low-jitter control applications.

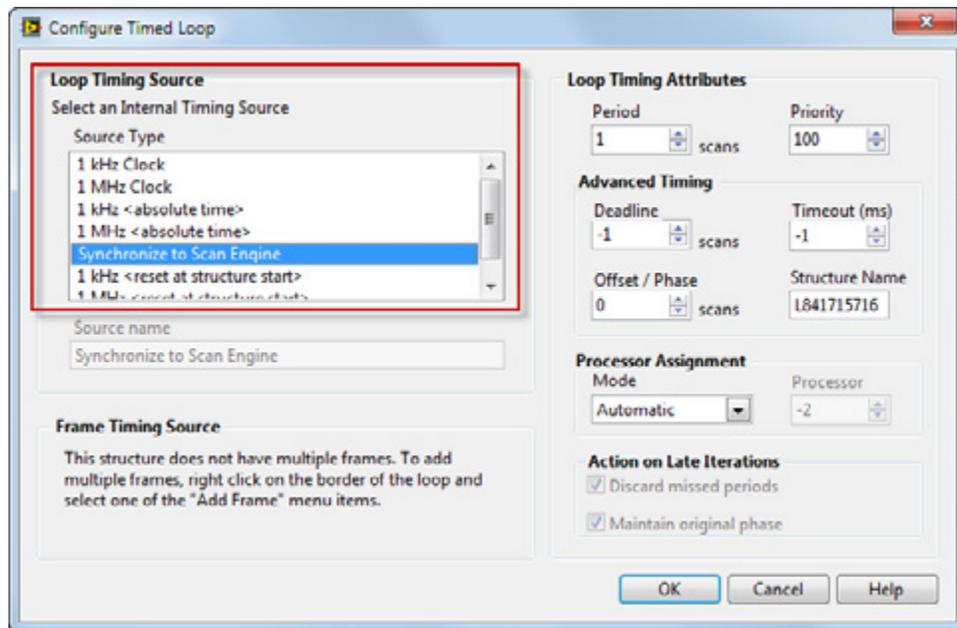


Figure 2.6. Use a Timed Loop with the clock source set to Synchronize to Scan Engine to synchronize the loop to I/O updates.

You can find example programs for using Scan Mode at **LabVIEW\examples\ScanEngine**.

When programming in Scan Mode, another option for interfacing to I/O variables is through the I/O Variable programmatic API, shown in the Figure 2.7. The programmatic API offers several benefits over the static I/O Variable Node API:

- You can iterate through multiple variables at once without dropping down a large number of I/O Variable Nodes
- Deployment to a CompactRIO target is less complicated since the items are not bound to a LabVIEW project
- The programmatic API promotes scalability
- You can change I/O variable configuration settings while running an application, and configuration settings are viewable on the block diagram

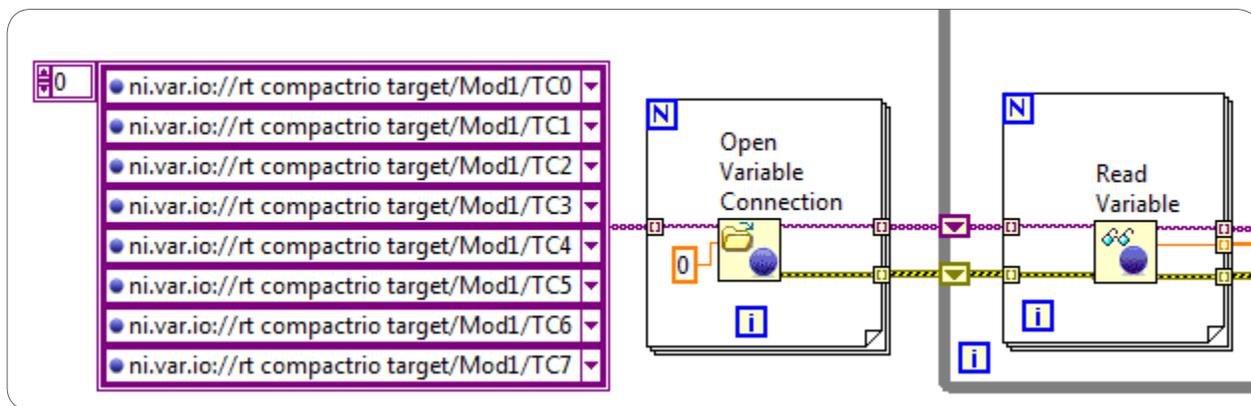


Figure 2.7. Use the I/O Variable Programmatic API

While the programmatic API offers several advantages, it does require more programming. The I/O Variable programmatic API can be found in the LabVIEW functions palette under **Data Communication»Shared Variable»I/O Variable**.

When deploying Scan Engine I/O variables to a CompactRIO controller, note that these variables have the potential to take up a significant portion of the CPU bandwidth. If you deploy a chassis in Scan Mode that physically contains all the modules you want to use, it publishes a variable called PercentHWScanUtilization. This value tells you what percentage of CPU bandwidth is dedicated to reading or writing I/O variables based on your scan time and deployed hardware. Based on this, you have an idea of how much time can be dedicated to other processes.

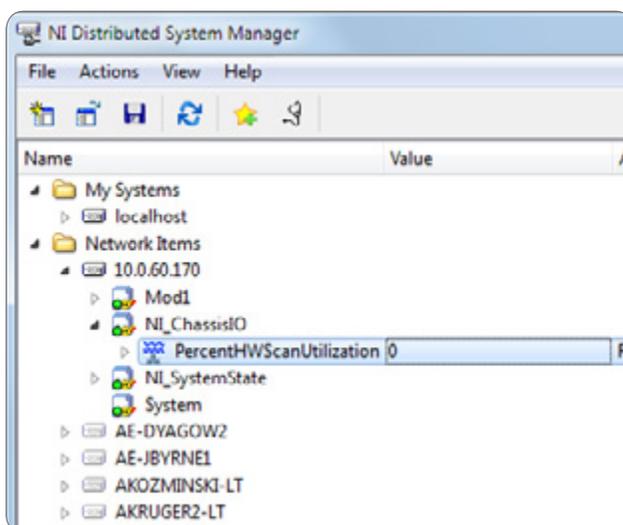


Figure 2.8. Monitor the CPU bandwidth that is used for reading and writing to Scan Engine I/O variables.

When to Use Hybrid Mode

Using Scan Mode and LabVIEW FPGA at the same time on a target is known as Hybrid Mode. With this approach, the modules you select to program directly with LabVIEW FPGA are removed from the I/O scan, and the remaining modules communicate with the RIO Scan Interface. When you compile your LabVIEW FPGA VI, if any I/O modules are configured to use Scan Mode, the necessary components of the RIO Scan Interface are included in the compile. The result is a single bitfile that works with the Scan Mode features for modules configured to use Scan Mode as well as your custom FPGA logic that communicates directly with the remaining I/O modules.

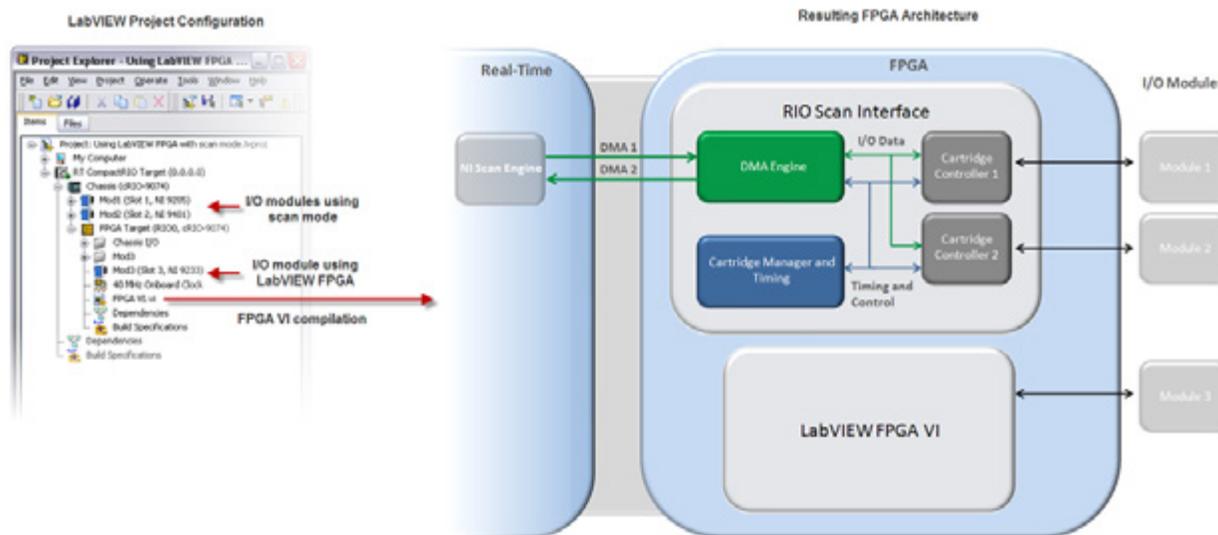


Figure 2.9. After activating Hybrid Mode, write an FPGA VI to interface with the module and pass data to the real-time host.

In Hybrid Mode, you can continue using the RIO Scan Interface on some modules while programming others directly on the FPGA. You can use the FPGA programming model to handle high-speed operations, inline processing, waveform buffered acquisition, and certain modules that do not feature Scan Mode support. 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 with the scan interface for other modules. You can access the FPGA I/O from the real-time VI by using either the FPGA Host Interface Functions or User-Defined Variables.

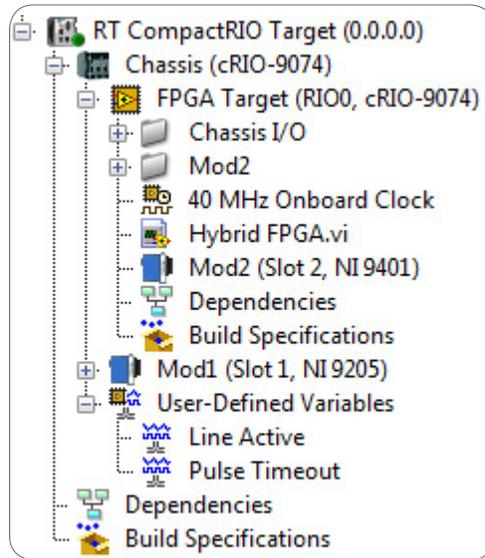


Figure 2.10. Activate FPGA programming for a particular module by dragging and dropping the module project item from under the CompactRIO chassis to under the FPGA target.

You need to note a few important limitations when using Hybrid Mode. First, the compile time significantly increases because the compiler has to combine the default RIO Scan Interface bitfile as well as the FPGA code that was created into one bitfile. Secondly, the number of DMA channels that you can use in the FPGA code is reduced since the Scan Engine uses two channels. Most FPGAs have three DMA channels, so most applications have only one DMA channel left to use in the FPGA code.

CHAPTER 3

Designing a LabVIEW Real-Time Application

When creating a LabVIEW Real-Time application for CompactRIO, start with a design diagram as described in [Chapter 1: Designing a CompactRIO Software Architecture](#). If you begin your software development without having some sort of architecture or flowchart to refer to, you may have difficulty keeping track of all software components and communication paths. Once you have created a design diagram, the next step is to implement your LabVIEW Real-Time code.

Designing a Top-Level RTOS VI

A good starting point for designing a LabVIEW Real-Time program is to create a top-level skeleton VI similar to the VI shown in Figure 3.1. In this skeleton VI, the processes (loops) are contained within subVIs. Containing your processes within subVIs helps you create a readable and maintainable application. Each subVI might contain one or more processes.

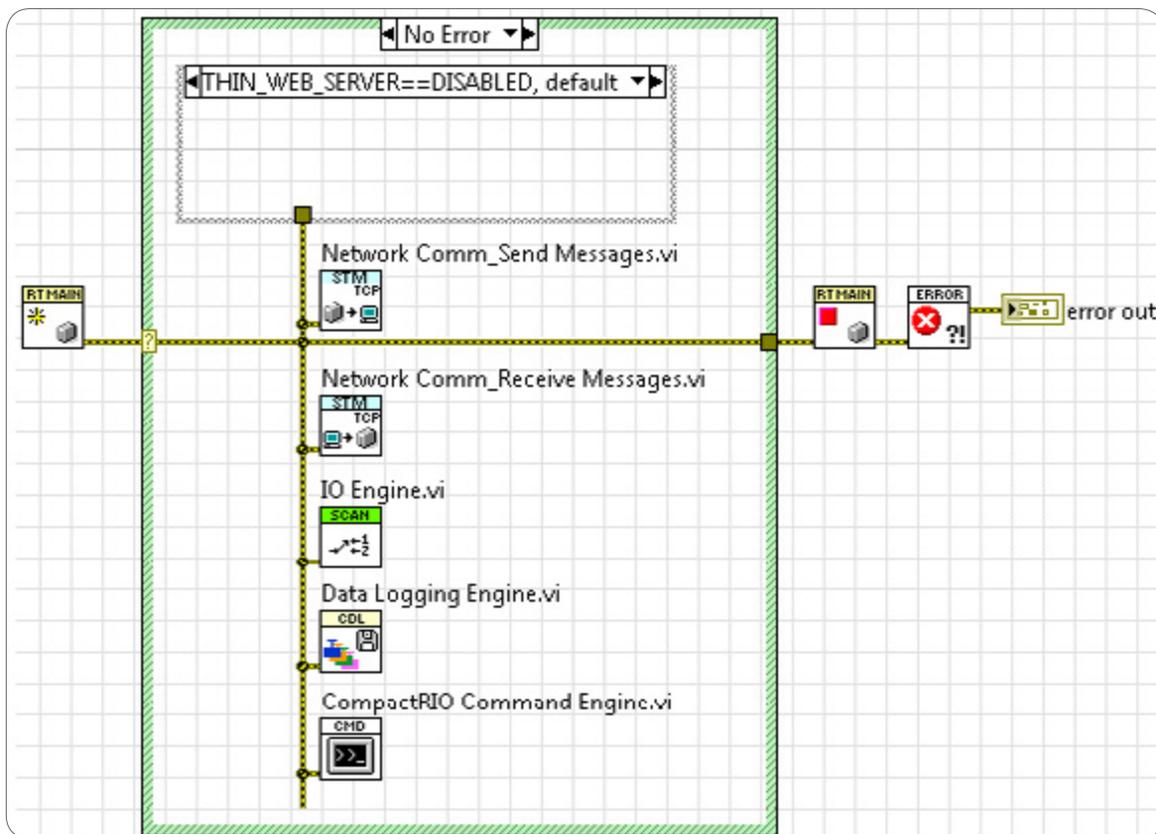


Figure 3.1. Creating a top-level VI with the general structure of the program filled out is a good starting point for developing a LabVIEW Real-Time application.

Once you have designed your top-level RTOS VI, the next step is to implement each of your processes. This chapter provides best practices for designing processes in LabVIEW Real-Time, including deterministic versus nondeterministic processes, interprocess data communication, design patterns, and memory management.

Deterministic and Nondeterministic Processes

For many engineers and scientists, running a measurement or control program on a standard PC with a general-purpose OS installed (such as Windows) is unacceptable. At any time, the operating system might delay execution of a user program for many reasons: to run a virus scan, update graphics, perform system background tasks, and more. In addition, applications that have high uptime requirements, meaning the application needs to run without a reboot for days, weeks, or years, often require an RTOS.

When designing a real-time application, you can choose from two levels of reliability. The first level of reliability guarantees that your program runs at a certain rate without interruption. The RTOS used on CompactRIO provides this level of reliability without requiring any special programming techniques. You can implement your processes as regular While Loops with timing functions to achieve much higher reliability than you would with a general-purpose OS.

The second level of reliability involves determinism and jitter. If determinism describes how a process responds to external events or performs operations within a given time limit, jitter is the measure of the extent to which execution timing fails to meet these expectations. Some applications require timing behavior to consistently execute within a small window of acceptable jitter, as shown in Figure 3.2.

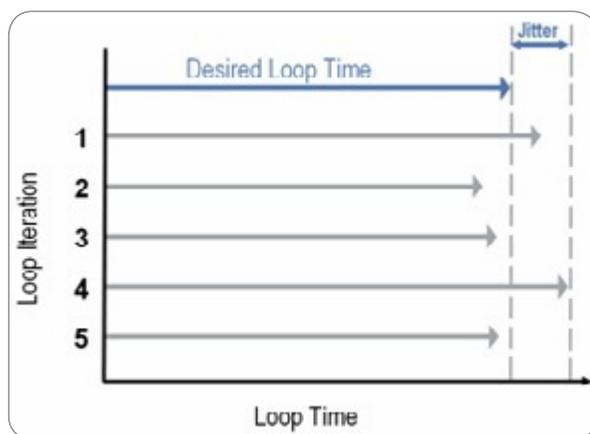


Figure 3.2. Jitter is the measure of how much the execution time of a task differs over subsequent iterations.

CompactRIO helps you bound the amount of jitter, which may be a requirement for mission-critical applications. You can choose from two options for designing a deterministic application with one or more processes that require a hard bound on the amount of jitter:

1. **Implement critical tasks using LabVIEW FPGA**—Since your LabVIEW FPGA code is running in hardware rather than within an OS, you can guarantee that a critical task does not crash or hang. An RTOS is more deterministic and reliable than a general-purpose OS, but it is still subject to software crashes if not programmed correctly.
2. **Implement critical tasks using Timed Loops in LabVIEW Real-Time**—Slightly more complex than a While Loop, a Timed Loop is designed to execute deterministic tasks with minimal jitter.

The next section describes implementing deterministic processes using Timed Loops and VI priorities. If your application does not contain any tasks that require bounded jitter, use While Loops with timing functions instead of Timed Loops. Timed Loops offer many useful built-in features but also involve more overhead and complexities than While Loops. You can benefit from the reliability of an RTOS without using Timed Loops, but you cannot place a bound on or minimize jitter.

Implementing Deterministic Processes

Chapter 1 discusses one method for designing your CompactRIO application—separating your tasks and identifying them as processes (loops). Your application might consist of both deterministic and nondeterministic tasks. Deterministic processes must complete on time, every time, and therefore need dedicated processor resources to ensure timely completion. Separate the deterministic tasks from all other tasks to ensure that they receive enough processor resources. For example, if a control application acquires data at regular intervals and stores the data on disk, you might need to handle the timing and control of the data acquisition deterministically. However, storing the data on disk is inherently a nondeterministic task because file I/O operations have unpredictable response times that depend on the hardware and the availability of the hardware resource. Therefore, you want to use a Timed Loop for the data acquisition task and a While Loop for data logging. While Loops should be used for nondeterministic tasks or unbounded code (file I/O, network communication, and so on).

Table 3.1 includes other examples of deterministic and nondeterministic tasks.

Deterministic Tasks	Nondeterministic Tasks
Closed-loop control	File I/O
Decision-making logic	Network or serial communication
Safety logic	Tasks involving large memory allocations
FPGA or RIO Scan Interface	Calls to nondeterministic libraries or drivers

Table 3.1. Examples of Deterministic and Nondeterministic Tasks

Setting the Priority of a Deterministic Task

If programmed correctly, an RTOS can guarantee that a program runs with consistent timing. RTOSs do this by providing programmers with a high degree of control over how tasks are prioritized and typically the ability to make sure that important deadlines are met. You can set the priority of a task within LabVIEW Real-Time by using a Timed Loop or setting the priority of a subVI. It's generally better to use Timed Loops when setting priorities if possible for readability purposes. With a Timed Loop, the priority is noted on the block diagram for any developer to see, whereas the priority of a subVI is only shown within the VI properties dialog.

Timed Loops

A Timed Loop is similar to a While Loop but includes additional features for executing code deterministically. With a Timed Loop, you can specify the period and priority at which your code executes. The higher the priority of a Timed Loop, the greater the priority the structure has relative to other timed structures on the block diagram. The Timed Loop on your block diagram with the highest priority is referred to as your time-critical loop. You can configure the priority and period dynamically on the block diagram or through a configuration window that you launch when you double-click the left input node on the Timed Loop.

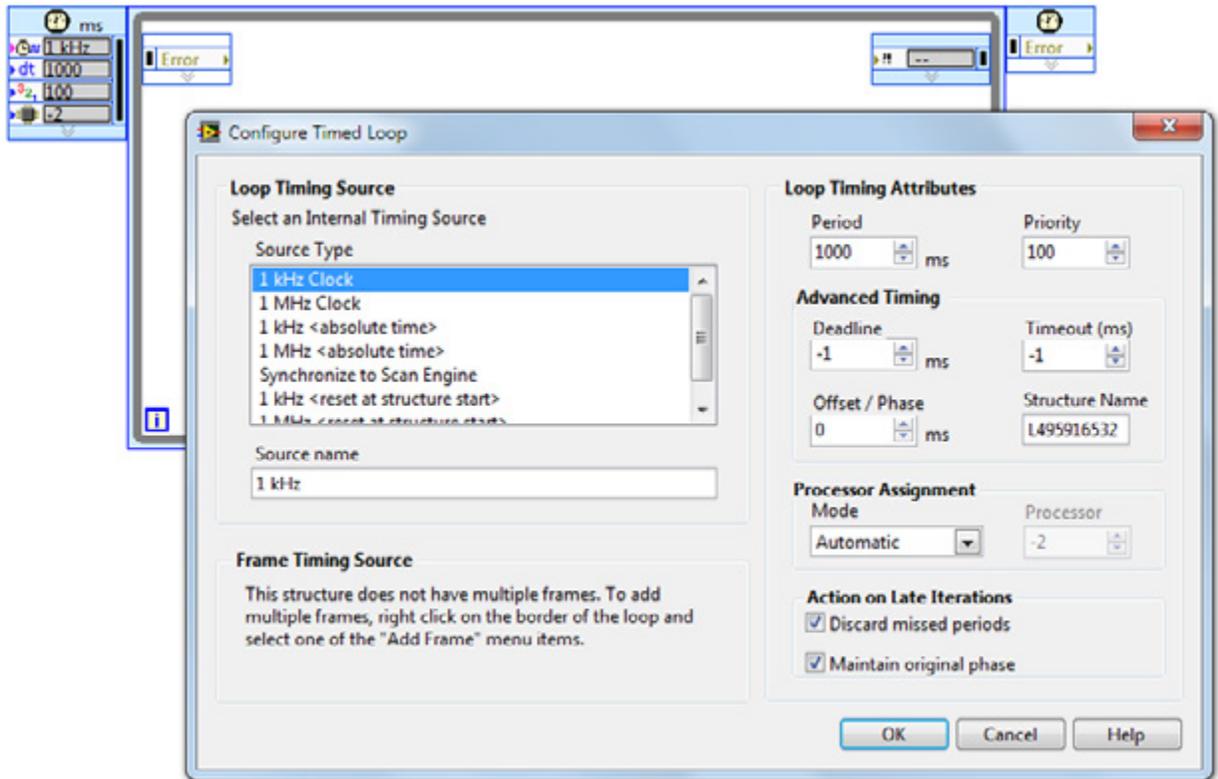


Figure 3.3. You can specify the period and priority of a Timed Loop through the Timed Loop configuration dialog.

When using Timed Loops on an RTOS, timing is important. When a time-critical loop is called, it is the only loop on the block diagram that executes. If a time-critical loop is set to run 100 percent of the time and never go to sleep, it completely monopolizes one core of your system. Be sure to specify a period that is reasonable for your time-critical loop but allows other background tasks within your application to execute.

One way to prevent undesired timing behavior is to create a time budget for your application. Time budgeting involves determining the amount of time required to execute each loop in the application and setting the rates of the loops accordingly. Your timing budget should restrict your application to using no more than 80 percent of the available CPU. Find instructions for creating a time budget in the LabVIEW Real-Time Help file [Avoid Jitter \(Real-Time Module\)](#).

VI Priorities

You can also set the priority of a deterministic section of code by placing it within a subVI and modifying the VI priority through the VI Properties dialog as shown in the Figure 3.4.

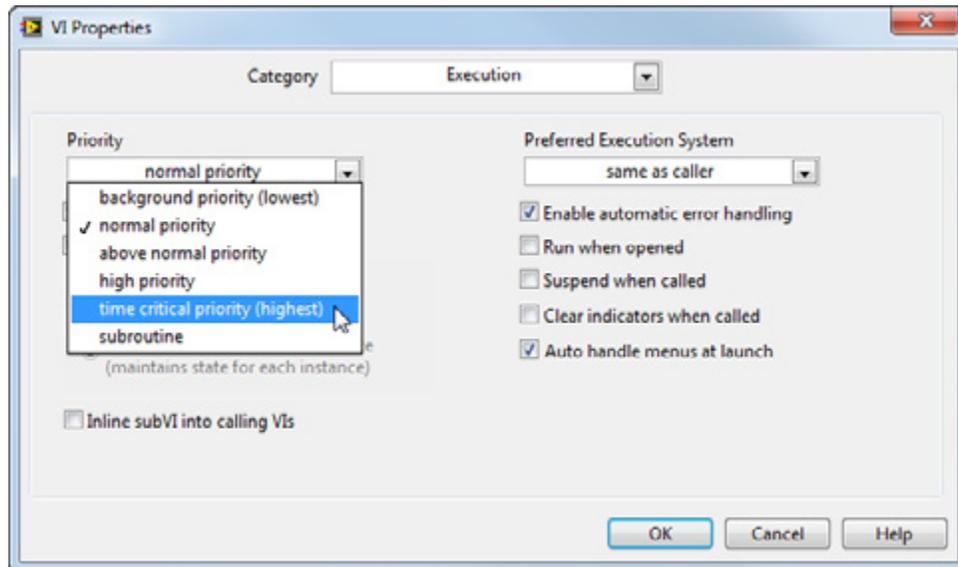


Figure 3.4. You can specify the priority of a VI through the VI Properties dialog under the "Execution" category.

Try to avoid using Timed Loops and VI priorities in the same application. If you must use both, you should have a solid understanding of how they relate. If you have a VI with a priority that contains a Timed Loop with a priority, the Timed Loop ignores the VI priority. In comparison to the VI priorities shown in Figure 3.4, Timed Loops execute at just below time-critical priority (highest) and at just above VIs with high priority. VIs with time-critical priority settings take precedence over Timed Loops. The While Loop executes at normal priority.

To understand how LabVIEW schedules parallel tasks based on priority, Figure 3.5 summarizes the priorities and execution systems available. An execution system is a pool of related threads. More information can be found in the LabVIEW Real-Time Help document Understand the Priority-Based Scheduling Model.

Priority	Above Time-Critical	NI Scan Engine Thread	Execution Systems						
	Time-Critical		1 Thread Per CPU						
	Timed Structure							1 Thread Per Timed Structure	
	High	Numerous Threads of Various Priorities	4 Threads Per CPU	1 Thread Per CPU					
	Above Normal		4 Threads Per CPU	1 Thread Per CPU					
	Normal		1 Thread Per CPU	4 Threads Per CPU	4 Threads Per CPU	4 Threads Per CPU	4 Threads Per CPU	1 Thread Per CPU	
	Background		4 Threads Per CPU	1 Thread Per CPU					
		System Threads	User Interface	Standard	Instrument I/O	Data Acquisition	Other 1	Other 2	Timed Structure Threads

Figure 3.5. Priorities and Execution Systems available in LabVIEW Real-Time

Creating Modular Code With SubVIs

Modularity, by definition, means to use modules or smaller parts for the overall objective. Within LabVIEW, program modularity means creating smaller sections of code known as subVIs. SubVIs are the same as VIs. They contain front panels and block diagrams, but you call them from within a VI. A subVI is similar to a subroutine in text-based programming languages. The turbine testing real-time VI uses the subVI shown in Figure 3.6 to calculate how many elements to read from the DMA FIFO at a time to avoid a buffer overflow or underflow condition. This subVI is called from the Data Acquisition and Analysis Loop.

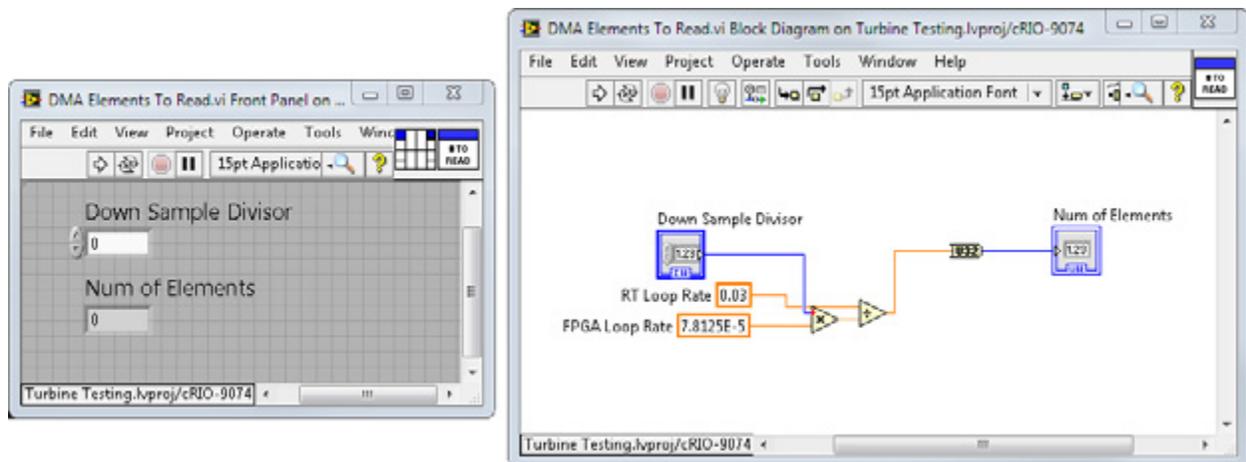


Figure 3.6. Create modular code by creating subVIs.

For more information on creating subVIs, see the video [Creating SubVIs in NI LabVIEW](#).

Interprocess Data Communication

One of the most important factors to consider when designing an embedded application is data communication. This section examines recommended data transfer mechanisms for each of the data communication models discussed in Chapter 1 for interprocess communication on a real-time target. Table 3.2 provides a recommended data communication mechanism for each model of communication. The data communication mechanisms recommended for data transfer between two nondeterministic loops are generally more flexible than the mechanisms recommended for data transfer involving a time-critical or deterministic loop.

Communication Model	Description	Data Transfer Mechanism (between two non-critical loops)	Data Transfer Mechanism (to or from a time-critical loop)
Current Value (Tag)	The transfer of the latest values only. Typically distributed to multiple processes.	Single-Process Shared Variable or Current Value Table (CVT)	Single-Process Shared Variable with RT FIFO Enabled (single-element)
Streaming	The high-throughput transfer of every data point, typically from one process that transfers data from the FPGA to another process that saves the data to disk or sends it across the network.	RT FIFOs	RT FIFOs
Command-/Message-Based	The low-latency data transfer from one process that triggers a specific event on another process. Command-based communication is typically infrequent and requires you to not miss any data points.	Queues	RT FIFOs

Table 3.2. Recommended Interprocess Data Communication Mechanisms for Each Communication Model

Note that the transfer type is only one of the considerations when choosing a data communication mechanism. You may have valid use cases for implementing an update using a command-based mechanism or vice versa based on the other considerations mentioned such as ease of implementation, scalability, or performance. The next section offers best practices for implementing each type of communication model in LabVIEW Real-Time.

Single-Process Shared Variables

Shared variables are configured software items that can send data between two locations in a block diagram that cannot be connected with wires, between two VIs running on a real-time target, or between two VIs across a network running on different targets. Use single-process shared variables to share current value data or tags on a real-time target and use network-published shared variables to share data between VIs on different targets. Single-process shared variables are implemented as global variables under the hood.

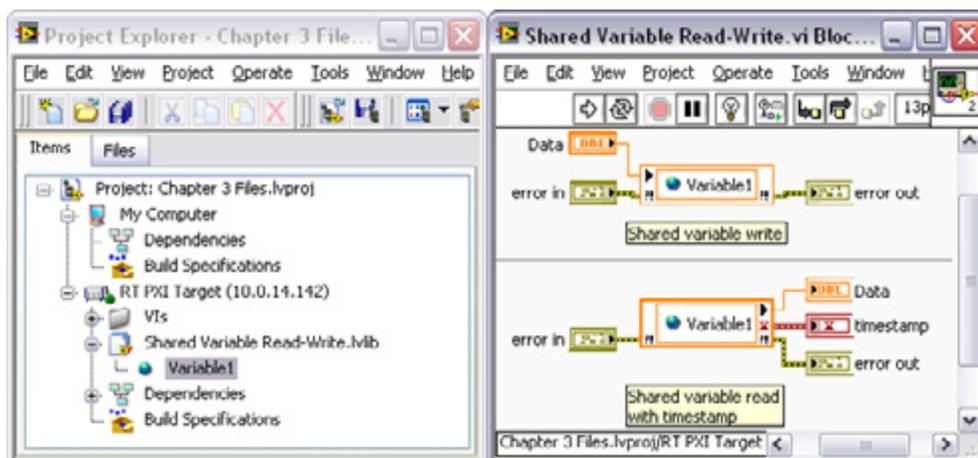


Figure 3.7. You can use single-process shared variables for variable communication between multiple processes.

To create a single-process shared variable, right-click a real-time target in the Project Explorer window and select **New » Variable** from the shortcut menu. In the Shared Variable Properties dialog box, you can set the variable type as Single Process (a single-process shared variable is a global variable).

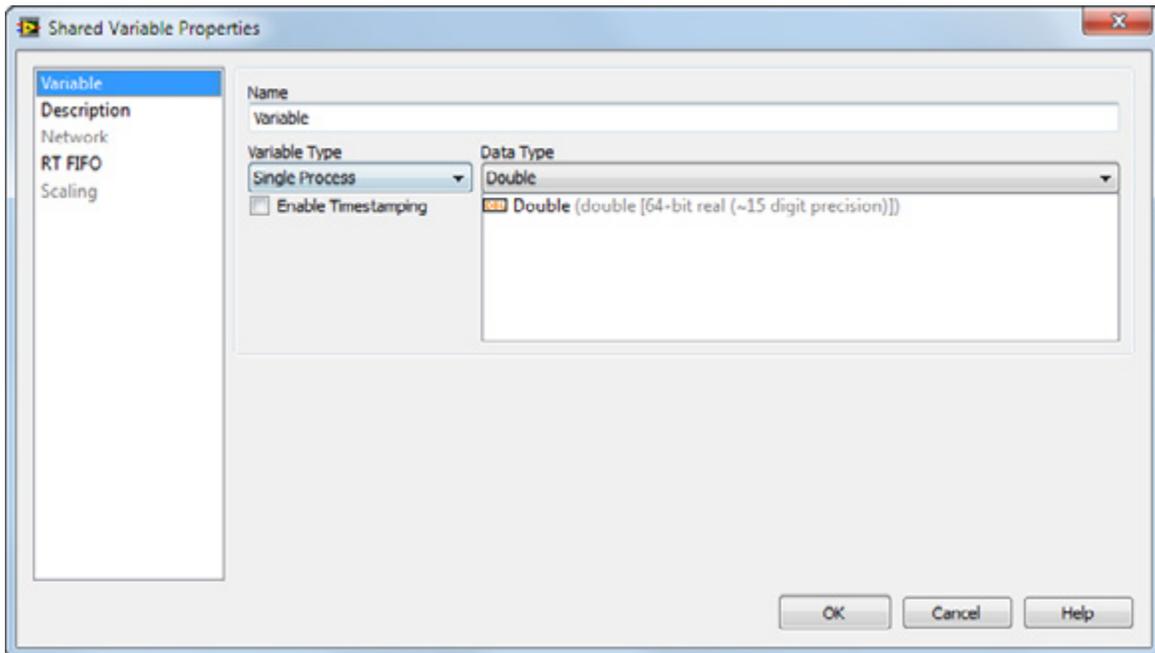


Figure 3.8. Recommended Interprocess Data Communication Mechanisms for Each Communication Model

If your application contains a relatively large number of variables and you require tag management features such as dynamic tag lookup, you should consider the current value table described below, which is designed to handle a large number of tags efficiently.

Since single-process shared variables are based on global variables, they act as a shared resource, which can introduce jitter if used within a time-critical loop. If you are interested in using single-process shared variables within a time-critical loop, be sure to enable RT FIFOs in the variable properties dialog. Single-process shared variables with RT FIFOs enabled are discussed in the upcoming section titled [RT FIFOs](#).

Current Value Table (CVT)

The current value lookup table offers a mechanism for sharing tags across an application with dynamic tag lookup and group management features that are useful when dealing with a large number of tags. Single-process shared variables and global variables do not offer dynamic lookup features. The current value table (CVT) allows application components to share a common data repository and have direct access to the most up-to-date value of any variable used between components. Application operations such as alarm detection, user interface updates, process logic, and so on can then be handled by separate processes sharing the same data repository.

The CVT contains three sets of API functions to provide different interfaces that you can choose according to your specific application needs. The basic API offers simple write and read functionality. Two additional APIs, the static API and index API, provide a higher performance interface to the CVT but place some restrictions on the application when using the CVT. Example programs are included for each API.

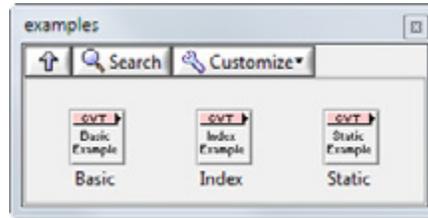


Figure 3.9. Examples for each API are installed with the CVT library as a starting point.

The basic API writer example is shown in Figure 3.10. In the API call, the variable name is used to look up the index of the variable, and then the value is accessed from the CVT using the index. This example assigns random number data to Var0–Var4 tags and writes to the Stop and Index tags. You can initialize the tags using a cluster array, as shown in Figure 3.10, or load them from a file using the CVT Load Tag List VI.

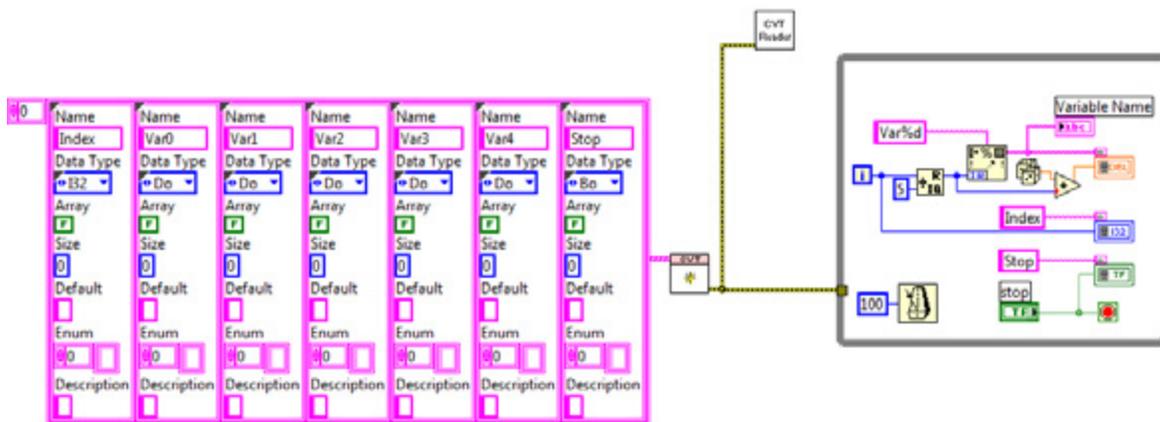


Figure 3.10. Basic CVT Writer Example

The Basic CVT Reader example in Figure 3.11 reads the current value of the tags within another instance of the application. Both examples use the Format Into String function to dynamically look up Var0–Var4 tags.

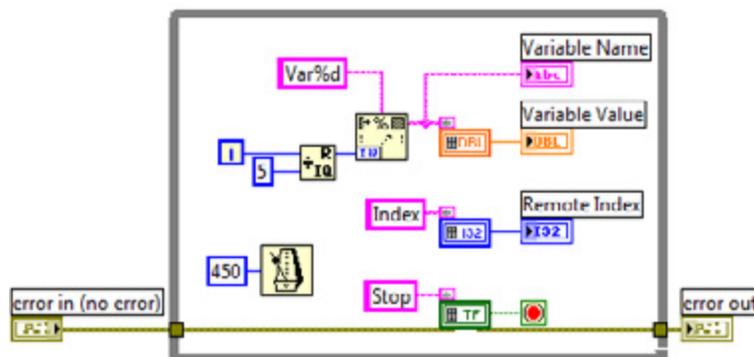


Figure 3.11. Basic CVT Reader Example

You can find more information on the CVT in the NI Developer Zone document [Current Value Lookup Table Reference Library](#).

Instructions for Installing the CVT Library

- Step 1: Navigate to ni.com/labviewtools
- Step 2: Download and install the VI Package Manager
- Step 3: Within VI Package Manager, search for "CVT"

Queues

Queues are the recommended protocol for command- or message-based communication between two nondeterministic processes because they are flexible, easy to use, and allow you to transfer buffered data. When sending messages, you might need to send multiple data types within a single packet. Queues support the Cluster data type, which you can use to bundle together different data types, both fixed size and variable size such as strings. You can use queues on both real-time and Windows OSs.

The block diagram in Figure 3.12 shows a loop that performs a data acquisition task. The data acquisition loop shares the acquired data with the second loop using queues. The second loop logs the acquired data that it reads from the queue to disk on the real-time target.

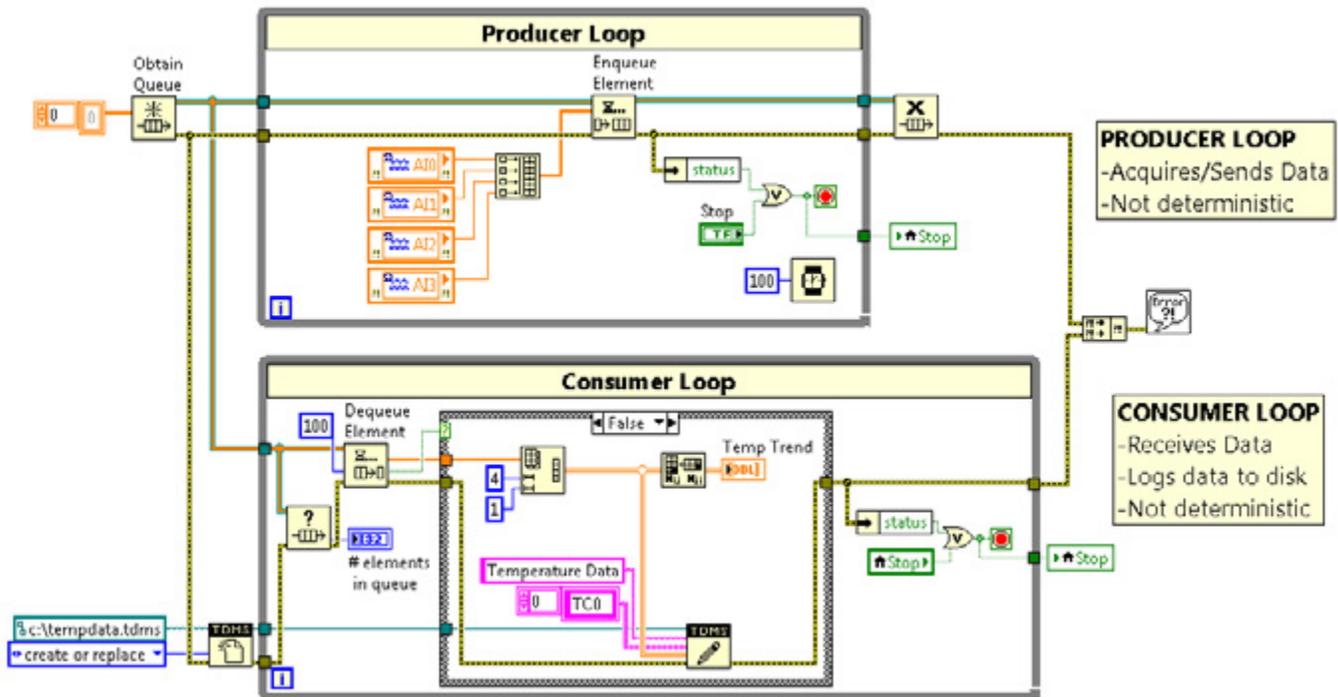


Figure 3.12. You can use queues to transfer buffered data between two nondeterministic loops.

The Queue Operations palette is included in LabVIEW on the Functions palette under **Synchronization»Queue Operations**. You can find example programs using queues in the NI Example Finder.

RT FIFOs

RT FIFOs are first-in-first-out memory buffers that behave deterministically. You can implement them using RT FIFO functions that are similar to the Queue functions, or with an RT FIFO-enabled shared variable. Both mechanisms are the same under the hood. While RT FIFO-enabled shared variables have a simpler implementation than RT FIFO functions, RT FIFO functions give you more control. For example, with RT FIFO functions, you can decide when the FIFOs are created and when they are shut down. A shared variable is automatically created when it is first called, and it is shut down when the application is stopped. You also have control over configuration options such as polling versus blocking for optimizing performance, and you can use more complex data types such as clusters (as long as they contain fixed-size elements). This section discusses both RT FIFO functions and RT FIFO-enabled shared variables.

RT FIFO Functions

RT FIFO functions are similar to Queue functions but are less flexible and more deterministic. They are recommended for streaming data between any two processes on an RTOS because they always preallocate memory and have a maximum buffer size. RT FIFOs are also recommended for transferring commands or messages to or from a time-critical loop. They provide a deterministic data transfer method that does not add jitter. Queues and RT FIFOs differ in the following ways:

- Queues can handle string, variant, and other variable size data types, but RT FIFOs cannot.
- RT FIFOs are fixed in size, so they can communicate data between Timed Loops without dynamically allocating new memory. Queues can grow as elements are added to them.
- Queues use blocking calls when reading/writing to a shared resource, which can block another loop from running and impact determinism. RT FIFOs do not use blocking calls.
- RT FIFOs execute regardless of errors at input to maintain determinism; queues do not execute on error.

The block diagram in Figure 3.13 is similar to the previous example using queues, but since the data acquisition loop is now deterministic, you use RT FIFO functions to share data. The deterministic loop with a priority of 100 shares the acquired data with the nondeterministic loop using an RT FIFO function. The nondeterministic loop logs the acquired data that it reads from the RT FIFO to disk on the real-time target. You can download a more advanced example of RT FIFOs from [Chapter 4: Best Practices for Network Communication](#).

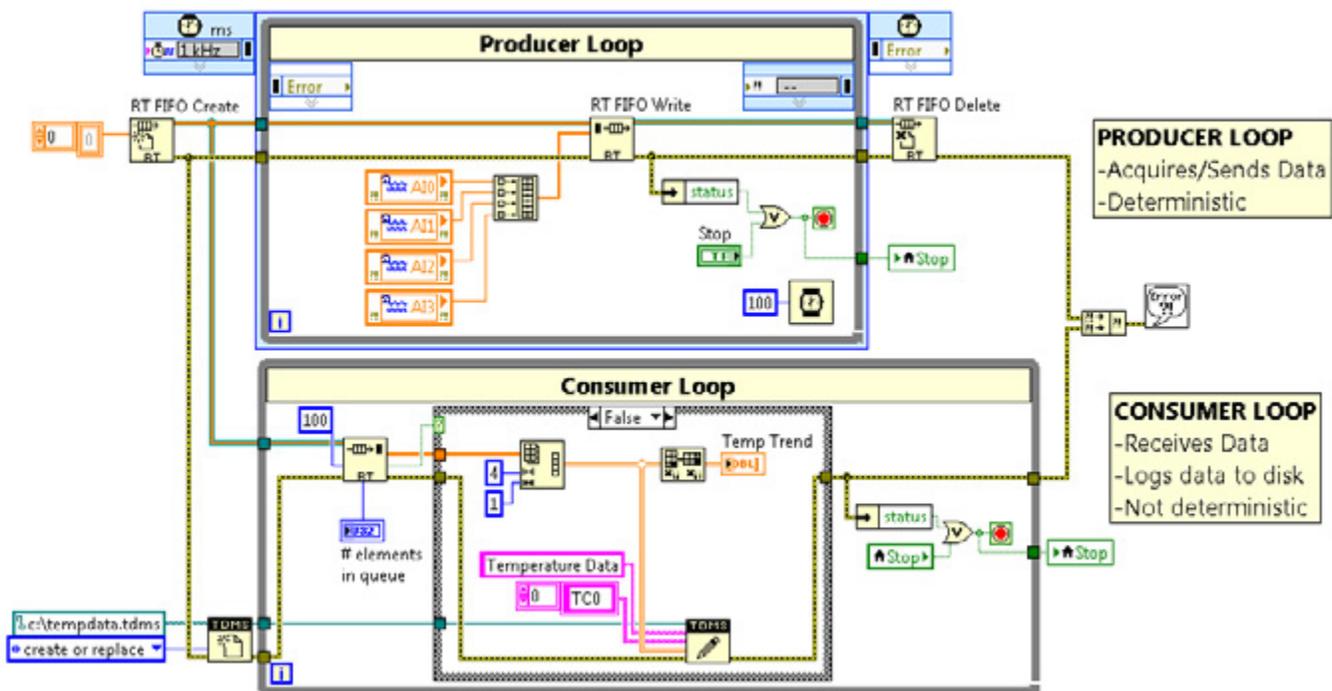


Figure 3.13. You can use RT FIFOs to transfer buffered data to or from a time-critical loop.

Because of the fixed-size restriction, an RT FIFO can be a lossy communication method. Writing data to an RT FIFO when the FIFO is full overwrites the oldest element. You must read data stored in an RT FIFO before the FIFO is full to ensure the transfer of every element without losing data. Check the **overwrite** output of the RT FIFO Write function to ensure that you did not overwrite data. If the RT FIFO overwrites data, the **overwrite** output returns a TRUE value.

The RT FIFO palette is included in LabVIEW Real-Time on the Functions palette under **LabVIEW Real-Time»RT FIFO**. You can find example programs featuring RT FIFOs in the NI Example Finder.

RT FIFO-Enabled Shared Variables

You can deterministically transfer data between two loops using single-process shared variables with RT FIFOs enabled and deterministically transfer data across a network using network-published shared variables with RT FIFOs enabled. The block diagram in Figure 3.14 is similar to the previous example, but it uses RT FIFO-enabled shared variables instead of RT FIFO functions to share data. The time-critical loop with a priority of 100 shares the acquired data with the nondeterministic loop using a real-time FIFO-enabled shared variable named **Loop Comm**. The nondeterministic loop logs the acquired data that it reads from the shared variable **Loop Comm** to disk on the real-time target.

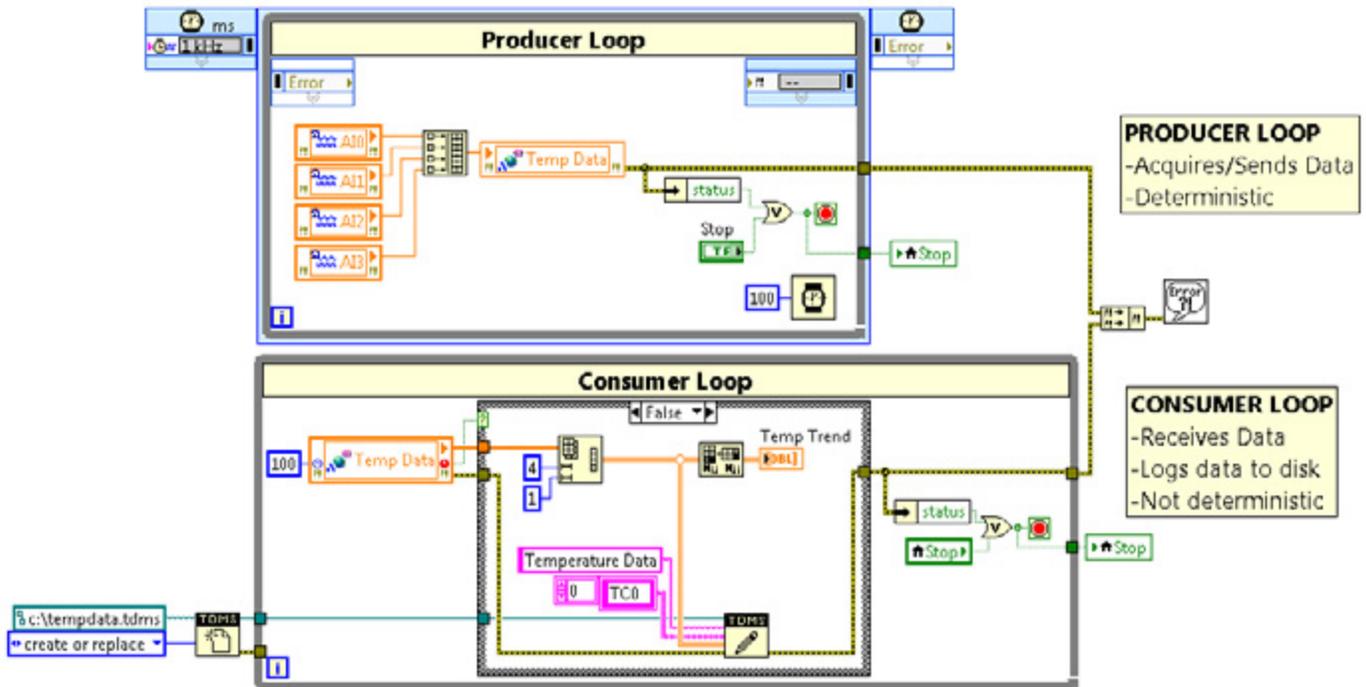


Figure 3.14. You can use RT FIFO-enabled shared variables to transfer buffered data to or from a time-critical loop.

To enable the RT FIFO on a shared variable, navigate to the RT FIFO page of the Shared Variable Properties dialog box and place a checkmark in the Enable Real-Time FIFO checkbox.

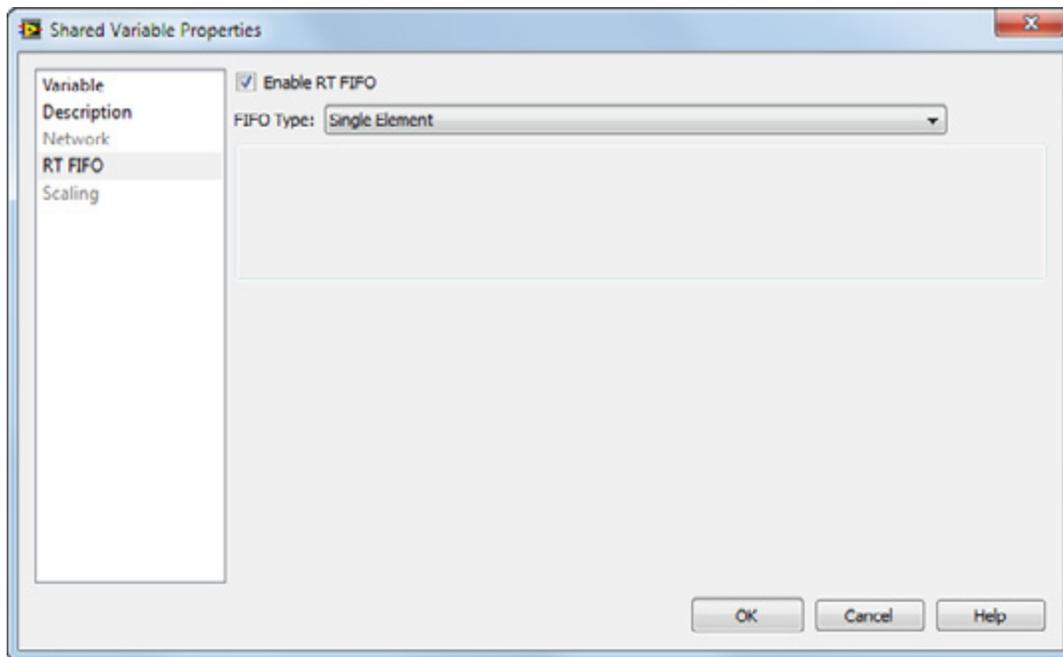


Figure 3.15. By enabling the RT FIFO, you can read and write to a shared variable from multiple parallel loops without inducing jitter.

Use a single-element RT FIFO to transfer the latest values (tags or updates) and use a multi-element RT FIFO to transfer the buffered values (messages or streams). A multi-element FIFO features a buffer for which each write adds another element to the buffer and each read removes an element from the buffer. As long as the consumer loop checks the FIFO frequently, you can leave the buffer at the default of two elements and not miss any triggers/commands.

To check the FIFO, the consumer task must read the shared variable and check the error status. If the FIFO is empty, the shared variable returns the 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 consumers receive a command from the same FIFO, but you can have multiple commanders putting commands into the FIFO.

LabVIEW Real-Time Design Patterns

A design pattern is a reusable solution to a common problem in software engineering. By using design patterns in your LabVIEW Real-Time applications, you can take advantage of the accumulated experience of the software engineering community. The benefits of design patterns include the following:

- Faster development time
- Greater reliability
- Better code reusability
- Easier debugging
- Enhanced maintainability

Before viewing some common design patterns, you need to understand your options for synchronizing loops in LabVIEW Real-Time. Synchronization is the foundation of any design pattern.

Synchronization and Timing

When implementing a design pattern in LabVIEW Real-Time, one of your most important considerations is the synchronization of your loops. Almost all LabVIEW Real-Time applications consist of two or more loops that have different synchronization needs. For example, consider the turbine testing application that you downloaded from Section 1 Downloads. The turbine testing application consists of multiple loops with multiple types of synchronization. Each loop must have only one source of synchronization (with the state machine being an exception). For example, if you have one task that needs to run continuously and one task that is waiting on messages, implementing them within the same loop would be problematic.

Periodic Loop

A periodic loop runs continuously at a defined period. The Turbine PWM Characteristics loop in the turbine testing application is periodic. This loop, shown in Figure 3.16, executes at a period of 100 ms or 10 Hz. It uses variables for communication. Be careful not to add any functions that would introduce blocking or other types of synchronization (for example, queues with no timeout) when designing period loops.

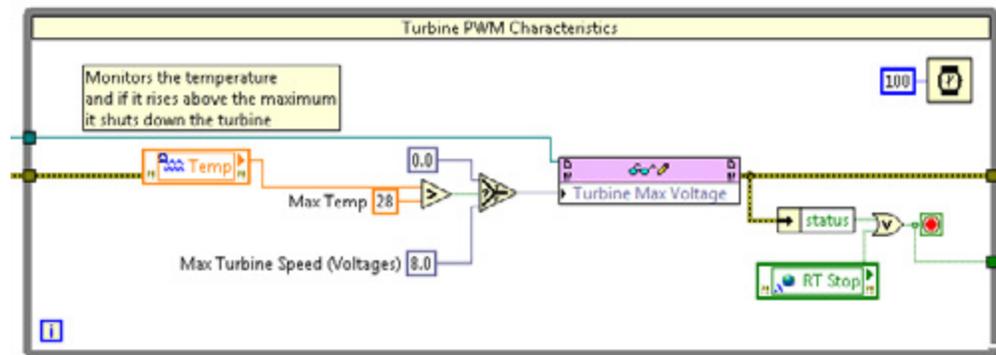


Figure 3.16. The Turbine PWM Characteristics loop in the turbine testing application executes periodically.

You also can implement periodic loops for a higher priority task by using a Timed Loop, similar to the Data Acquisition and Analysis loop. You generally should use Timed Loops only when implementing a periodic task since any type of blocking introduces jitter.

Event-Driven Loop

An event-driven loop executes only when an event occurs. Often the event is receiving a message from another process. The message loop in the turbine testing application is event driven. In the command parser example, the synchronization source is the Dequeue Element function. The timeout is set to default (-1), which means the function never times out and the loop executes only when a command is received from the host. Other commonly used functions that force an event-driven architecture include the following:

- Queues
- Notifiers
- RT FIFOs
- User events (Event Structure)
- Network Streams

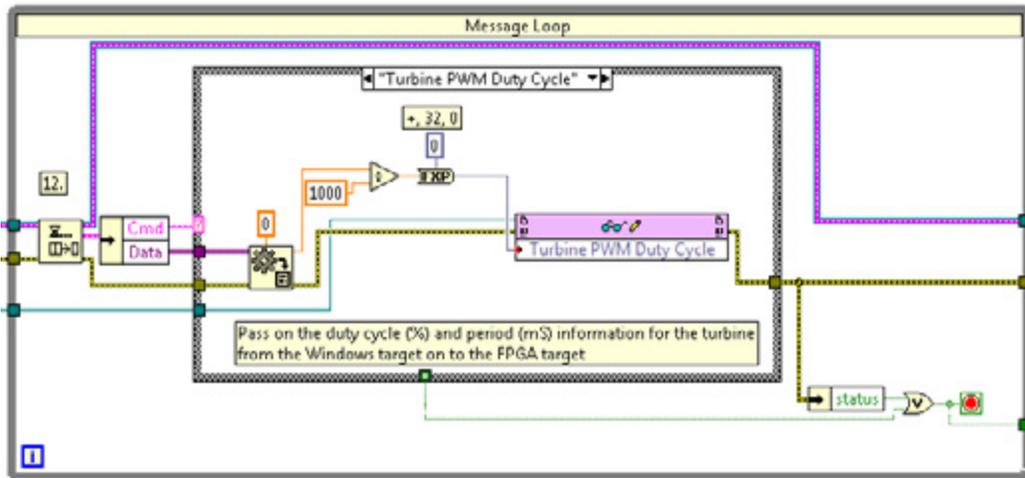


Figure 3.17. Example of a Loop With Event-Driven Synchronization

When using an event-driven design pattern, you can block until a message is received or until a timeout occurs. When blocking with no timeout, you are efficiently using the CPU, but you are also preventing any background tasks from running. If you add a timeout to any of these functions, you can have a “timeout” state during which you can execute background tasks while waiting on messages.

Scheduled Loop

A scheduled loop executes at an absolute time. An example is a maintenance routine that executes every day at midnight. This type of synchronization is less common.

Design Patterns

The design patterns common in LabVIEW Real-Time applications are similar to those used in LabVIEW for Windows applications. This section examines two design patterns: state machine and producer consumer.

State Machine

A state machine is a common and useful design pattern. You can use a state machine to implement any algorithm that can be explicitly described by a state diagram or flowchart. A state machine usually illustrates a moderately complex decision-making algorithm, such as a diagnostic routine or a process monitor. To learn the basics of the state machine architecture in LabVIEW, see the NI Developer Zone document [Application Design Patterns: State Machine](#).

Producer Consumer

A producer consumer design pattern is used to transfer data between processes that produce and consume data at different rates. On Windows OSs, a producer consumer is commonly implemented with Queue functions. In LabVIEW Real-Time, it can be implemented with Queues or RT FIFOs. You can implement a producer consumer design pattern so that it shares data between two loops or it shares events.

Data-Sharing Producer Consumer

Use a data-sharing producer consumer design pattern when you need to execute a process such as data analysis, when a data source, such as a triggered acquisition, produces data at an uneven rate and you need to execute the process when the data becomes available.

Event-Sharing Producer Consumer

Use an event-sharing producer consumer design pattern when you want to execute code asynchronously in response to an event without slowing the user interface responsiveness.

An example of a producer consumer event-sharing design pattern is the HMI included in the turbine testing application, shown in Figure 3.18. This application has one loop dedicated to receiving events from the user interface and a second loop that processes the commands and sends them across the network to the CompactRIO controller. Since network communication functions rely on an external resource, the network, they could impact the responsiveness of the user interface if placed in a UI Event Handler loop.

Because this application runs on a Windows host PC, you use queues to share data between the two loops. A real-time application can use either queues or RT FIFOs to share data depending on whether a time-critical loop is involved.

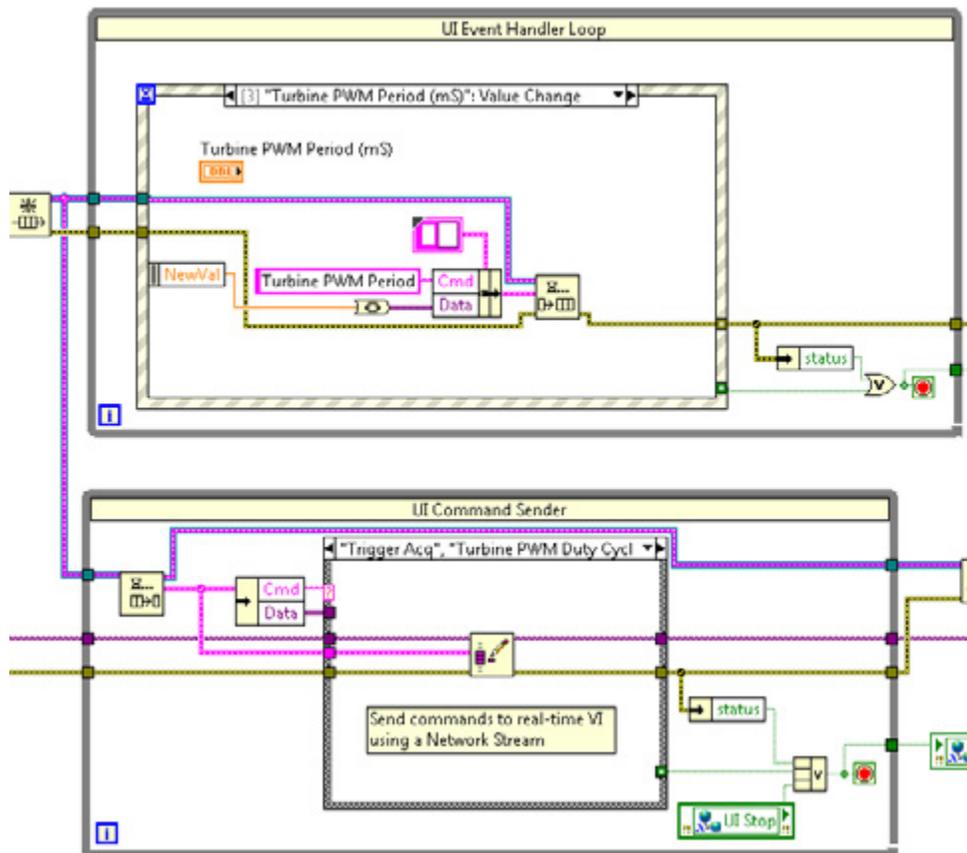


Figure 3.18. Example of a Producer Consumer With Events

Working With Constrained Resources

Embedded hardware targets are typically much more resource constrained than desktop PCs. Disk space, RAM, and CPU bandwidth are typically limited. Because of this, you need to monitor these resources when developing and testing applications to ensure that you do not run into limits that introduce negative side effects. This section discusses tools and techniques to help you overcome constrained hardware resource issues.

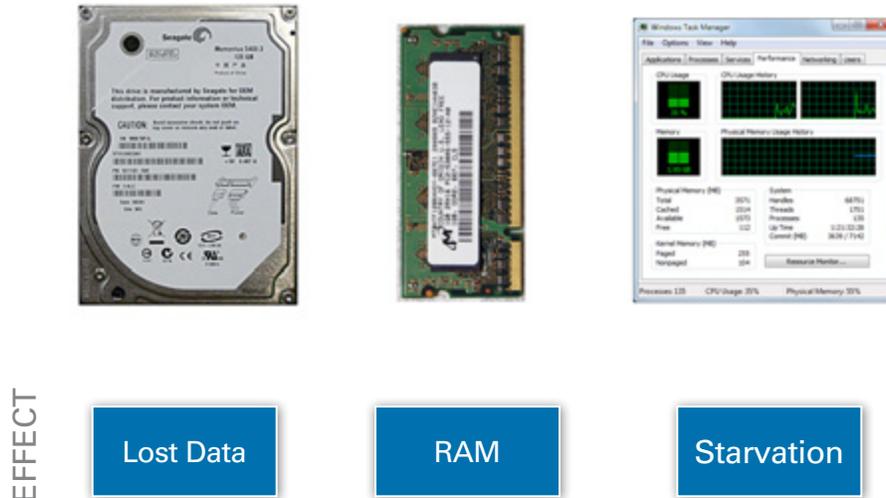


Figure 3.19. Disk space, RAM, and CPU bandwidth are typically limited in embedded application design.

Working With Limited Disk Space

If you are going to log data to a CompactRIO controller, you need to know your storage capacity for saving information. The amount of hard disk or nonvolatile storage on a CompactRIO controller depends on several factors including the installed software and files on the controller. You can check the memory on a real-time target using NI Measurement & Automation Explorer (MAX) configuration software or programmatically within the real-time VI. To view the memory in MAX, select your CompactRIO target under Remote Systems and view the Free Disk Space in the System Monitor section.

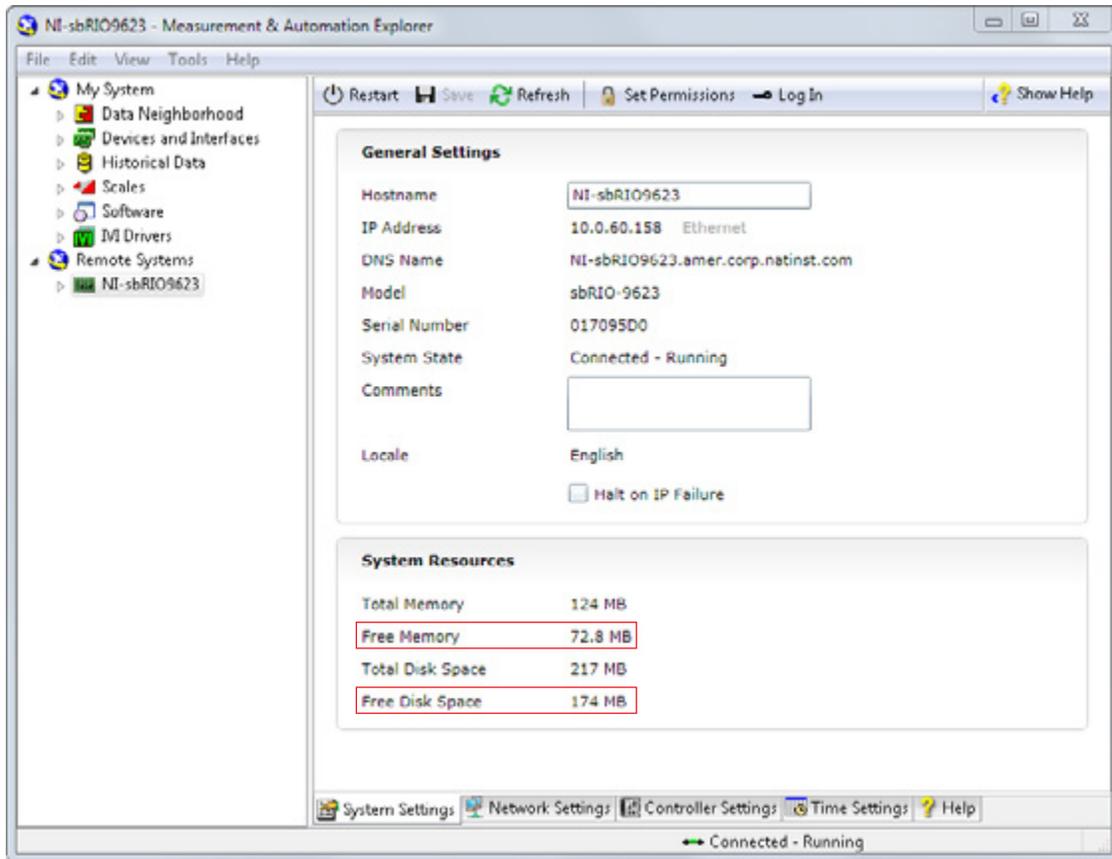


Figure 3.20. Check free disk space and memory within MAX.

To programmatically obtain the free disk space, use the Get Volume Info function when targeted to the controller. You can find this function on the Advanced File Functions palette (**Programming»File I/O»Advanced File Functions»Get Volume Info**).

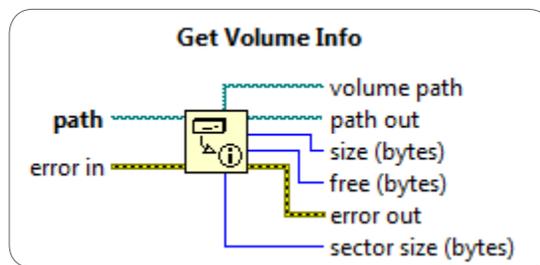


Figure 3.21. Check free disk space programmatically.

Working With Limited RAM

Since CompactRIO systems have less RAM than a desktop PC, memory management is extremely important. You need to know two types of memory allocations when designing software for a CompactRIO system: memory preallocation and dynamic memory allocations

Memory Preallocation

One type of memory preallocation occurs when memory is allocated for memory stored in the data space of your VI. If you attempt to preallocate more memory than what is available, you receive an error message during deployment or your VI terminates at the beginning of execution.

Fixed-size memory that is allocated during initialization is also included in this category since it does not affect fragmentation or engage the memory manager after the first call. A common example of a fixed-size memory allocation is the allocation of a large array, similar to the block diagram shown in Figure 3.22.

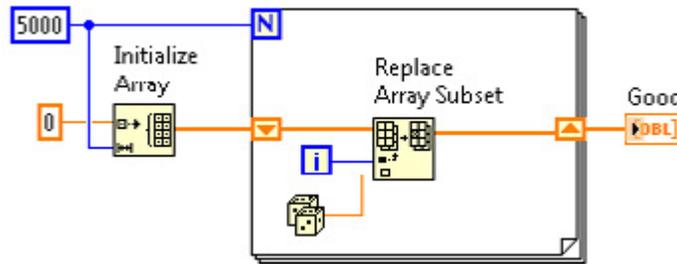


Figure 3.22. Preallocating a Large Block of Memory by Initializing a Large Array

If your LabVIEW code does not fit onto your real-time target, or you find that you are allocating more fixed-size memory than what is available during initialization, you can take one of the following actions:

- Eliminate memory copies
- Eliminate unnecessary drivers
- Choose a hardware target with more onboard memory

Eliminate Memory Copies

You can use the Show Buffer Allocations Window to identify where LabVIEW can create copies of data. To display the Show Buffer Allocations Window, select **Tools»Profile»Show Buffer Allocations**. Place a checkmark next to the data type(s) you want to see buffers for and click the Refresh button. The black squares that appear on the block diagram indicate where LabVIEW creates buffers to allocate space for data.

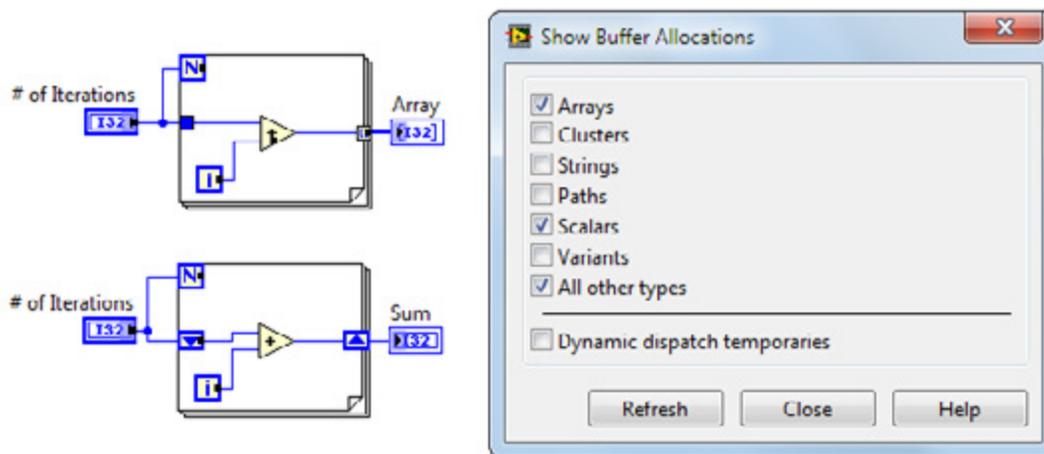


Figure 3.23. Show Buffer Allocations Window

Once you know where LabVIEW creates buffers, you might be able to edit the VI to reduce the amount of memory LabVIEW requires to run the VI. For tips on reducing memory copies, see the NI Help document "VI Memory Usage" starting at the section [Determining When Outputs Can Reuse Input Buffers](#).

Eliminate Unnecessary Drivers

To free up some of your system memory, you can uninstall any drivers that you are not using on your real-time system. To customize the software stack, right-click Software under your real-time target in MAX and select Add/Remove Software. Select the recommended software set (item that has the gold ribbon next to it) and remove any unnecessary drivers.

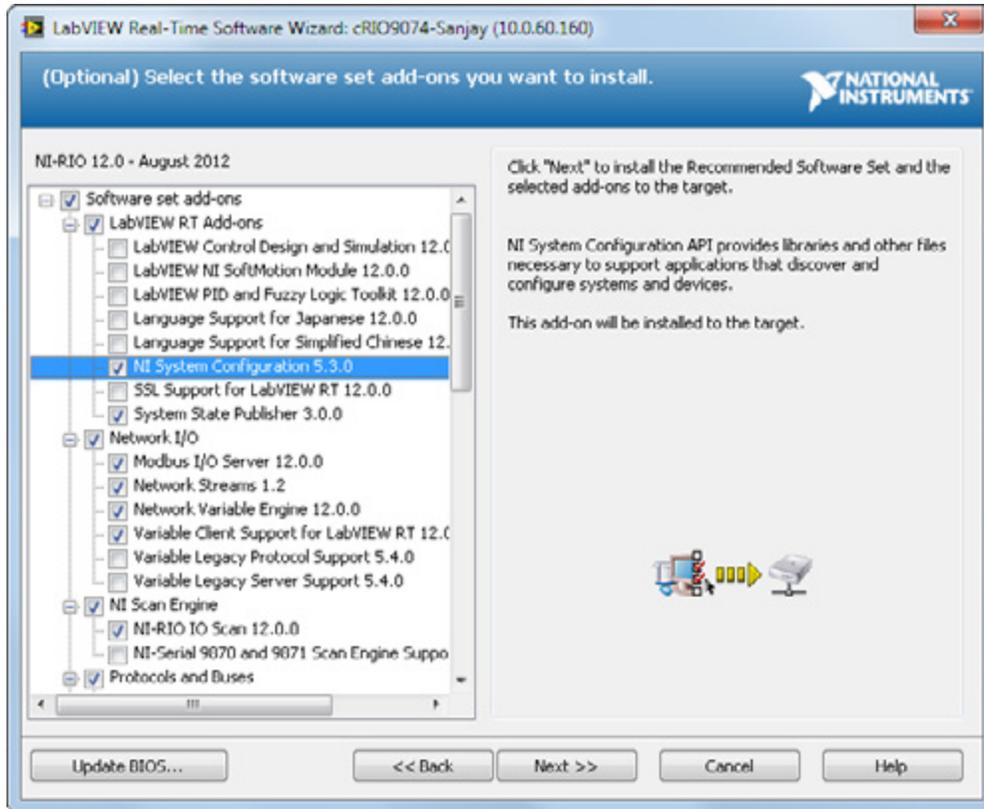


Figure 3.24. Uninstall software drivers that you are not using to increase memory.

Choose a Hardware Target With More Onboard Memory

You can choose from a variety of CompactRIO and NI Single-Board RIO targets with varying amounts of onboard memory. Consider using a CompactRIO system with more onboard memory if you have issues with static memory allocations.

Dynamic Memory Allocation

Dynamic memory allocation is memory allocated during run time. Dynamic memory allocations affect the performance and determinism of an application because they call into the memory manager, which acts as a shared resource. The time to dynamically allocate memory varies depending on the amount of memory you need to allocate and the current state of the memory. Dynamic memory allocation also affects the reliability of a real-time application. If the memory manager cannot find a large enough contiguous segment in memory to fill a request, it terminates the program.

The level of effort required to design your code so that it avoids dynamic memory allocations should correlate to the required uptime of your embedded system. Systems that can handle regular reboots or maintenance can use dynamic memory freely as long as they monitor the memory status. As soon as a system is rebooted, the memory frees up. If the system cannot handle regular reboots, then you should consider the following techniques to reduce dynamic memory allocations. For maximum reliability, move your code to the FPGA or create a redundant system.

Avoid Memory Leaks by Closing References

A memory leak is an allocation of a resource that is never released. Memory leaks might take a long time to deplete the memory but could eventually cause your system to crash. Common sources of memory leaks are listed below:

- Calls to libraries with leaks
- Unclosed file handles
- Unclosed VI server references
- Unclosed driver handles
- Unclosed Shared Variable API references
- Unclosed TCP connection IDs or Listener IDs

Avoid memory leaks by closing all of the references that you open during initialization. It is also a safe practice to replace the terminals of any reference with shift registers.

Avoid Overallocation by Using Fixed-Size Data

Overallocation occurs when the program tries to store too much data in RAM. It typically results from a queue or buffer without a fixed size. The graph in Figure 3.25 shows a situation where a buffer is expanding. Note that once a buffer expands, it generally does not contract. The buffer could be getting periodically emptied as the code runs, but it still retains the size of its largest value.

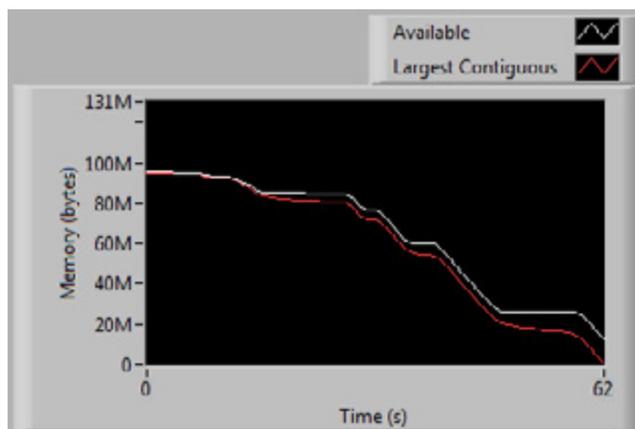


Figure 3.25. An unbounded buffer decreases memory over time.

The following items are common sources of dynamic memory allocation:

- Queues without fixed size
- Variable-sized arrays (and waveforms)
- Variable-sized strings
- Variants

Two methods for sharing data between processes on a real-time target are discussed in this chapter. By default, queues have a variable-sized buffer. Even with a fixed number of elements, a queue that contains variable-sized data (strings, variants) is still variable sized. This is also true for Network Streams and shared variables. If you are working on an application for which dynamic memory allocation is a concern, use RT FIFOs for transferring data between processes. RT FIFOs limit you to a fixed-sized buffer and fixed-sized data types.

Keep Your Contiguous Memory Healthy

Contiguous memory is a continuous, unfragmented block of memory. When programs contain a mixture of dynamic allocations, memory can become fragmented and make it more difficult to find a large block of contiguous memory. In the example in Figure 3.26 even though the RTOS has more than enough memory to store the 40-byte requested allocation, the program fails and crashes because the RTOS does not have a large enough contiguous segment.

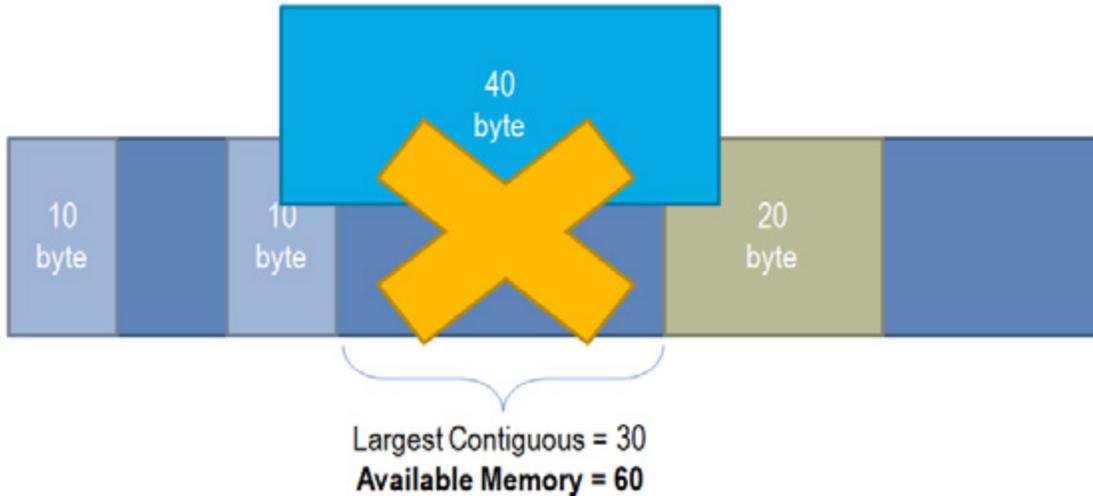


Figure 3.26. Memory on an RTOS becomes fragmented over time, making it difficult for the memory manager to locate a large block of contiguous memory.

You can keep your contiguous memory healthy by minimizing dynamic memory allocations and preallocating space for arrays equal to the largest expected array size. Figure 3.27 shows how you can preallocate memory for an array by using the Initialize Array and Replace Array Subset functions. The array is created only once and Replace Array Subset can reuse the input buffer for the output buffer. You should preallocate an array if you can determine the upper size limit of the array.

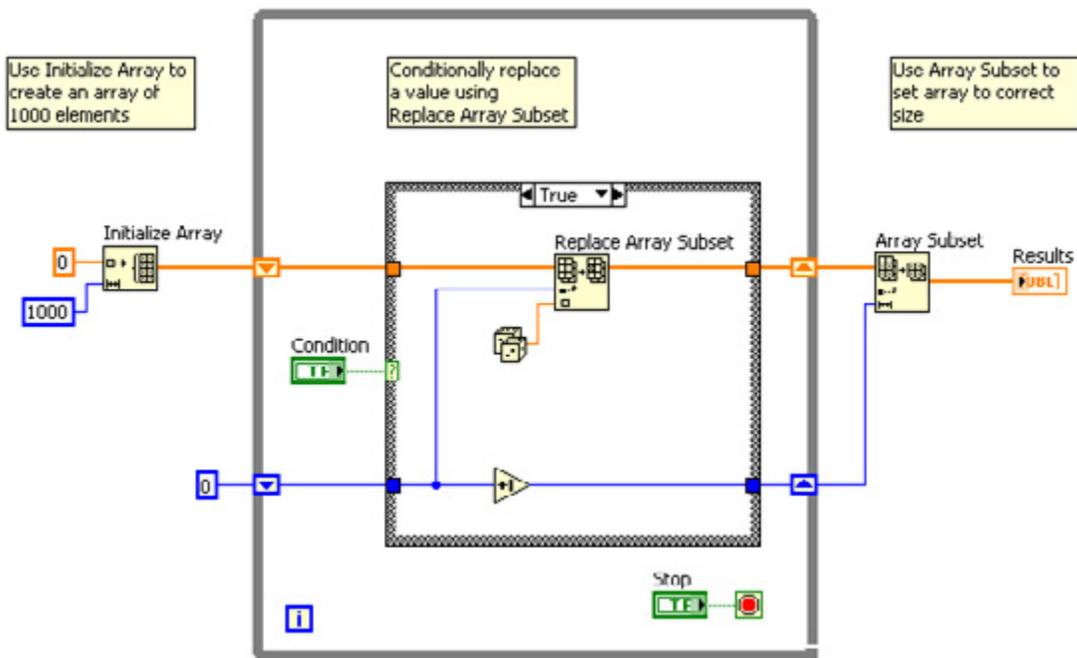


Figure 3.27. Preallocate memory for arrays when possible.

Monitor System Memory

When designing your system, consider including memory monitoring code that reboots the target safely when memory is low. You can programmatically access the memory status of your system using the RT Get Memory Usage VI when targeted to the real-time target.

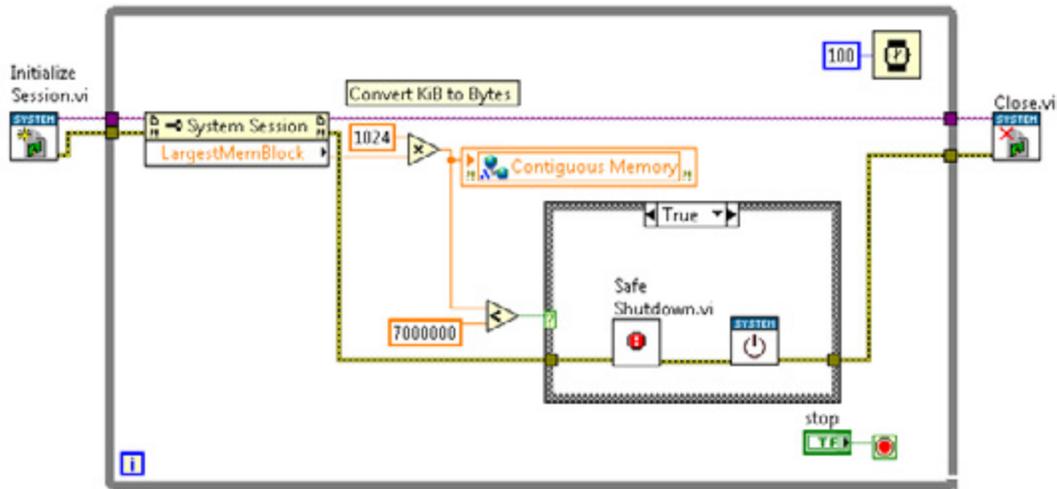


Figure 3.28. Programmatically monitor memory with the NI System Configuration API.

You should also include a safe shutdown routine that performs the following functions:

- Closes files to prevent corruption
- Terminates communication to avoid conflicts
- Sets hardware outputs to a safe state

Check out the NI Developer Zone document [Fail-Safe Control Reference Design](#) for more information on this topic.

You can also monitor the memory usage using the NI Distributed System Manager in LabVIEW under the Tools menu.

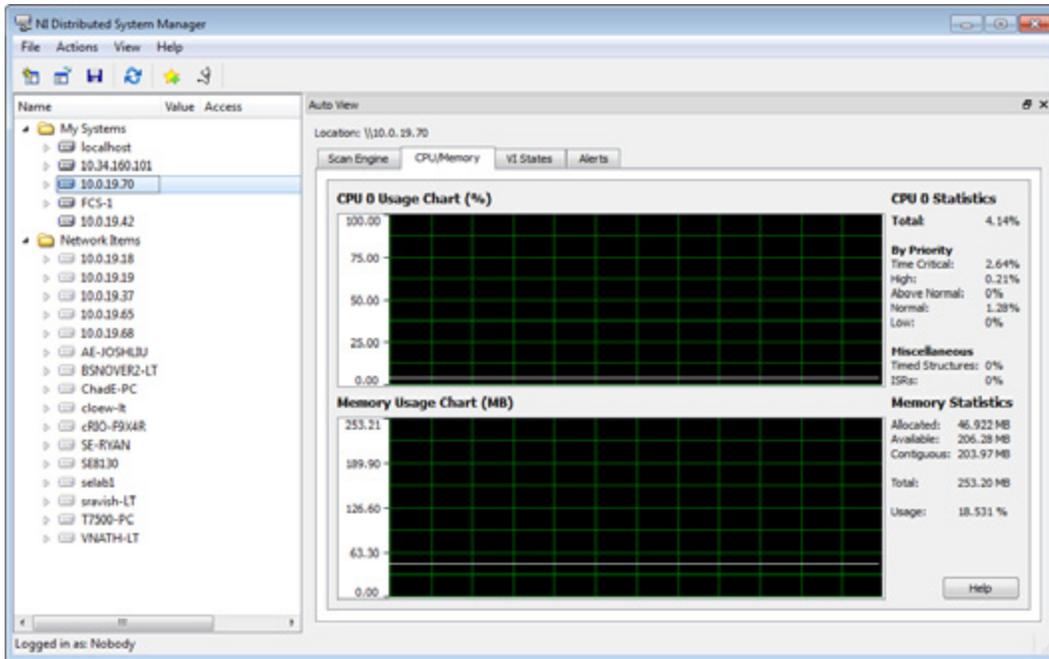


Figure 3.29. Monitor CompactRIO memory usage with the NI Distributed System Manager.

Working With Limited CPU Resources

When designing your real-time target application, you should aim for a CPU usage below 70 percent. You can monitor CPU usage with the NI Distributed System Manager or programmatically with the RT Get CPU Loads VI. Consider the following when trying to reduce the CPU usage of your application.

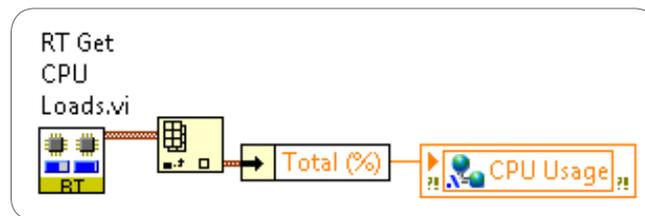


Figure 3.30. Monitor CPU usage with the RT Get CPU Loads VI.

Be Careful When Using Timed Structures

If your application does not contain any tasks that require deterministic behavior, use While Loops with timing functions instead of Timed Loops. Timed Loops offer many useful built-in features but also have more overhead than a While Loop. In addition, if your Timed Loops are not implemented properly, they can cause your system to become unresponsive. You should use a Timed Loop only when absolutely necessary.

Run Loops Only as Fast as Necessary

Although you may want to run each loop in your application as fast as possible, this practice can lead to undesired timing behavior, including increased jitter and even system deadlocks. For example, running a user interface data-publishing loop faster than the human operator can process and respond to the data taxes the CPU of the real-time target needlessly. In most cases, a rate of 2 Hz to 15 Hz is adequate for a loop that publishes user interface data over the network.

Avoid Using Too Many Network-Published Shared Variables

Network-published shared variables incur significant CPU and memory overhead when hosted on the CompactRIO controller. If your application uses a large number of network-published shared variables (more than a few dozen), host the shared variables on the Windows host PC if possible. Note that when hosting shared variables on a host PC, you cannot enable RT FIFOs for deterministic data transfer to and from a time-critical loop.

Offload Tasks When Possible

To minimize CPU usage on the real-time target, consider offloading certain tasks to either a desktop PC or an FPGA target, if available. For example, you can implement any time-critical tasks such as a PID control loop in LabVIEW FPGA, which increases the reliability of the system.

Ensuring Reliability With Watchdog Timers

When designing embedded systems that will be deployed, consider incorporating a watchdog timer to ensure reliability. A watchdog timer is a hardware counter that interfaces with the embedded software application to detect and recover from software failures. An example of a software failure is your application running out of memory, causing your application to hang or crash. Even if you followed the best practices for managing memory listed in the previous section, it's always important to have a backup plan.

All CompactRIO and NI Single-Board RIO controllers include a hardware timer that you can access from the LabVIEW Real-Time Module. 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, as shown in the Figure 3.31.

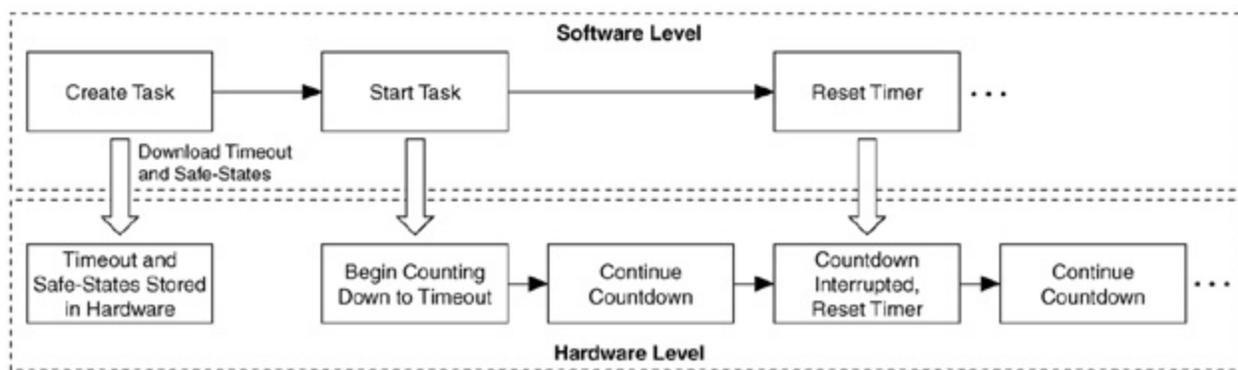


Figure 3.31. An application periodically resets the watchdog timer when the application responds on time.

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, as shown in Figure 3.32.

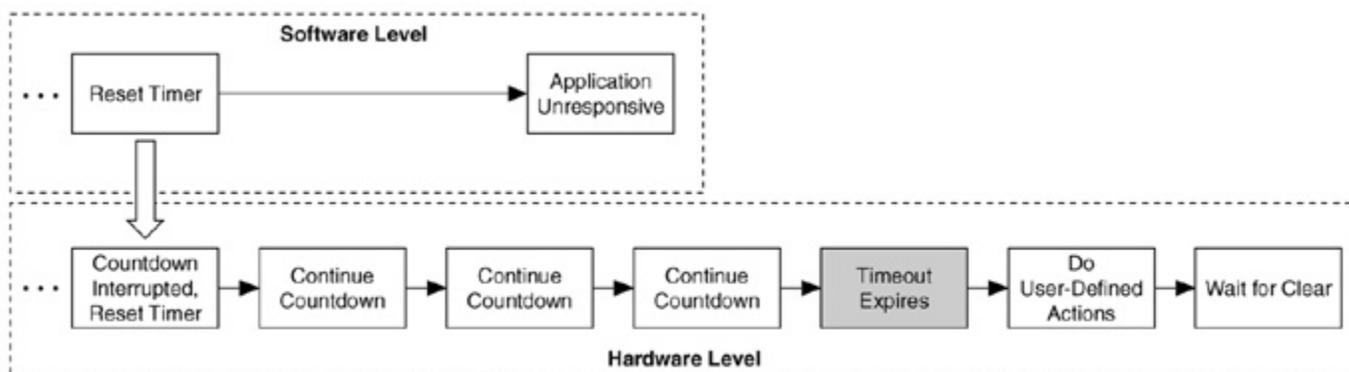


Figure 3.32. The hardware timer triggers a recovery procedure when the watchdog timer expires.

When preparing your embedded system for deployment, you can choose from two options for implementing a hardware-based watchdog timer within LabVIEW. You can access the built-in watchdog hardware that is available in all CompactRIO and NI Single-Board RIO controllers using the LabVIEW Real-Time Module, or you can implement your own watchdog timer using the LabVIEW FPGA Module. If you are writing to any hardware outputs from your LabVIEW FPGA VI, it may be beneficial to implement your watchdog timer in LabVIEW FPGA. If something goes

wrong, you can immediately and reliably put all of your hardware outputs into a safe state. Each option is described in more detail in the following sections.

LabVIEW Real-Time Watchdog

The LabVIEW Real-Time watchdog uses a hardware counter built in to the real-time controller that interfaces with the embedded software application. The RT Watchdog API can be found in the Real-Time palette, as shown in Figure 3.33.



Figure 3.33. The RT Watchdog API interfaces with a hardware counter that is built-in to CompactRIO and NI Single-Board RIO controllers.

When programming with the RT Watchdog API, you first need to configure the watchdog and set a timeout value. The appropriate range of timeout values depends on the specific performance characteristics and uptime 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 uptime requirements. In Figure 3.34, the watchdog timeout is set to 10 seconds.

Next you need to configure the expiration actions. Specifically, you must determine how you want the system to respond to a watchdog timeout. You have the option to reset the target or trigger an occurrence, which can be used to execute another piece of code if your watchdog loop becomes unresponsive. More information can be found in the LabVIEW Real-Time Help File titled Watchdog Configure VI.

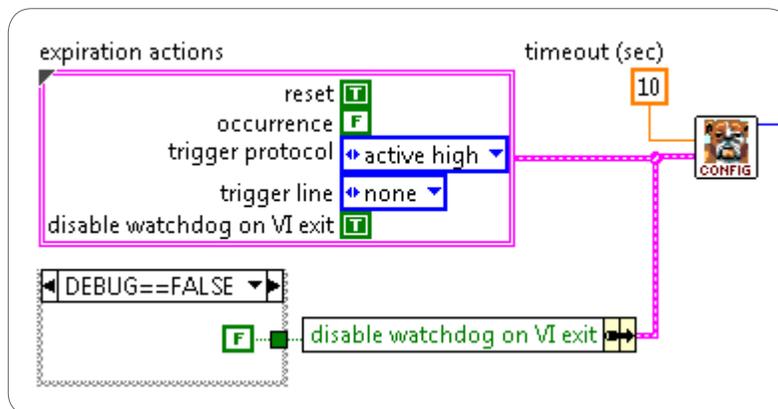


Figure 3.34. Configure the timeout value and expiration actions with the Watchdog Configure.vi.

Finally you want to whack or pet the watchdog periodically. Every time you whack the watchdog, you are resetting the watchdog timer. If something in your system causes the watchdog loop to become unresponsive (low memory,

low CPU bandwidth, and so on), the watchdog timer does not reset and you can recover your system based on the expiration actions that you defined during initialization. It is important to note that the watchdog timer does not start until the Watchdog Pet.vi or the Watchdog Start.vi has been executed.

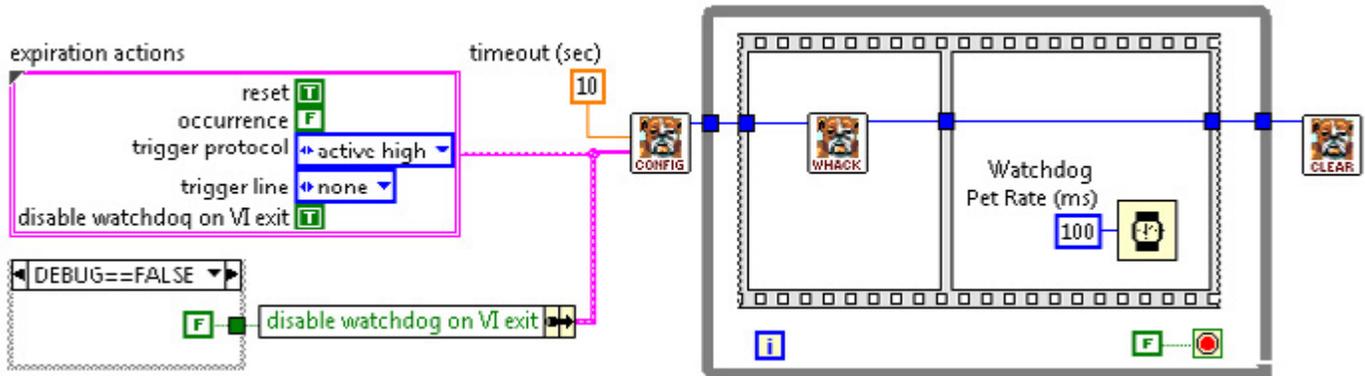


Figure 3.35. Configure the timeout value and expiration actions with the Watchdog Configure.vi

LabVIEW FPGA Watchdog and Fail-Safes

If your embedded application uses LabVIEW FPGA for any hardware outputs, you should consider implementing a watchdog timer on the FPGA fabric. This increases the reliability of your system and helps you put all of your hardware outputs into a safe state upon a software failure. When controlling dangerous or critical machinery, it is necessary to implement fail-safes to ensure that the machine operates safely even when elements of the control hardware or software fail.

Figure 3.36 shows an example of how you might implement logic in LabVIEW FPGA that determines when your system should go into a safe state. Note that one of the conditions you are monitoring is whether the watchdog is safe.

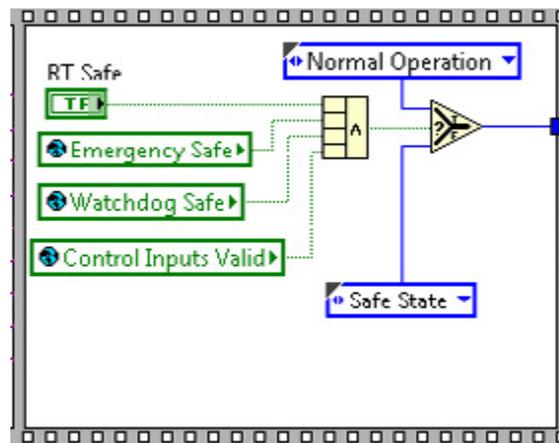


Figure 3.36. Define failure conditions within LabVIEW FPGA when implementing safe states.

You can use the following two reference designs to help you implement a LabVIEW FPGA based watchdog timer and safe states:

- **The Fail-Safe Control Reference Design for CompactRIO white paper**—This reference design written by NI Systems Engineering provides a framework that demonstrates FPGA safe states and FPGA-monitored watchdogs for the real-time controller.
- **The LabVIEW FPGA Control Sample Project**—This Sample Project included in LabVIEW 2012 is based on the Fail-Safe Control Reference Design linked above.

Software Watchdogs

Along with implementing hardware-based watchdogs, you can implement software-based watchdogs. The Fail-Safe Control Reference design has multiple software loops check-in with a software watchdog loop in addition to a LabVIEW FPGA-based watchdog. If any of these loops becomes unresponsive, the software watchdog can take action to fix it or reboot the system. This software watchdog loop then checks in with the hardware watchdog in case something happens to it or to the entire system.

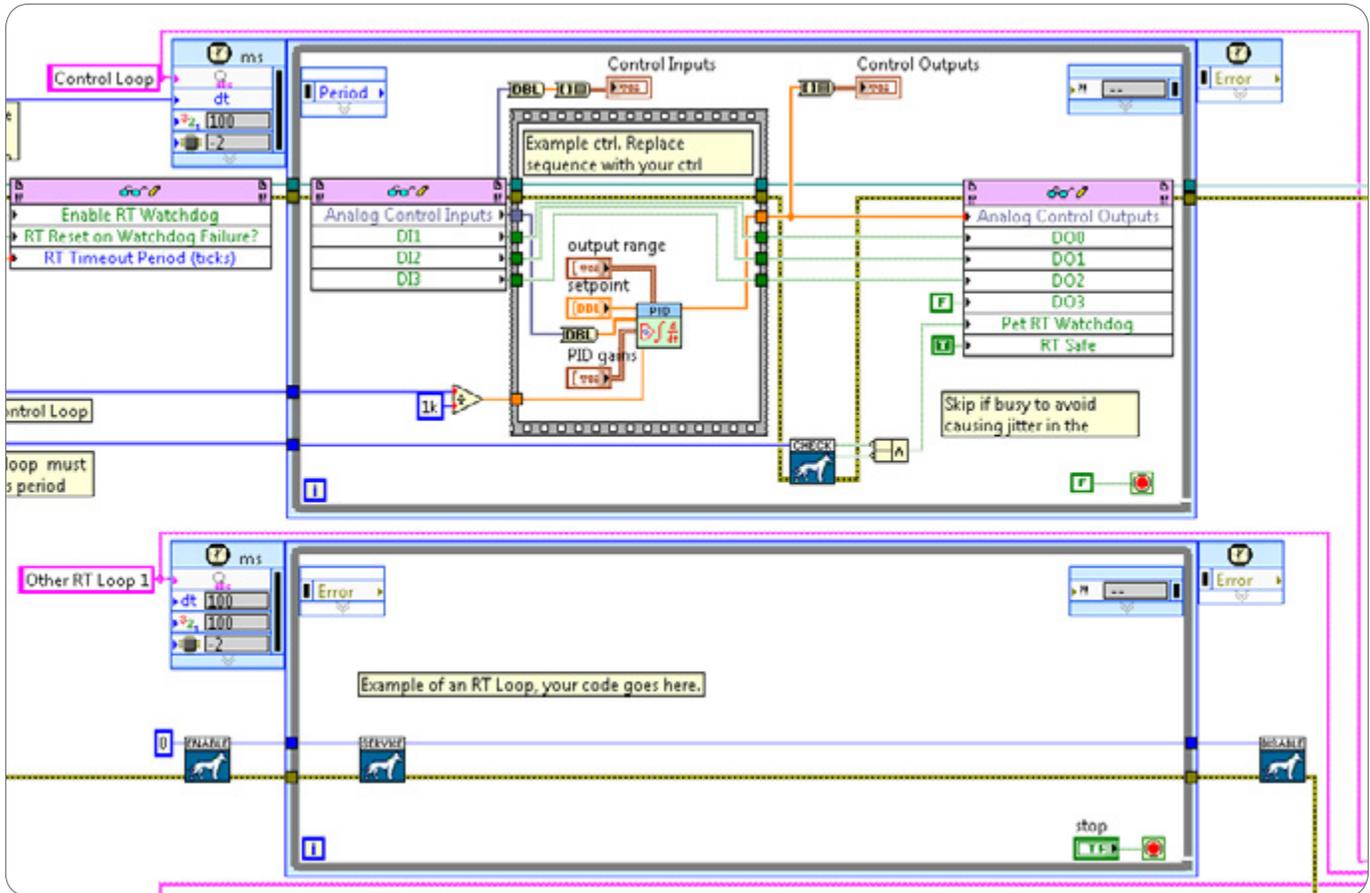


Figure 3.37. A software watchdog can take action if any loops become unresponsive and check in with the hardware watchdog in case something happens to the entire system.

CHAPTER 4

Best Practices for Network Communication

Network Communication

For embedded CompactRIO applications, communication with a remote client is often a critical part of the project. Embedded applications typically function as “data servers” because their primary role is to report information (status, acquired data, analyzed data, and so on) to the client. They are also usually capable of responding to commands from the client to perform application-specific activities.

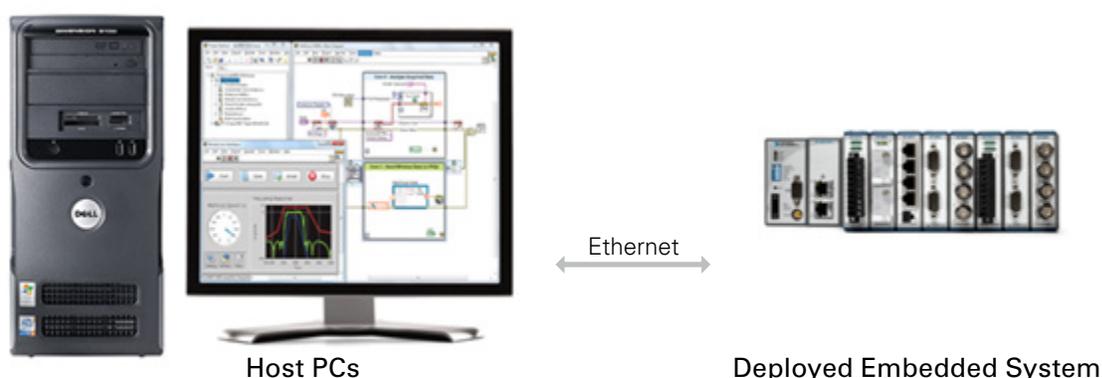


Figure 4.1. A CompactRIO system communicates to remote clients over Ethernet.

Selecting a Networking Protocol

You can choose from several data mechanisms in LabVIEW for one or more of the communication models listed in Table 4.1. When choosing a data transfer mechanism, consider the following factors.

Communication Model

The communication models discussed in chapters 1 and 3 pertain to both interprocess communication and network communication. These models are current value data (tags), updates, streaming, and command-/message-based communication. Understanding the type of data transfer you need is the most important consideration when choosing a networking mechanism. Table 4.1 provides examples of when you might use current value data (tags), updates, streaming, or command-based data communication for networked applications.

Communication Model	Description	Example
Current Value (tag)	The transfer of the latest values only. Typically distributed to multiple processes.	An embedded controller acquires temperature data and periodically transfers a data point to the host computer so users can monitor the state of the system.
Updates	The transfer of the latest values only when the latest value does not equal the previous value. Typically distributed to multiple processes.	A host PC is monitoring the heartbeat of multiple deployed CompactRIO systems and needs to be notified if one of the systems shuts down.
Streaming	The high-throughput transfer of every data point, typically from one process that transfers data from the FPGA to another process that saves the data to disk or sends it across the network.	An embedded controller acquires vibration data from a motor and streams every point of that data to a host computer for analysis and processing.
Message/Command	The low-latency data transfer from one process that triggers a specific event on another process. Command-based communication is typically infrequent and requires you to not miss any data points.	A user clicks the stop button in an HMI application, and the application shuts down a conveyer belt by sending the stop command to the CompactRIO target controlling the conveyer belt.

Table 4.1. Examples of Communication Model Use Cases

Network Configuration

The network configuration is another important consideration. In multiclient applications, clients may connect and disconnect at random times. The TCP/IP protocol requires you to implement a connection manager to dynamically accept and service any number of incoming connections in addition to tracking client requirements and servicing each client individually. While this is possible, it requires a decent amount of work on the development side. On the other hand, network-published shared variables feature the Shared Variable Engine with a built-in connection manager, which handles any incoming client connections for you. Since TCP/IP Messaging (STM) and CVT Client Communication (CCC) are both based on TCP, they also require you to develop a connection manager.

Network Streams also require some additional development when using them within a 1:N or N:1 configuration because they are implemented as a point-to-point communication model. This means that an application based on Network Streams that connects to 10 clients requires at least 10 streams. However, since Network Streams are the most reliable and easy-to-use mechanism for streaming data, they are still a good choice for streaming. For messages and commands, consider using shared variables over Network Streams.

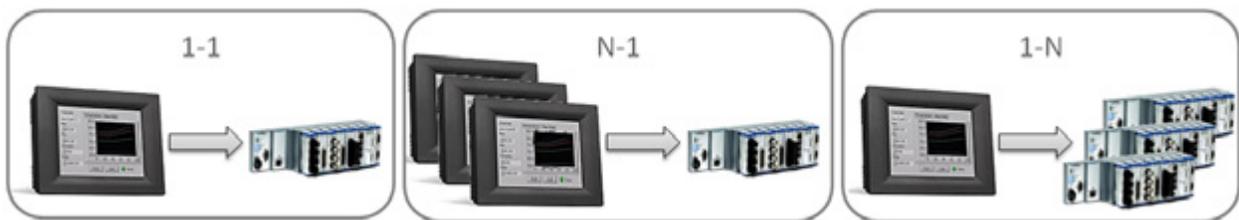


Figure 4.2. Consider your system configuration when choosing a networking protocol.

Interfacing to OSs and Third-Party Applications

Both Network Streams and network-published shared variables are based on proprietary protocols and offer limited support for communication to certain OSs and third-party applications, as shown in Table 4.2. Other mechanisms discussed later, including TCP/IP, UDP, STM, CCC, and web services, are based on standard protocols enabling them to transfer data to any OS or third-party application.

	Network Streams	Shared Variables
Supports data transfer to non-LabVIEW applications	No	NI LabWindows™/CVI and Measurement Studio only
Supports data transfer to Linux or Mac OS	No	Yes

Table 4.2. Network Stream and Shared Variable Support Limitations

LabVIEW Version

Several of the data transfer mechanisms recommended in Table 4.3 do not work in all versions of LabVIEW:

- Network Streams work in LabVIEW 2010 and later
- Web services work in LabVIEW 2009 and later
- Shared variables work in LabVIEW 8.0 and later

If you have selected one of these three mechanisms but are not using a compatible version of LabVIEW, use the mechanism recommended for the same communication model and network configuration that can communicate with a third-party UI.

Security

If you require security features such as built-in encryption and authentication, use HTTPS-based web services to implement communication. Web services support data communication to LabVIEW UIs, third-party UIs, and web browsers. If you are using the web server on NI's real-time targets to implement HTTPS communication, consider enabling the web services API keys for added security. To learn more about security with LabVIEW web services, refer to the *Configuring Web Services Security* LabVIEW Help document.

Ease of Use Versus Performance and Reliability

Your last consideration is the trade-off between ease of use and performance and reliability. If you are developing a quick prototype that you do not foresee using later in your mission-critical application, then you can use drag-and-drop network-published shared variables for most of your communication needs. However, these variables incur more CPU and memory overhead than other mechanisms, which may not be acceptable for high-performance or high-reliability applications. TCP/IP, Simple TCP/IP Messaging (STM), CVT Client Communication (CCC), UDP, and web services are based on standard protocols and offer both high performance and reliability when implemented correctly. Network Streams, while based on a proprietary protocol, are optimized for performance and have an enhanced connection management that automatically restores network connectivity if a disconnection occurs because of a network outage or other system failure.

Table 4.3 lists general recommendations for choosing a data transfer mechanism based on the three factors discussed previously: communication model, network configuration, and the type of application to which you are sending data. Other factors discussed in this section should be taken into account, but this table provides a starting point.

Network Configuration	Message or Command	Update	Stream	Current Value (Tag)
1:1 (CompactRIO to LabVIEW UI)	Network Streams (flushed)	Shared Variable (blocking)	Network Streams	Network-Published Shared Variable or CCC
1:1 (CompactRIO to Third-Party UI)	Simple TCP/IP Messaging (STM)	UDP	TCP/IP	Web Services or CCC
N:1 or 1:N (CompactRIO to LabVIEW UI)	Network Streams (flushed)	Shared Variable (blocking)	Network Streams	Network-Published Shared Variable or CCC
N:1 or 1:N (CompactRIO to Third-Party UI)	Simple TCP/IP Messaging (STM)	UDP	TCP/IP	Modbus or Web Services
CompactRIO to Web Client	Web Services	Web Services	Web Services	Web Services

Table 4.3. Recommended Networking Protocols and When to Use Them

The next section explains how to implement each of the networking protocols described in Table 4.3 and features instructions for downloading and installing the protocols that are not included in LabVIEW.

Network-Published Shared Variables

One method for sharing tags across a network is network shared variables. The term *network variable* refers to a software item on the network that can communicate between programs, applications, remote computers, and hardware. Network shared variables are ideal for 1:N or N:1 setups since they have a built-in connection manager that manages incoming clients. With other mechanisms such as CCC, STM, and Network Streams, you must create your own connection manager. At the same time, you need to understand the trade-off between performance and ease of use. While network shared variables have a high ease-of-use factor, they do not perform as well as other mechanisms when it comes to throughput and latency.

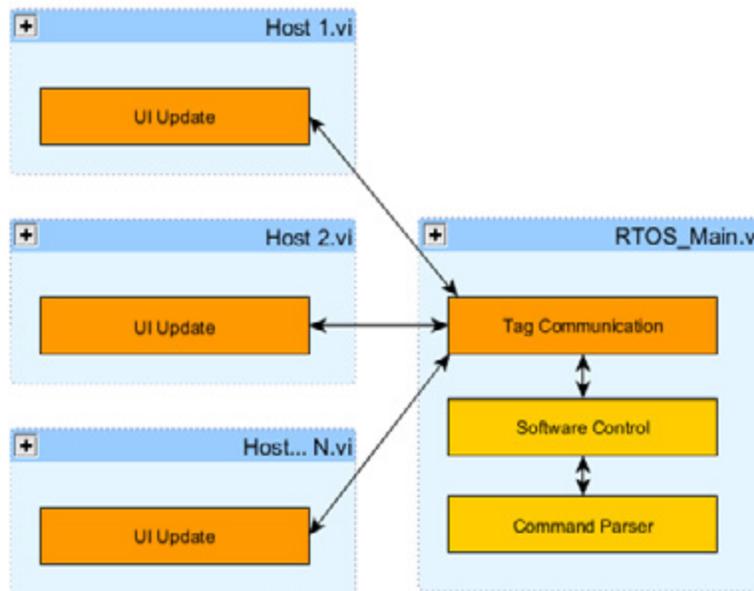


Figure 4.3. Network shared variables are ideal for tag communication in a 1:N or N:1 configuration.

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 4.4 shows a given network variable, its network path, and its respective item in the project tree.

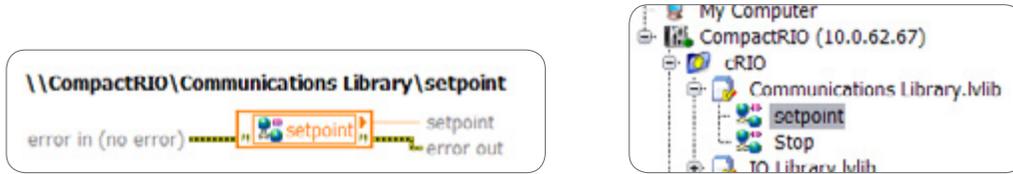


Figure 4.4. A Network Variable Node and Its Project Item

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 4.5 shows an example of a distributed system where the Shared Variable Engine runs on a desktop machine and where multiple real-time controllers exchange data through a network variable.

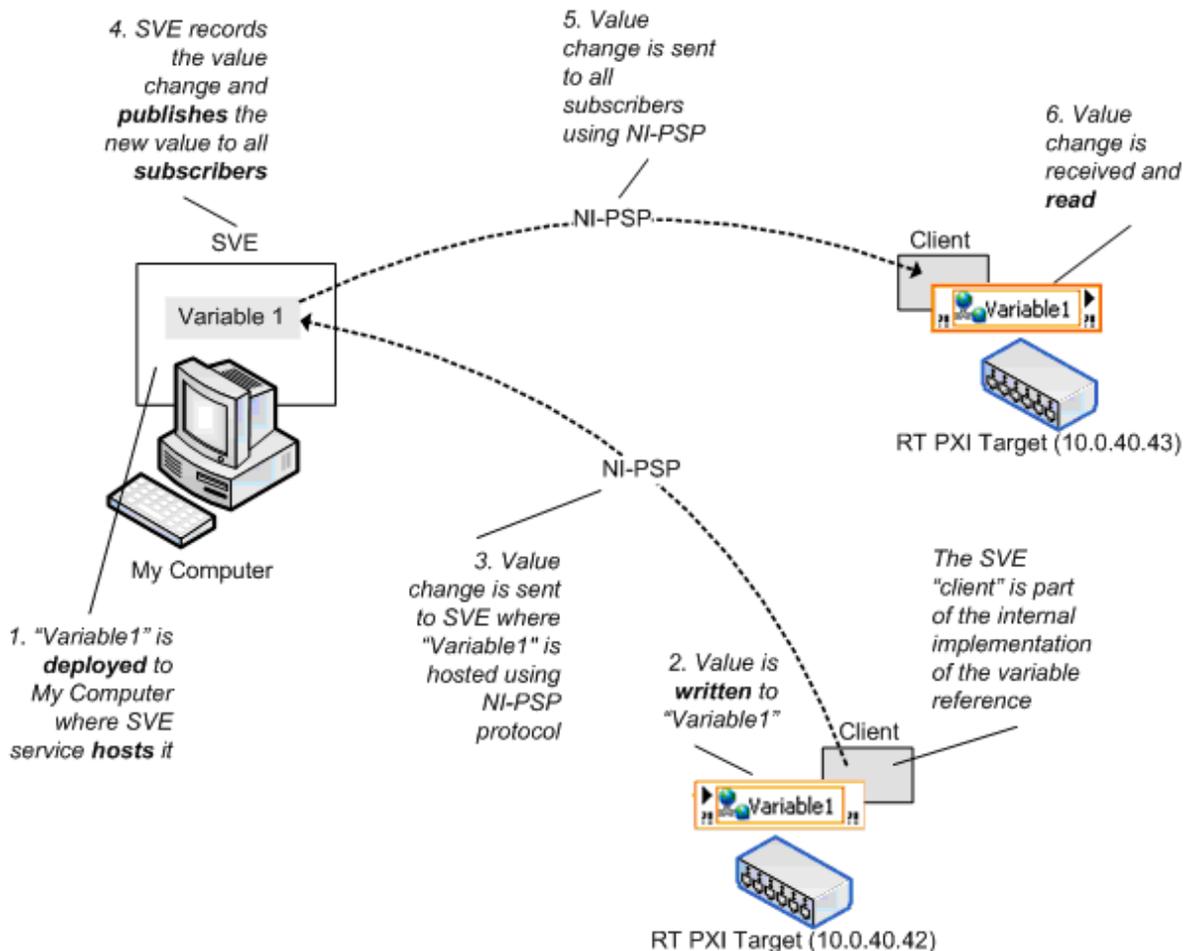


Figure 4.5. Distributed System Using a Network-Published Shared Variable to Communicate

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 that transmits data to subscribed clients only 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

Enabling the buffering option makes programming with shared variables significantly more complex, so disable this option for the majority of your applications. If you are interested in turning on shared variable buffering, first review the NI Developer Zone document [Buffered Network-Published Shared Variables: Components and Architecture](#). You can verify that buffering is disabled by right-clicking your shared variable node and launching the Shared Variable Properties dialog shown in Figure 4.6. By default, Use Buffering is turned off.

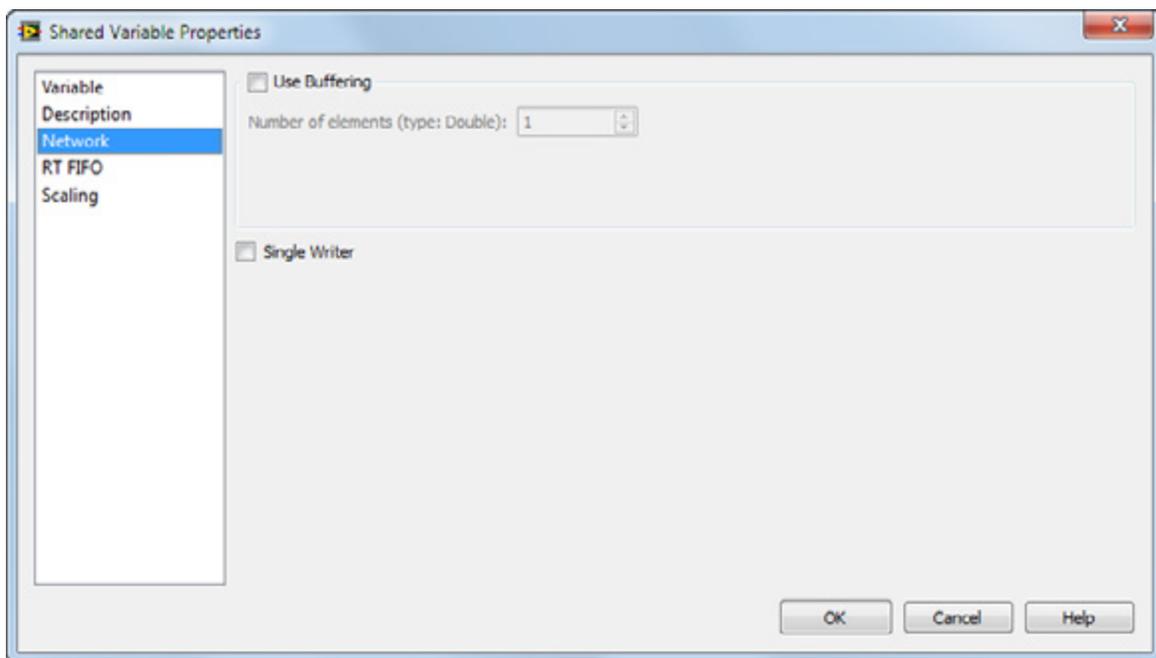


Figure 4.6. Ensure that buffering is disabled when using shared variables for tag communication.

Determinism

Network-published shared variables are extremely flexible and configurable. You can create a variable that features a real-time FIFO to include a network communication task within a time-critical loop. When you do this, LabVIEW automatically runs a background loop to copy the network data into a real-time FIFO, as shown in Figure 4.7. Keep in mind that this prevents jitter from occurring within the time-critical loop while performing network communication, but it does not mean that the network communication itself is deterministic.

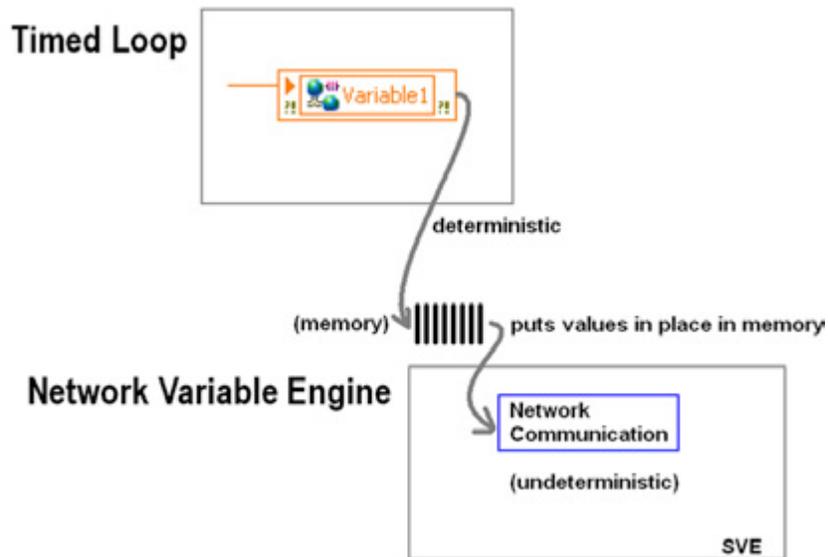


Figure 4.7. When you enable the real-time FIFO for network-published shared variables, a hidden background loop runs on the real-time target to copy the network value into a real-time FIFO.

This feature can simplify your program, but it has some limitations:

- Some capabilities of network-published shared 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

Another option for applications that involve both network communication and a time-critical loop is to use regular network-published shared variables for network communications and maintain a separate loop for network communication tasks. You can communicate between these two loops using the interprocess communication mechanisms discussed in [Chapter 3: Designing a LabVIEW Real-Time Application](#).

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 Scan Engine 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 NI-PSP protocol. They are published by a normal priority thread associated with the 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.

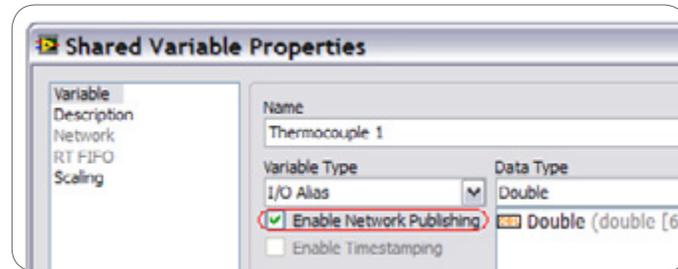


Figure 4.8. Enabling Network Publishing for an I/O Variable

Published I/O variables are optimized for I/O monitoring. They do not work with all network-published shared variable features and 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 CompactRIO system, so for large distributed applications, NI recommends dedicating a system to running the Shared Variable Engine.

Required features

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

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 the LabVIEW development environment.

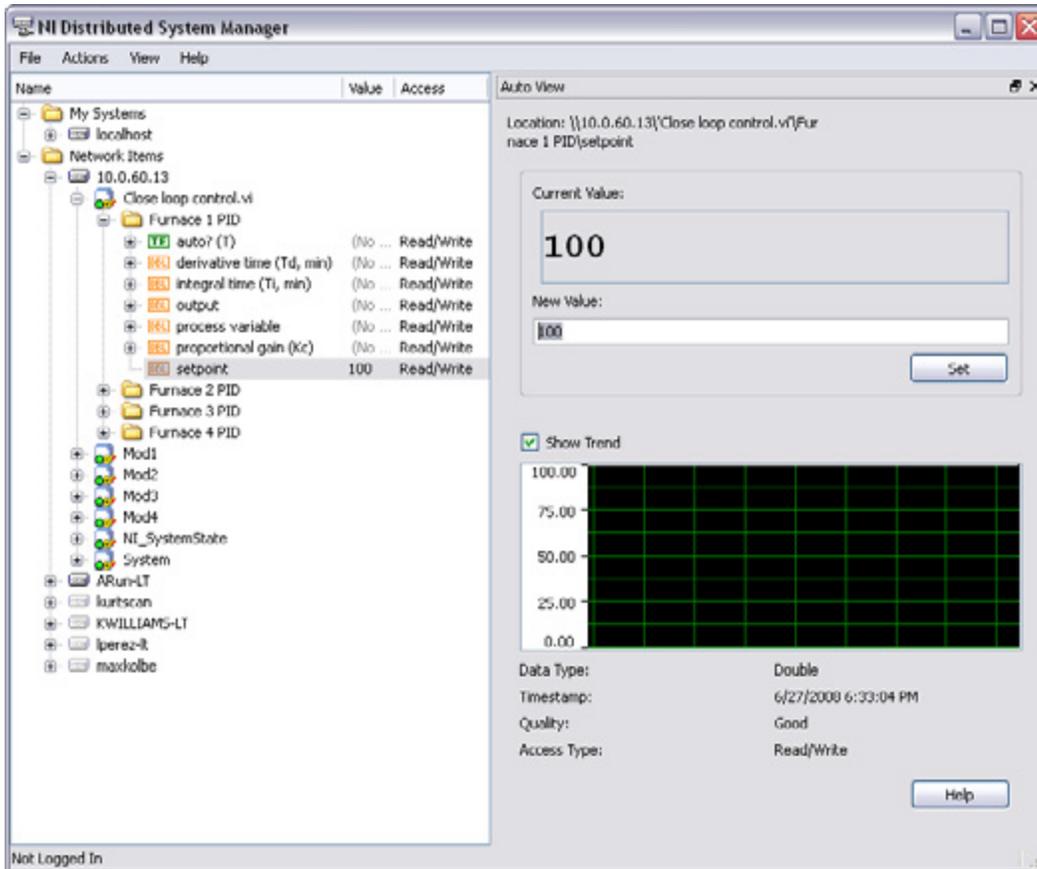


Figure 4.10. NI Distributed System Manager

With the NI Distributed System Manager, you can write to network-published shared 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.

Tips for Using Network Shared Variables Effectively

When programming with network shared variables, you can follow three tips to maximize performance and avoid any unwanted behavior. Figure 4.11 shows an initialization process that incorporates each tip.

Tip 1: Initialize Shared Variables

Initialize shared variables to known values at the beginning of your application. If the shared variable is not initialized, the first couple of iterations may output incorrect data or throw an error. After initializing, you may experience a small delay when loading the Shared Variable Engine after your application starts.

Tip 2: Serialize Shared Variable Execution

Serialize the execution of network shared variable nodes using the error wire to maximize performance. When executing shared variable nodes in parallel, thread swaps may occur and impact performance. Shared variable nodes that are serialized execute faster than when implemented in parallel.

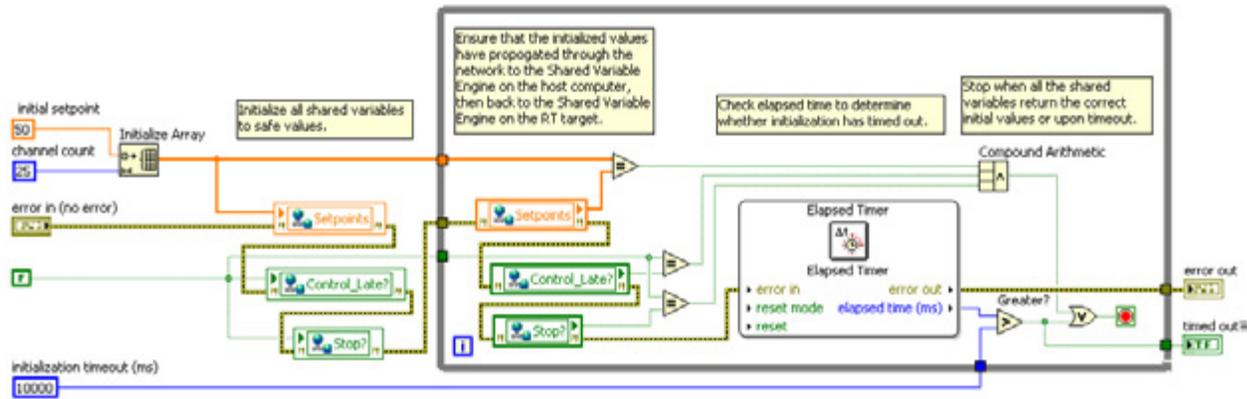


Figure 4.11. Initialize variables to known values and serialize variable execution.

There are also times when you may not want to serialize your variables. When multiple variables are serialized and an error occurs within the first variable, the variables further down the chain do not execute. If you want to ensure that every variable is processed, even if an error occurs within one variable, you should avoid serializing them.

Tip 3: Avoid Reading Stale Shared Variable Data

To prevent reading the same value repeatedly in a loop, use the **ms timeout** input of a shared variable node or the Read Variable with Timeout function. To add an **ms timeout** input to the shared variable node, right-click the shared variable node and select **Show Timeout** from the shortcut menu.

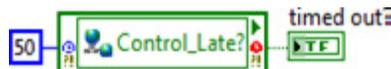


Figure 4.12. Use a timeout to prevent reading the same value repeatedly in a loop.

Network Streams

Network Streams are similar to Queue functions in that they are FIFO based, but unlike Queue functions, Network Streams have network scope. They were designed and optimized for lossless, high-throughput data communication over Ethernet, and they feature enhanced connection management that automatically restores network connectivity if a disconnection occurs due to a network outage or other system failure. Streams use a buffered, lossless communication strategy that ensures data written to the stream is never lost, even in environments that have intermittent network connectivity.

Because Network Streams were built with throughput characteristics comparable to those of raw TCP, they are ideal for high-throughput applications for which the programmer wants to avoid TCP complexity. You also can use streams for lossless, low-throughput communication such as sending and receiving commands.

Network Streams use a one-way, point-to-point buffered communication model to transmit data between applications. This means that one of the endpoints is the writer of data and the other is the reader. Your application might require multiple streams to communicate multiple types of data, as shown in Figure 4.13.

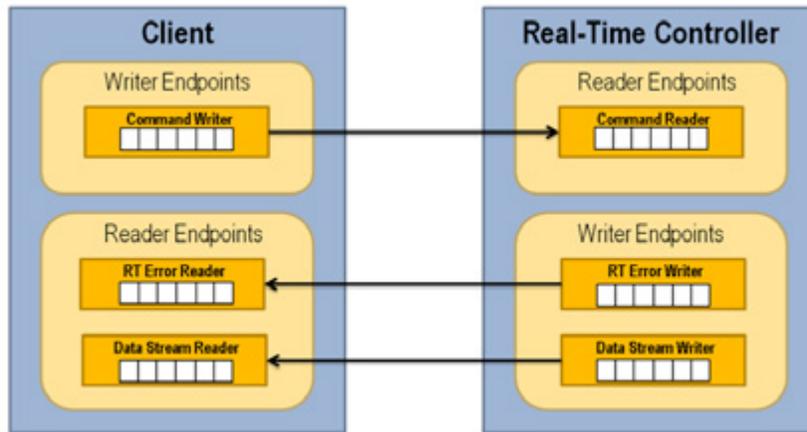


Figure 4.13. Network Streams use a one-way, point-to-point buffered communication model.

Figure 4.13 shows a basic Network Streams implementation. One process executes on the real-time target, and the other process executes on the host computer. You can follow three basic steps to set up a Network Stream:

1. Create the endpoints and establish the stream connection
2. Read or write data
3. Destroy the endpoints

Create the Endpoints

Connections are established once compatible reader and writer endpoints are created. The order of creating endpoints is not important. At least one endpoint must be active for the stream to attempt a reconnect.

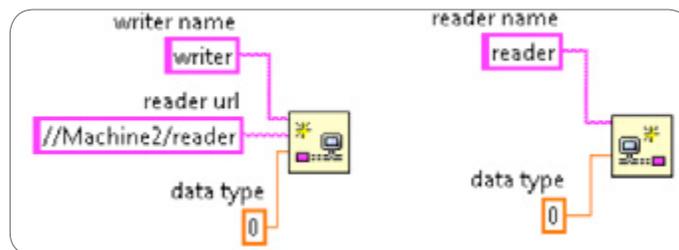


Figure 4.14. Create a reader and writer endpoint.

If a disconnection occurs and one endpoint becomes inactive, a reconnection is performed automatically by the protocol in the background. This protocol retries forever, preserving the lossless nature of the data stream. While the protocol is trying to reconnect, the active endpoint outputs an error message notifying the user that the endpoints cannot resynchronize. You can also monitor whether the stream is connected by using the Network Stream Endpoint Property Node shown in Figure 4.15.

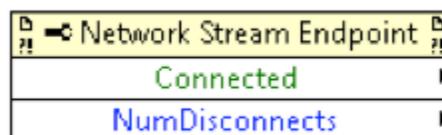


Figure 4.15. Monitor the Network Stream Endpoint connection status with a property node.

Read or Write Data

When reading from or writing to a Network Stream, you can write single elements or multiple elements at a time depending on which Network Stream function you choose. Data is never overwritten or regenerated, and no partial data transfers occur. The read/write functions either succeed or time out.

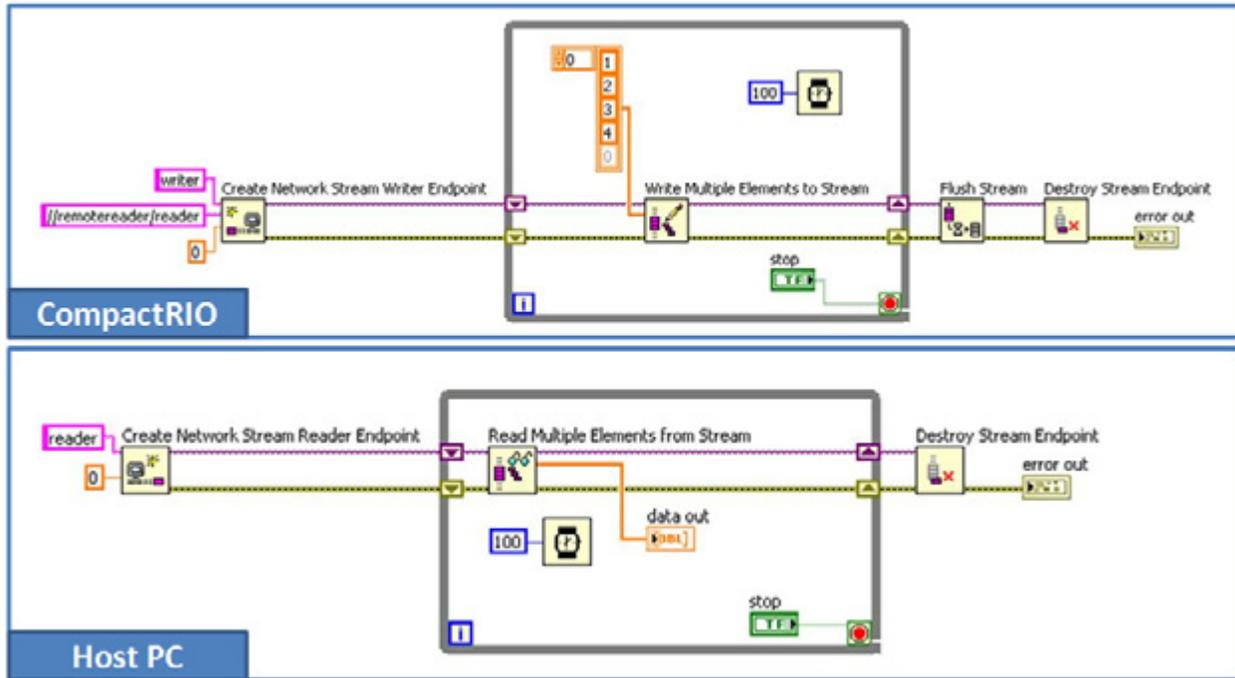


Figure 4.16. Basic Implementation of Network Streams to Stream Data Across a Network

Destroy the Endpoints

Before destroying your Network Stream Endpoints, you can use the Flush Stream.vi to ensure that all data has been transferred from the writer endpoint. If you select “All Elements Read from Stream,” the writer endpoint does not destroy itself until all elements from the writer buffer have been placed into and have been read from the reader buffer. If you select “All Elements Available for Reading,” the writer endpoint destroys itself as soon as all elements are placed into the reader buffer.

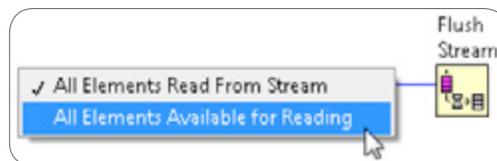


Figure 4.17. Use the Flush Stream.vi to shut down a Network Stream.

Using Network Streams to Send Messages and Commands

By default Network Streams are designed to maximize throughput, but you can easily implement them to maximize low latency for sending commands or messages.



LabVIEW example code
is provided for this section.

Command Sender Architecture

The command sender is any source of commands that the CompactRIO controller must react to. A common commander is a UI Event Handler that is part of an HMI and that translates UI events into commands for the CompactRIO 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.

In the next example, a host VI running on a desktop PC displays acquired data while allowing the user to adjust the frequency of the data as well as the window type for the Power Spectrum fast Fourier transform (FFT) analysis. The frequency and window commands are sent to the CompactRIO controller and processed in real time.

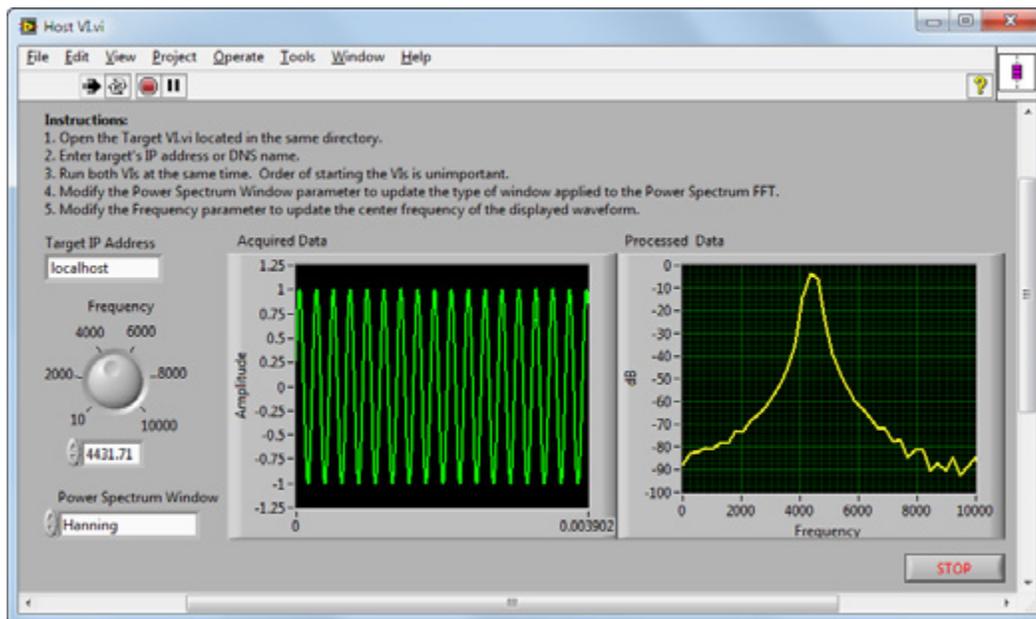


Figure 4.18. Front Panel of Example Host VI

When using Network Streams to send commands, do the following to ensure low latency:

1. Specify a small buffer size (the example uses a buffer size of 10 elements)
2. Use the Flush Stream VI to flush the buffer immediately after a command is sent

You can implement the commander architecture for events that originate on a UI by using the standard LabVIEW UI-handling templates. Translate the UI event into the appropriate command by building the command message and writing it to the Network Stream. Be sure to flush the buffer after the command is sent to ensure low latency.

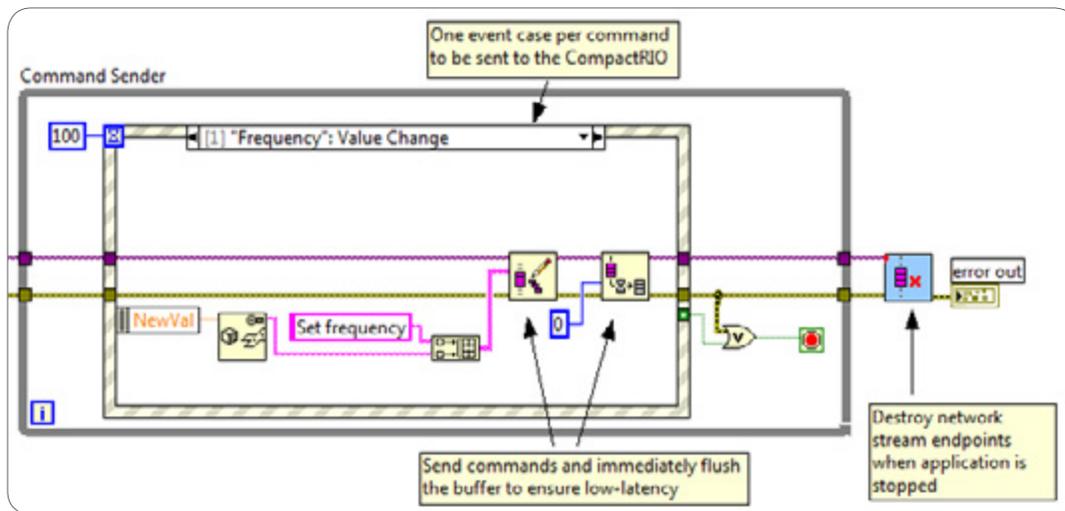


Figure 4.19. Command Sender Architecture Example

Command Parser Architecture

On the CompactRIO real-time target, you can use a command parser process to parse incoming commands and distribute them through the real-time application. In this example, RT FIFO functions are used for streaming data between the Acquire Data subVI and the Send Data subVI. Always use RT FIFO functions for streaming data between processes on a real-time target whether or not a time-critical loop is involved. Since RT FIFOs are required for streaming, they are also used for distributing the incoming commands.

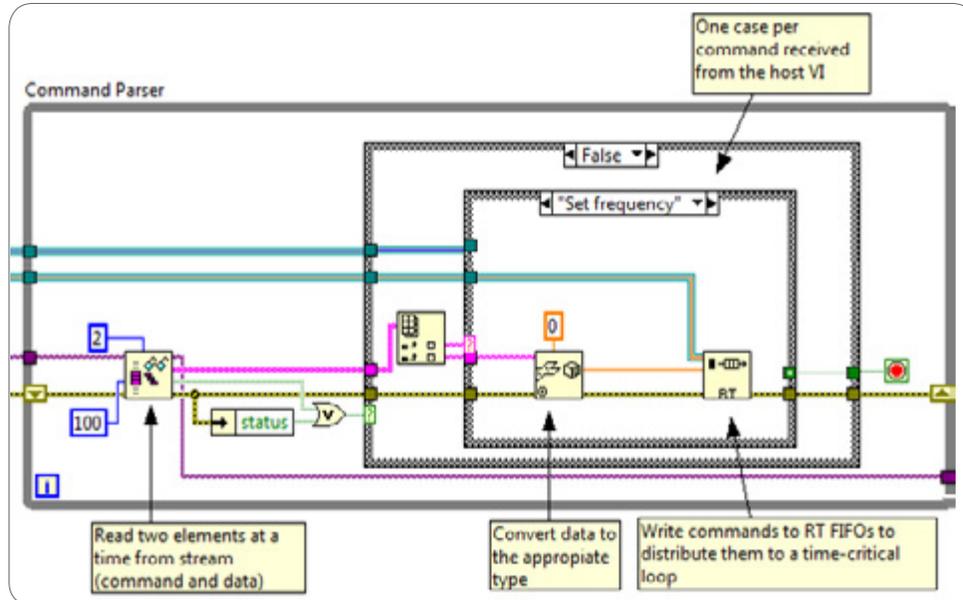


Figure 4.20. Command Parser Architecture Example

Installing Network Stream Support to CompactRIO

When you develop applications for LabVIEW Real-Time targets, keep in mind that Network Stream support is part of the LabVIEW Real-Time Module installation on the development computer. To enable Network Streams with LabVIEW Real-Time targets, select the Network Streams feature checkbox while installing software to the real-time target from MAX as shown in Figure 4.21.

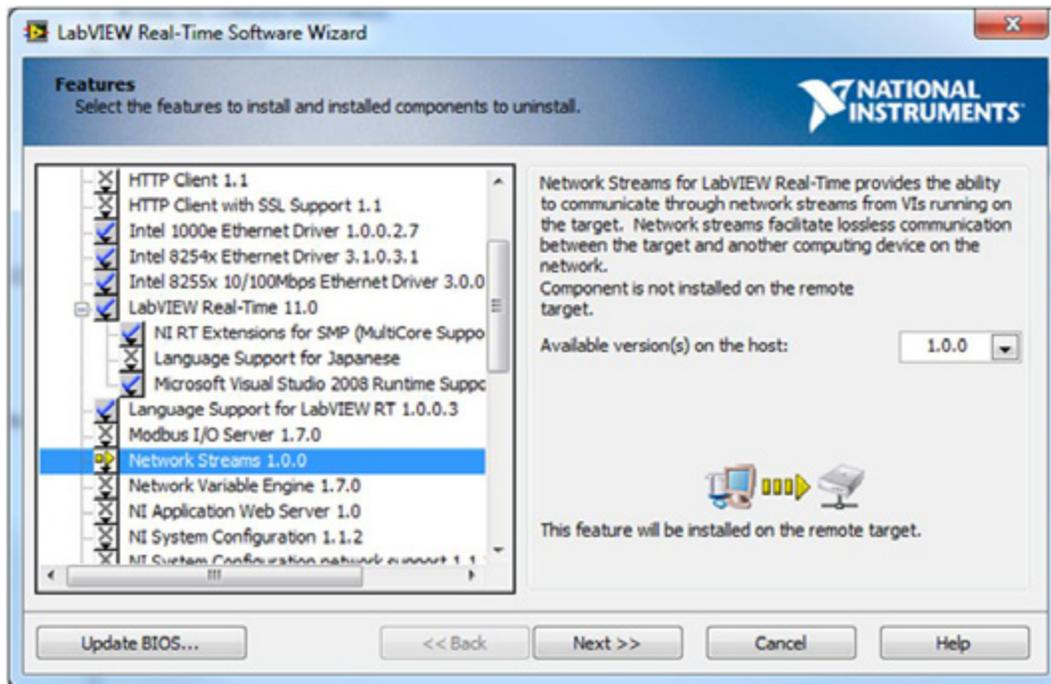


Figure 4.21. Install Network Stream Support on the Real-Time Target

You can find more information on Network Streams in the NI Developer Zone white paper [Lossless Communication with Network Streams: Components, Architecture, and Performance](#).

Raw Ethernet (TCP/UDP)

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 LabVIEW. They offer lower-level communication functions that are more flexible but less user friendly. You must handle functions such as establishing connections and packaging data at the application level.

TCP and UDP are good options if you need very low-level control of the communications protocol or if you are designing a custom protocol. They are also recommended for streaming data to third-party applications since Network Streams support communication only to LabVIEW applications. For sending messages to a third-party application, STM is easier to use and offers equivalent if not better performance. For sending current values or tags to a third-party application, CCC, web services, or Modbus, depending on your system configuration, also have easier implementations.

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 determine 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.

Simple TCP/IP Messaging (STM)

STM is a networking protocol that NI systems engineers designed based on TCP/IP. It is recommended for sending commands or messages across the network if you are communicating to a third-party API or require a standard protocol. It makes data manipulation more manageable through the use of formatted packets, and improves throughput by minimizing the transmission of repetitive data.



LabVIEW example code is provided for this section.

Download: You can download and install the STM library from the NI Developer Zone white paper [LabVIEW Simple Messaging Reference Library \(STM\)](#). The STM library is under the User Libraries palette.

Metadata

Metadata is implemented as an array of clusters. Each array element contains the data properties necessary to package and decode one variable value. Even if you have defined only the Name property, 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 4.22 shows an example of the metadata cluster configured for two variables: Iteration and RandomData.



Figure 4.22. 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 4.23 shows the packet format.



Figure 4.23. Packet Format

The metadata ID field is populated with the index of the metadata 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.

API

The STM API is shown in Figure 4.24. For basic operation, it consists of a Read VI and a Write VI. It also features two supplemental VIs to help with the transmission of the metadata, but their use is not mandatory. Each of the main VIs is polymorphic, which means you can use them with different transport layers. This document discusses STM communication based on the TCP/IP protocol, but STM also works with UDP and serial as transport layers. The API for each layer is similar.

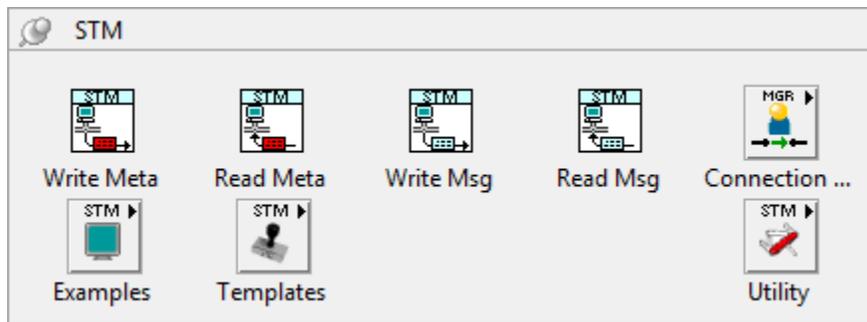


Figure 4.24. STM Functions

STM Write Meta

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

STM Read Meta

Receive metadata information from a remote computer with this VI. It reads and unpacks the metadata array, which you can pass to the Read and Write VIs.

STM Write Msg

Send any type of data to a remote host with this VI. It 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 the 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 data must be in string format for transmission. Use the Flatten to String function to convert the message data to a string.

STM Read Msg

Receive any type of data from a remote host with this VI. It reads and unpacks the metadata 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. In the example below, the variable named "RandomData" is always converted to an "Array of Doubles" data type.

This VI is usually used inside a loop. Since there is no guarantee that data will arrive at a given time, use the "timeout" parameter to allow the loop to run periodically and use the "Timed Out?" indicator to know whether to process the returned values.

Example

Figure 4.25 shows a basic example of STM being used to send RandomData and Iteration data across a network. The server VI is shown in Figure 4.25, and the client VI is shown in Figure 4.26. Notice that the server VI sends the metadata, implemented as an array of strings, 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.

You need to wire only the variable name to the STM Write Message VI, which takes care of creating and sending the message packet for you. Because of this abstraction, you can send data by name while hiding the underlying complexity of the TCP/IP protocol.

Also note that the application flattens the data to a string before it is sent. For simple data types it is possible to use a typecast, which is slightly faster than the Flatten to String function. However, the Flatten to String function also works with complex data types such as clusters and waveforms.

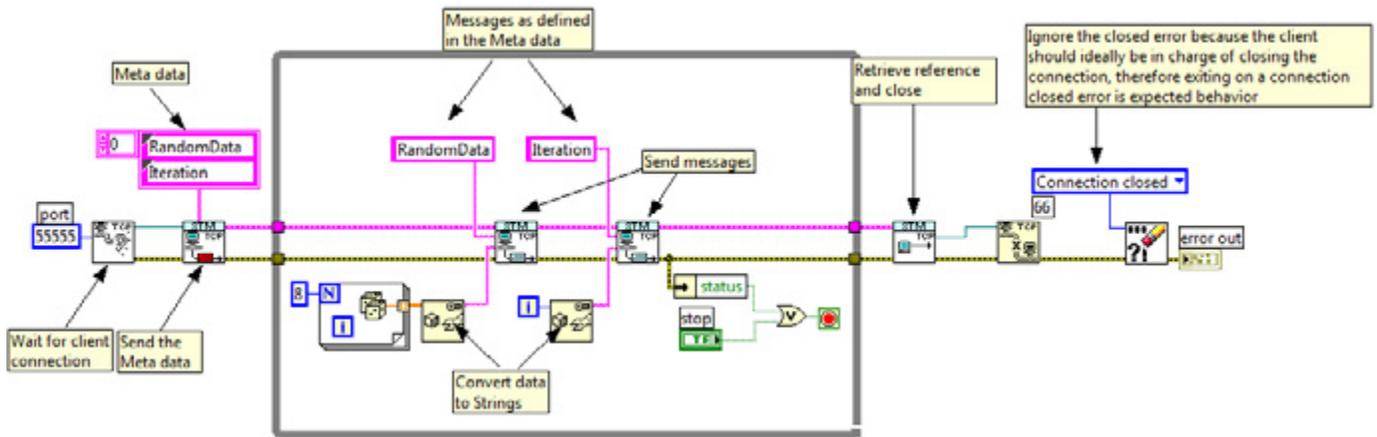


Figure 4.25. RTTarget VI Using STM Communication to Send Data to a Client

As you can see, you can customize the protocol and expand it to fit your application requirements. As you add variables, simply add an entry to the metadata array and a corresponding STM Write Message VI for that variable.

Receiving data is also simple. The design pattern shown in Figure 4.26 waits for the metadata when the connection is established with the server. It then uses the STM Read Message VI to wait for incoming messages. When it receives a message, it converts the data and assigns it to a local value according to the metadata name.

The Case structure driven by the data name provides an expandable method for handling data conversion. As you add variables, simply create a case with the code to convert the variable to the right type and send it to the right destination.

Note that an outer Case structure handles timeout events.

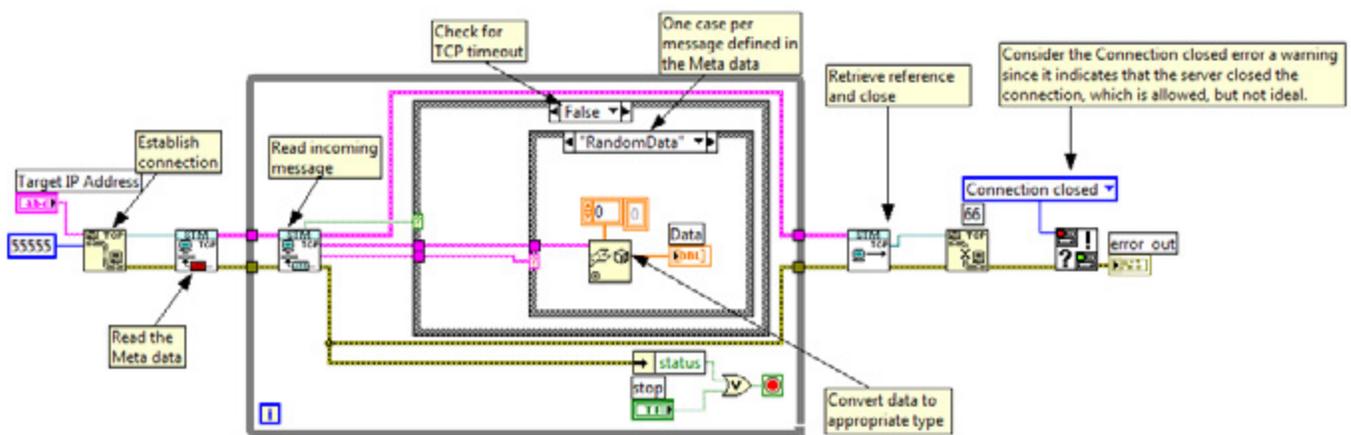


Figure 4.26. Host VI Using STM Communication to Read Incoming Data

One advantage of this design pattern is that it centralizes the code that receives data and assigns it to local values.

Another advantage is that the STM Read Message VI sleeps until it receives data (or a timeout occurs), so the loop is driven at the rate of incoming data. This guarantees that no data is lost and no CPU time is wasted polling for incoming data.

Note: Since the client has no knowledge of the metadata until run time, you must be sure that the application handles all possible incoming variables. It is a good practice to implement a “default” case to trap any “unknown” variables as an error condition.

For more information on STM, view the following white papers on ni.com:

[LabVIEW Simple Messaging Reference Library \(STM\)](#)

[Command-Based Communication Using Simple TCP/IP Messaging](#)

[A Multiclient Server Design Pattern Using Simple TCP/IP Messaging](#)

Dealing With Orphaned Sockets When Using TCP/IP

The inability to reestablish the listening TCP socket is one of the most common challenges when designing a TCP/IP-based application. This symptom is caused by an orphaned socket that occurs after you terminate the client or server application. If you design your code following the techniques described in this section, you can avoid this issue. This section provides an overview of orphaned sockets and a recommended design pattern when using TCP/IP for network communication.

Orphaned Socket Overview

With a TCP/IP-based application, you have a client/server scheme where the server listens on a particular port to which a client opens a connection. After the connection has been established, the client and server exchange data until the connection is terminated. A termination may be caused by the following:

- Intentional termination (the user stops the server or client application)
- Hardware becomes disconnected or goes down
- Client or server application crashes

If you attempt to resume the connection immediately after termination, you see an error dialog with error code 60 from the TCP Listen.vi that is similar to the error dialog in Figure 4.27. This inability to reestablish the listening TCP socket is caused by an orphaned socket. The orphaned socket is the socket that initiated the termination. If you wait for 60 seconds before attempting to reestablish the connection, the error message goes away. However, many systems cannot afford to be down for 60 seconds.

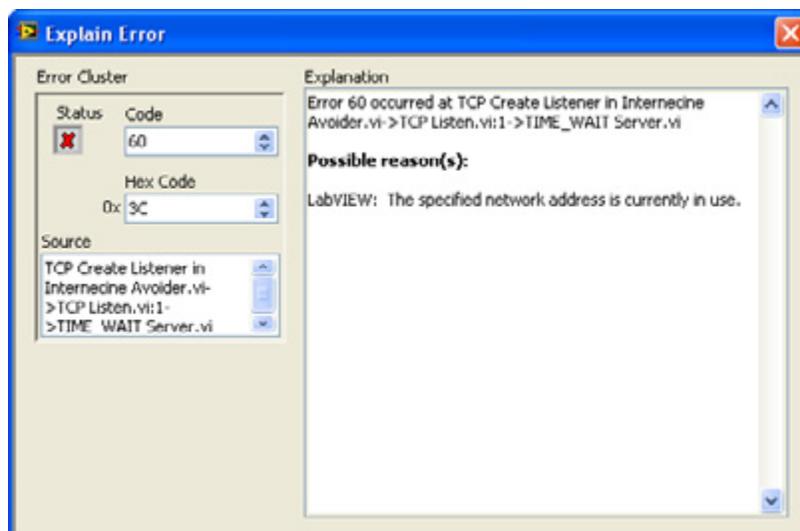


Figure 4.27. Error Code 60 is generated when the client/server connection is terminated.

The 60 second timeout is intentional. When an orphaned socket is identified, TCP/IP makes the socket unavailable for 60 seconds so that no other sockets can communicate with it. If you compare TCP/IP to a postal service, a termination is equivalent to a family moving out of its home. The postal service temporarily takes down that mailbox so that if new people move in, they do not receive mail that does not belong to them. TCP/IP makes the socket unavailable intentionally so that it can send data reliably over the network.

Preventing Orphaned Sockets

This section provides two methods for preventing orphaned sockets.

Design your application so that only the client can terminate the connection

In most cases, the orphaned socket issue is more severe on the server side. Typically, the client port is assigned dynamically but the server port is fixed. When the server closes the connection, the server port becomes locked out. If the connection is always terminated from the client side, you can significantly reduce the risk of dealing with error 60.

When designing your server application, you need to follow three rules:

1. Do not close the connection in the event of a timeout by ignoring the timeout error.
2. If you want to stop the server application, send a message to the client and let the client terminate the connection. Wait for a non-timeout error (62 or 56) and close the server application upon error.
3. Do not close the server application upon an event. If an event occurs that should stop the application, send a message to the client and have the client terminate the connection. Then close the server application upon error.

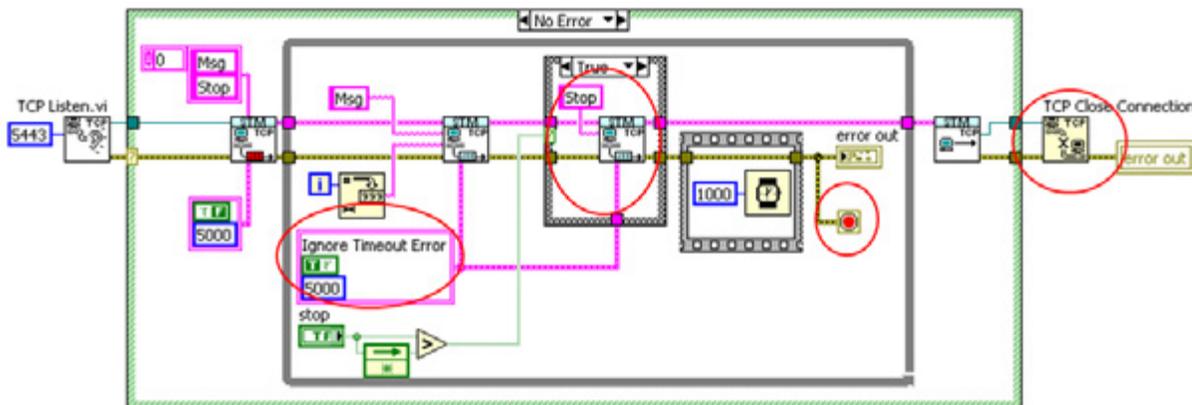


Figure 4.28. This example VI prevents the server from closing the network connection.

Assign Dynamic Ports

Another method for avoiding orphaned sockets is to assign a dynamic port on both the client and server. You have two ways to do this: manually using UDP to establish the port (if you have the STM reference library installed, see the STM Distributed Clients example as a starting point) or using the NI Service Locator. To use the NI Service Locator, you must have LabVIEW 8.5 or later, and the NI Service Locator must be running. When using the NI Service Locator, you pass a service name into the **service name** input on both the TCP Open Connection and TCP Create Listener functions.

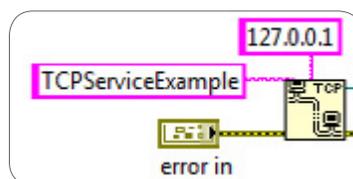


Figure 4.29. Use the NI Service Locator to ensure dynamic port assignment on both the client and server.

CVT Client Communication (CCC)

Consider using CCC if you are using the CVT discussed in Chapter 3 for interprocess communication. If you have already created CVT tags and you want to publish this data on the network, CCC is a simple and elegant solution. It is based on TCP/IP and works best for 1:1 system configurations. If you are working with a 1:N or N:1 system configuration, consider binding your CVT tags to network-published shared variables when implementing network communication.

The primary function of the client communication interface is to share information between CVT instances on the server (CompactRIO) and the client. Do this by mirroring portions of the CVT from one side to the other and vice versa.

Instructions for Installing the CCC Library

Step 1: Navigate to the NI Developer Zone white paper [CVT Client Communication \(CCC\) Reference Library](#).

Step 2: Follow the instructions under the Downloads section. The CCC library appears under the User Libraries palette.



LabVIEW example code is provided for this section.

Implementation

The underlying implementation of CCC is TCP/IP. Specifically, it is an adaptation of STM, which offers a platform-independent way of sending messages by name while maintaining the performance and throughput of raw TCP communication. In applications involving hundreds or perhaps thousands of tags, the importance of efficient communication is obvious.

The CCC interface is made up of two separate elements. The server portion of the interface acts as a TCP server and consists of a command parser that processes requests for data from the client. The client portion of the interface acts as the TCP client and initiates the communication with the server. It then sends commands to the server for configuring and sending/receiving data.

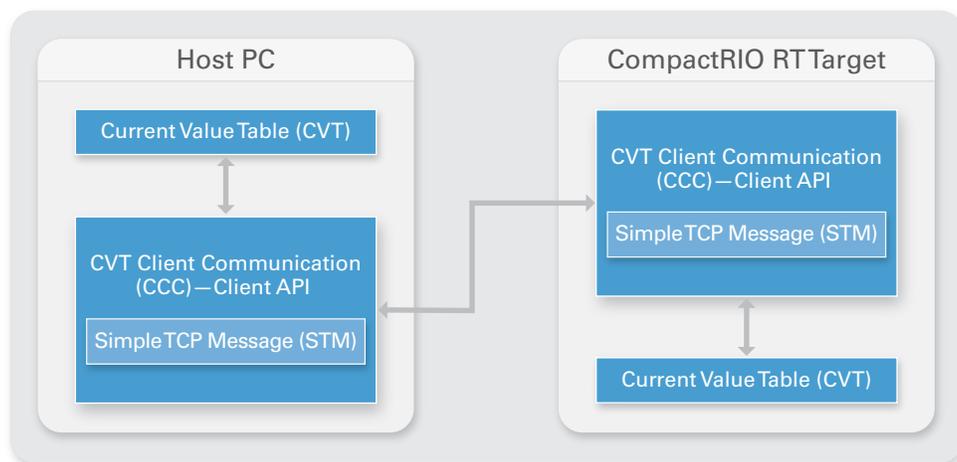


Figure 4.30. CCC Overview

The CCC protocol implementation emphasizes performance optimization by configuring as much as possible on the first call, leaving the repetitive operations with less work to do. Accordingly, the protocol is implemented in such a way that the client must first identify all of the tags of interest using the Bound Address parameter. On the first execution, the server looks up the tags by their index in the CVT. From that point forward, only the CVT index API is used to ensure the highest possible performance.

On both the client and server components, all of the repetitive operations are implemented with determinism in mind. They allocate all of the necessary resources on the first call of each function and use a functional global variable to store blocks of data between iterations. This ensures that no memory allocation takes place after the first iteration.

API and Example

The CCC server is a VI that is designed to run in parallel with the rest of the application. This allows the rest of the machine control to execute asynchronously, which provides better control over the application timing. The server API features functions for starting and stopping the CCC server. You need to initialize the CVT before calling the CCC Start Server.vi.

In most cases, you can use both the server and client elements of the interface as drop-in components. The server needs only the TCP port configured (default is 54444), and the client needs the server's IP address and port number. Figure 4.31 shows an example of a CCC server application that includes the following steps:

1. Initialize the server-side CVT
2. Initialize the CCC server process, which executes asynchronously from the rest of the application
3. Use the CVT API functions (tags) to read and write data to and from the server-side CVT
4. Stop the CCC server process

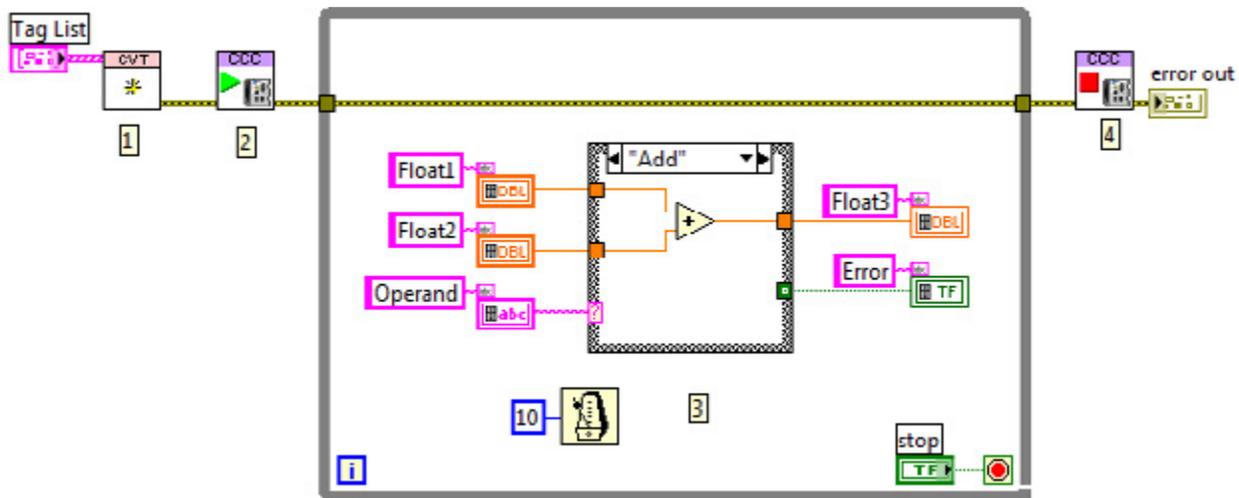


Figure 4.31. CCC Server Example—Static Tag List

In the corresponding client application, shown in Figure 4.32, the CCC write and read operations are implemented in series with the rest of the HMI code. This ensures that the values for both the read and write tags are updated on each iteration. The client application includes the following steps:

1. Initialize the client-side CVT
2. Initiate a connection with the server
3. Use the CVT API functions (tags) to read and write data to and from the client-side CVT
4. Use the CCC Client Read and Write VIs to transfer data between the client-side CVT and the server-side CVT
5. End a connection with the server

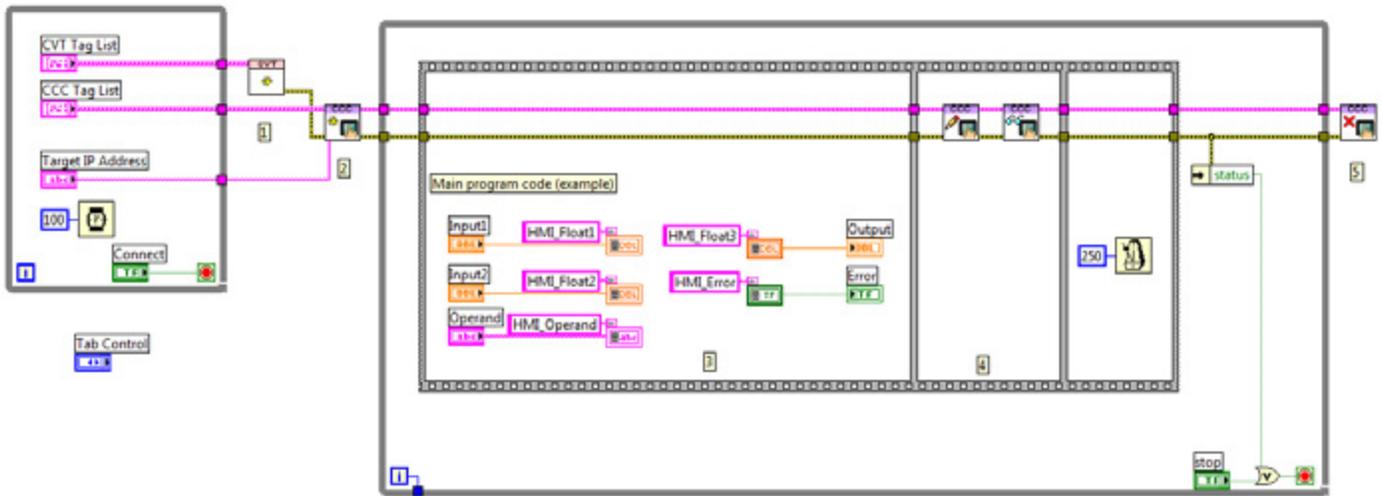


Figure 4.32. CCC Client Example—Static Tag List

For more information on CCC, see the NI Developer Zone white paper [CVT Client Communication \(CCC\) Reference Library](#).

Web Services

LabVIEW web services, introduced in LabVIEW 8.6, offer an open and standard way to communicate with VIs over the web. Consider a LabVIEW application deployed across a distributed system. LabVIEW provides features such as Network Streams for establishing communication, but many developers need a way to communicate with these applications from devices that do not have LabVIEW using standard web-based communication. With LabVIEW web services, you can:

- Communicate with embedded LabVIEW applications from any web-enabled device
- Establish machine-to-machine communication using standard HTTP protocols
- Remotely monitor and control LabVIEW applications using custom thin clients
- Stream any standard MIME data types, such as text, images, and videos
- Deploy web service VIs on a Windows or LabVIEW Real-Time target

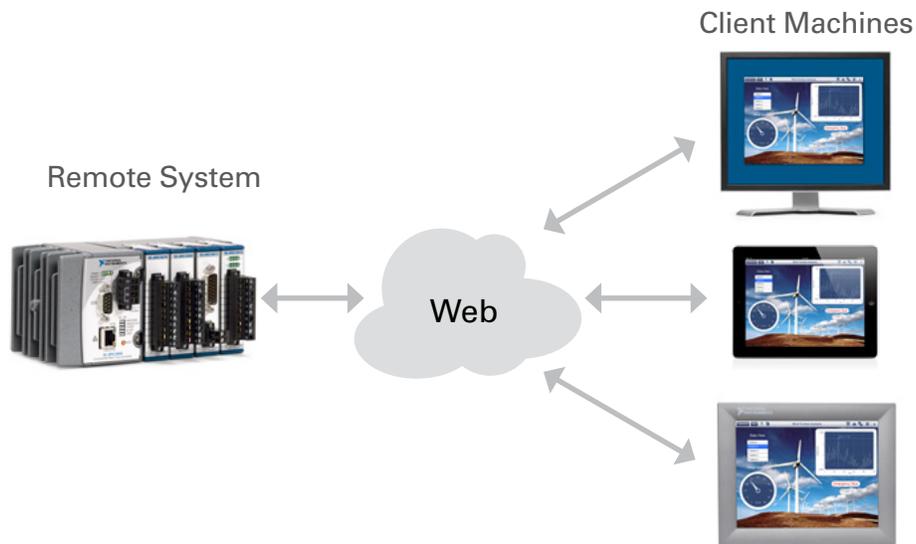


Figure 4.33. You can use web services to communicate data over the web.

Web services act as a web API to any type of software, whether that software is controlling a complex embedded system or simply a database store. To use a web service, a client sends a request to the remote system hosting the service, which then processes the request and sends back a response (typically an XML, or eXtensible Markup Language, message). The client can choose to display the raw XML data, but it is more common to parse the data and display it to the user as part of a GUI.

Using this approach, you can create one or more VIs for your CompactRIO LabVIEW Real-Time target and build them as web services. These web service VIs provide a standard interface for exchanging data between your embedded devices and any computer connected via a network.

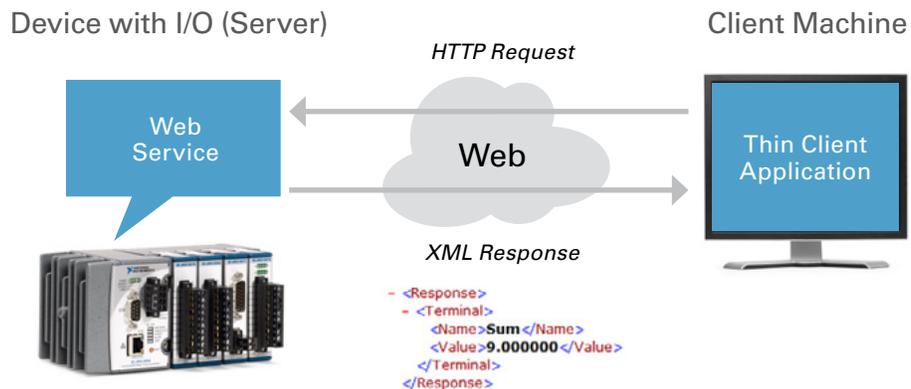


Figure 4.34. You can host and execute web services on a remote system and access them via the standard HTTP protocol.

Security Options With Web Services

You can use Secure Sockets Layer (SSL) encryption, user and group permissions, and API keys to establish secure communication between web clients and LabVIEW web services applications. For more information, read the LabVIEW Help document [Configuring Web Services Security \(Windows, ETS, VxWorks\)](#).

Adding Communication Mechanisms to Your Design Diagram

Once you have selected an appropriate mechanism for network communication, you can add this information to your design diagram. The diagram in Figure 4.35 represents the turbine testing application discussed in [Chapter 1: Designing a CompactRIO Software Architecture](#).

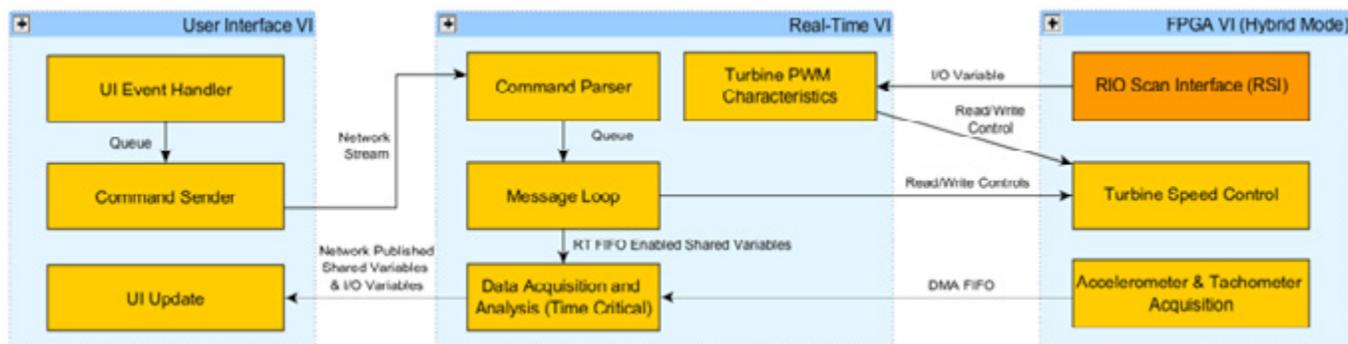


Figure 4.35. Add communication mechanisms to your design diagram.

The bioreactor application uses Network Streams to send commands from the host PC to the CompactRIO controller. Since this application uses the RIO Scan Interface to handle I/O, network-published I/O variables are used to send raw I/O data to the UI update process.

CHAPTER 5

Customizing Hardware Through LabVIEW FPGA

This chapter covers several best practices in addition to advanced tips and tricks for developing high-performance control and monitoring systems with the LabVIEW FPGA Module and CompactRIO. It examines recommended programming practices, ways to avoid common mistakes, and numerous methods to create fast, efficient, and reliable LabVIEW FPGA applications.

FPGA Technology

FPGAs offer a highly parallel and customizable platform that you can use to perform advanced processing and control tasks at hardware speeds. An FPGA is a programmable chip composed of three basic components: logic blocks, programmable interconnects, and I/O blocks.

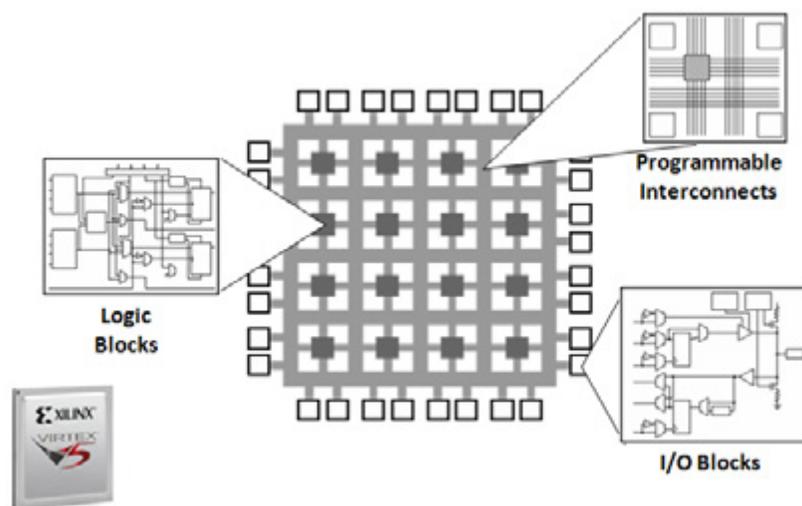


Figure 5.1. An FPGA is composed of configurable logic and I/O blocks tied together with programmable interconnects.

The logic blocks are a collection of digital components such as lookup tables, multipliers, and multiplexers where digital values and signals are processed to generate the desired logical output. These logic blocks are connected with programmable interconnects that route signals from one logic block to the next. The programmable interconnect can also route signals to the I/O blocks for two-way communication to surrounding circuitry. For more information on FPGA hardware components, see the [LabVIEW FPGA Help Document: Introduction to FPGA Hardware Concepts](#).

FPGAs are clocked at relatively lower rates than CPUs and GPUs, but they make up for this difference in clock rate by allowing you to create specialized circuitry that can perform multiple operations within a clock cycle. Combine this with their tight integration with I/O on NI reconfigurable I/O (RIO) devices and you can achieve much higher throughput and determinism as well as faster response times than with a processor-only solution. This helps you tackle high-speed streaming, digital signal processing (DSP), control, and digital protocol applications.

Because LabVIEW FPGA VIs are synthesized down to physical hardware, 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 up to several hours depending on how intricate your design is. Later in the chapter, learn more about debugging and simulating your FPGA VI so that you can compile less often.

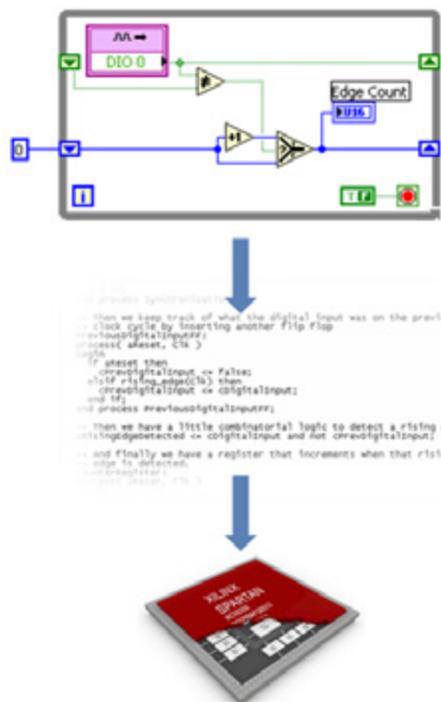


Figure 5.2. Behind the scenes, the LabVIEW FPGA compiler translates LabVIEW code into VHDL and invokes the Xilinx compile tools to generate a bitfile, which runs directly on the FPGA.

Establishing a Design Flow

Depending on the complexity of your LabVIEW FPGA application, you may want to quickly write a program and compile it down to hardware, or you may want to leverage the built-in simulator to debug, test, and verify your code without having to compile to hardware every time you make a change. This section of the guide outlines one example of a recommended design flow to enhance your productivity while programming in LabVIEW FPGA.

1. Establish functional and performance requirements (covered in Introduction and Basic Architectures)
2. Design a software architecture (covered in Introduction and Basic Architectures)
3. Implement LabVIEW FPGA code
4. Test and debug LabVIEW FPGA code
5. Optimize LabVIEW FPGA code
6. Compile LabVIEW FPGA code to hardware
7. Deploy your system

The next few sections reflect this recommended design flow, starting with implementing your LabVIEW FPGA code. The first two topics are discussed in the first section of the LabVIEW for CompactRIO Developer's Guide, Introduction and Basic Architectures.

Best Practices for Implementing LabVIEW FPGA Code

When you begin development, you should create any VIs underneath the FPGA target in the LabVIEW project so you can program with the LabVIEW FPGA palette, which is a subset of the LabVIEW palette plus some LabVIEW FPGA-specific functions.

You also should develop your VIs in simulation mode by right-clicking **FPGA Target** and selecting **Execute VI on»Development Computer with Simulated I/O**. By taking this approach, you can quickly iterate on your design and access all of the standard LabVIEW debugging features. If you need to access real-world I/O, change the execution mode to **Execute VI on»FPGA Target**.

Reading and Writing I/O

This section covers the basics of accessing I/O through a LabVIEW FPGA VI. For more detailed information on timing and synchronization with analog and digital I/O modules using LabVIEW FPGA, see [Chapter 6: Timing and Synchronization of I/O](#).

To develop a LabVIEW FPGA application, you need to add your FPGA target to a LabVIEW project in addition to any necessary I/O, clocks, register items, memory items, or FIFOs. The LabVIEW Help document [Using FPGA Targets in a LabVIEW Project](#) provides detailed instructions on how to set this up.

Once your LabVIEW project is configured to target an FPGA device, you can drag and drop an I/O channel from the LabVIEW project onto the LabVIEW FPGA VI block diagram to get an I/O node.

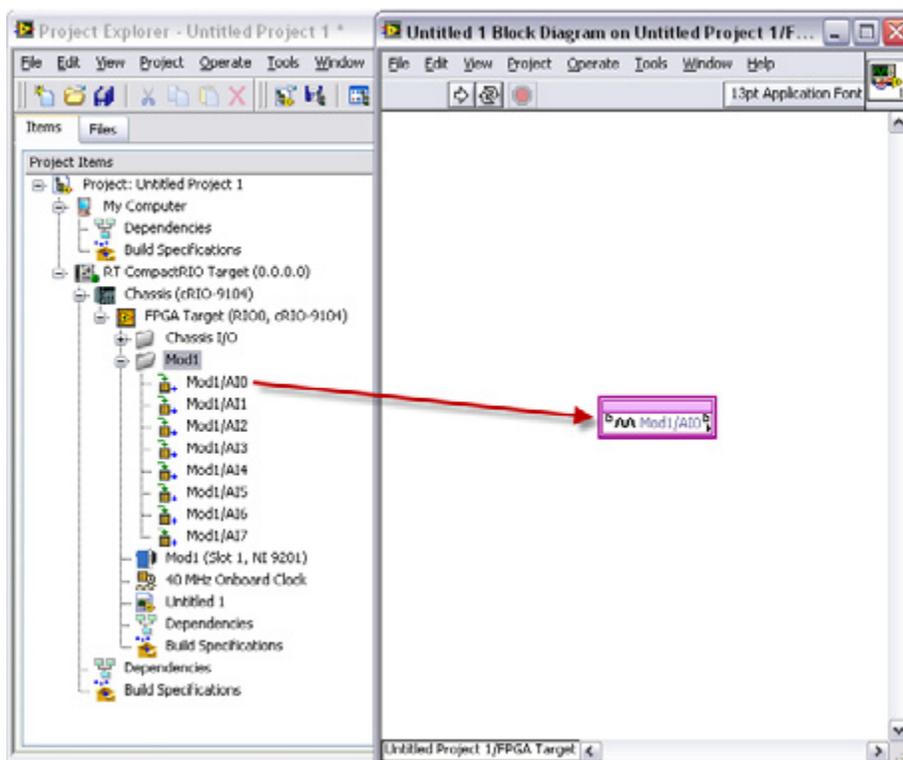


Figure 5.3. Drag and drop an I/O node to the FPGA block diagram.

The FPGA I/O Node returns single-point data when called. For most NI R Series devices and C Series I/O modules, you can use LabVIEW structures and logic to specify the sample rate and triggering. High-speed analog input modules that use delta-sigma converters and have their own onboard clock are an exception. For these modules, you control the sample rate with a Property Node. Table 6.2 in [Chapter 6: Timing and Synchronization of I/O](#) provides a list of NI C Series I/O modules that use delta-sigma modulation.

For most I/O nodes, you can use a loop with a Loop Timer Express VI to set the sample rate. The Sample Delay control sets the rate in ticks. Ticks are the pulses on the FPGA clock, which by default is 40 MHz for most CompactRIO targets. For example, implementing a sample delay of 20,000 ticks of the 40 MHz clock renders a sample 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 rate you specify, the module samples up to only the maximum rate specified in the module documentation.

You also have an option to use a Sequence structure, as shown in Figure 5.4. A Sequence structure can be used to guarantee that every time the LabVIEW FPGA VI is compiled, the timer and I/O Node are executed in the same sequence.

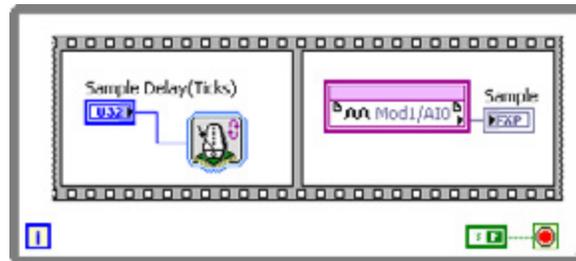


Figure 5.4. The Acquisition Scheme for a Standard Analog Input Module

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. With LabVIEW FPGA, you can implement basic or complex triggering schemes such as retriggering, pause triggering, or any type of custom triggering. Figure 5.5 shows an example of an application that takes in a start trigger and monitors the number of samples being collected until it reaches the number specified by the user. The code used in this example can be found in the Turbine Tester project, which can be downloaded from the [Introduction and Basic Architectures](http://ni.com/compactrio/devguide) section at ni.com/compactrio/devguide.

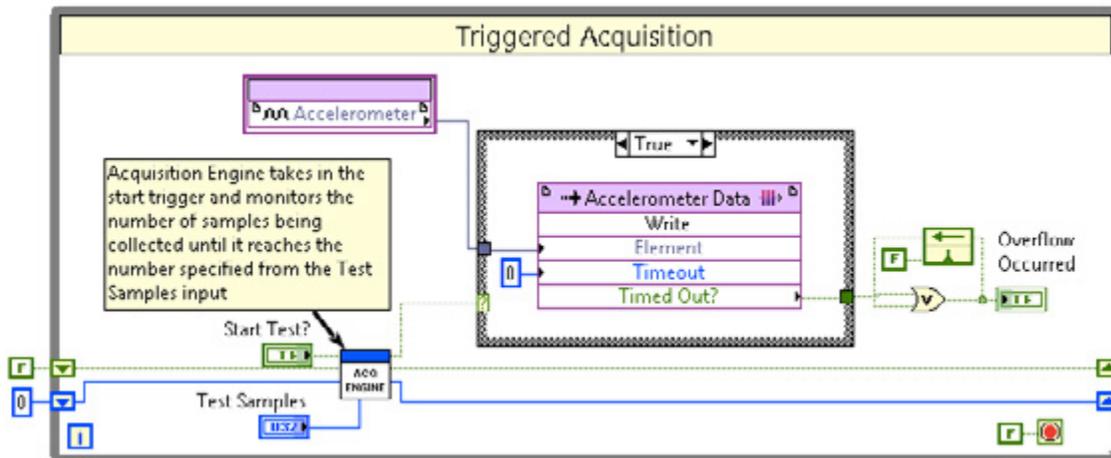


Figure 5.5. Simple One-Shot Trigger

Synchronize Your Loops

For most control and monitoring applications, the timing of when the code executes is important to the performance and reliability of the system. 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.

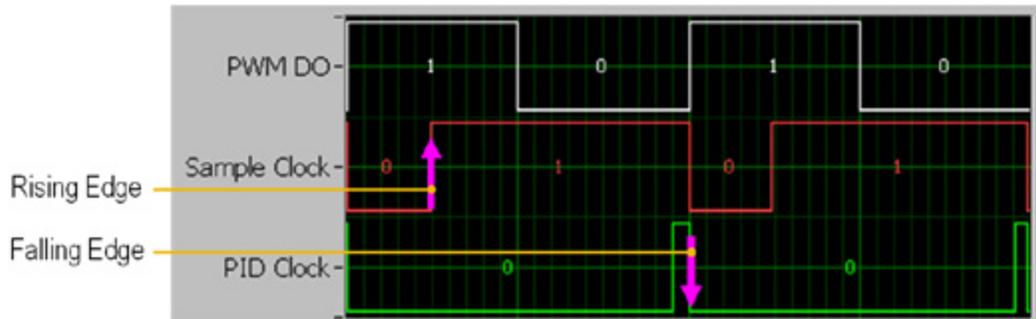


Figure 5.6. Motor Control Example With Two Different Clock Signals

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. This is often implemented with a Sequence structure that uses the first frame to wait for the trigger and the second frame to execute the triggered code, as shown in Figure 5.7.

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.)

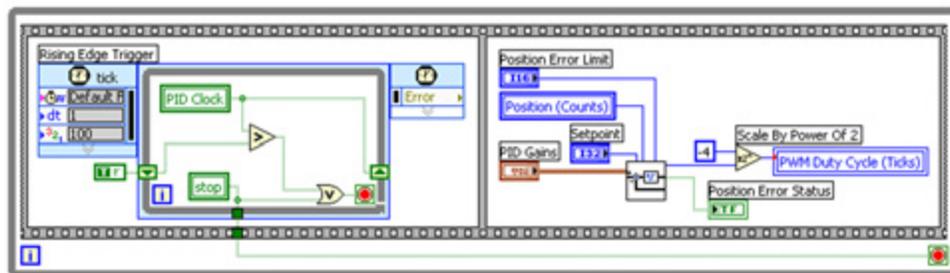


Figure 5.7. Rising Edge Trigger Example

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.)

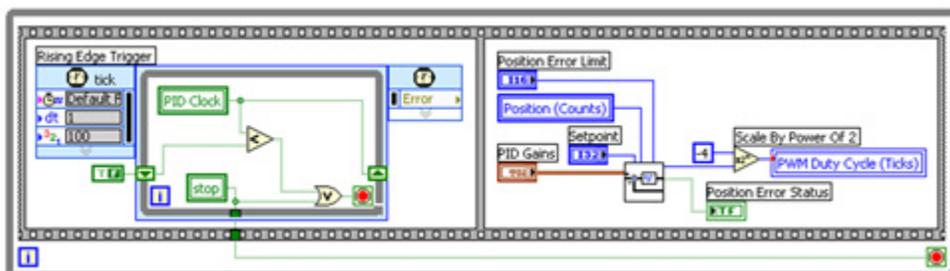


Figure 5.8. Falling Edge Trigger Example

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.

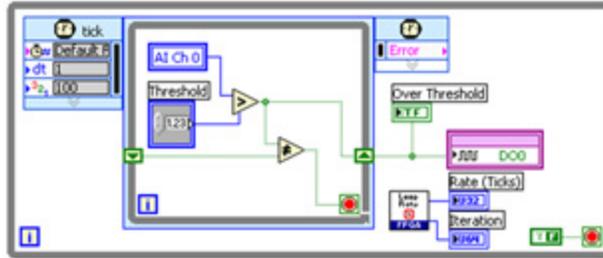


Figure 5.9. Analog Level Trigger Example

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.)

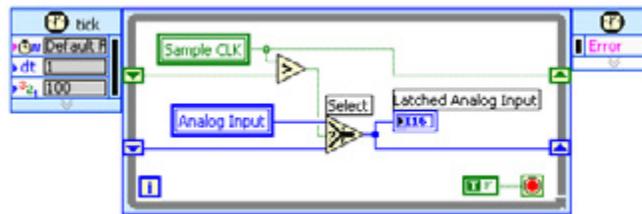


Figure 5.10. In this example, the actual analog input operation is occurring in another loop, and you are using a local variable for communication between the loops.

Creating Modular, Reusable SubVIs

Writing modular code is almost always a good idea, whether you are designing an application for a Windows, a real-time, or an FPGA device. SubVIs make code easier to debug and troubleshoot, easier to document and track changes, and typically cleaner, easier to understand, and more reusable. An example of a LabVIEW FPGA subVI is shown in Figure 5.11. This subVI counts the number of samples acquired once a trigger condition is met.

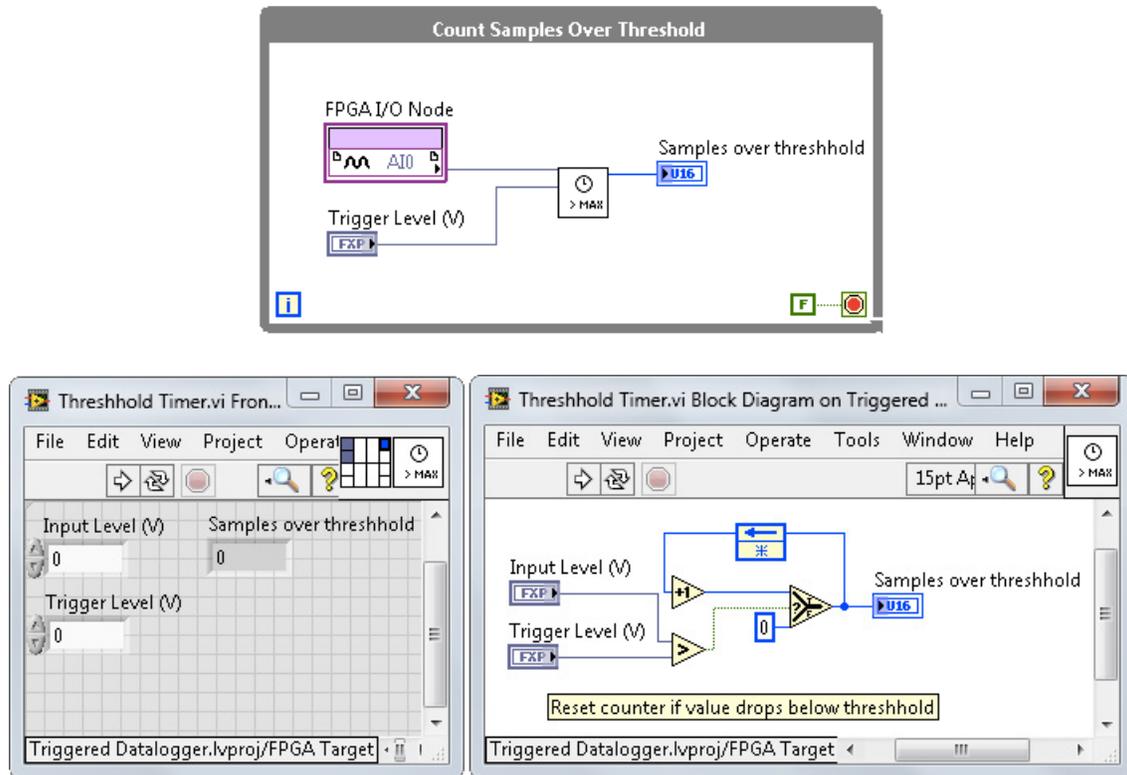


Figure 5.11. A subVI is used to count the number of samples acquired once a trigger condition is met.

Items to Avoid Placing Into SubVIs

When creating subVIs, you should consider keeping some items outside your subVI, specifically I/O nodes and Loop Timer or Wait functions.

Placing I/O nodes outside subVIs makes them more modular and portable and makes the top-level diagram more readable. This also reduces extraneous I/O node instances that might otherwise be included multiple times in the subVI, resulting in unnecessary gate usage. When accessing shared resources in LabVIEW FPGA, the compiler adds extra arbitration logic necessary to handle multiple callers. Arbitration is described in more detail on [page 86](#).

Another best practice is to avoid using Loop Timer or Wait functions within your modular subVIs. If the subVI has no delays, it executes as fast as possible and avoids slowing down the caller. Also, if you need to move your subVI into a single-cycle Timed Loop (SCTL) for optimization purposes, you must remove any delay functions since they are not supported.

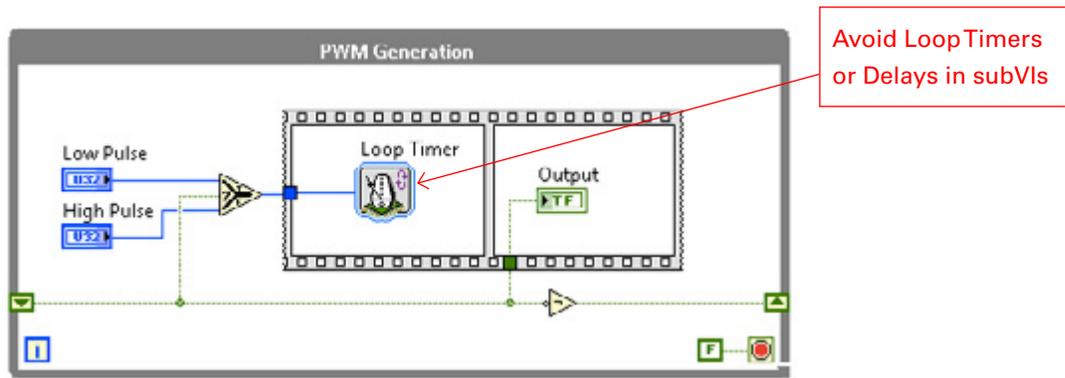


Figure 5.12. Avoid using Loop Timer or Wait functions within your modular subVIs.

The left side of Figure 5.13 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.

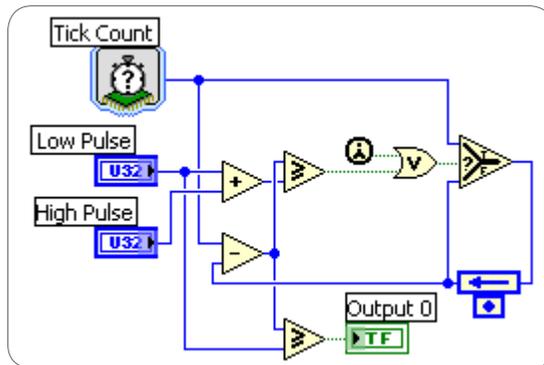


Figure 5.13. Adapt PWM code to use a Tick Count function rather than a Loop Timer function.

Understanding the Trade-Offs Between Reentrant and Nonreentrant SubVIs

Reentrancy is a setting in the subVI Execution properties. In LabVIEW FPGA, subVI execution is set to reentrant by default. Reentrant creates multiple copies of the subVI within the FPGA logic. This enables you to execute multiple copies of the subVI in parallel with distinct and separate data storage.

In the example in Figure 5.14, your subVI is set to reentrant, so both of these loops run simultaneously and any internal shift registers, local variables, or VI-scoped memory data points are unique to each instance.

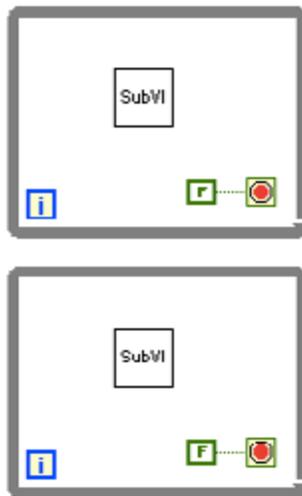


Figure 5.14. A subVI is set to reentrant, so all four of these loops run simultaneously, and any VI-scoped memory data, internal shift registers, or local variables are unique to each instance.

In the case of LabVIEW FPGA, this also means that each copy of the function uses its own FPGA slices, so reentrancy is effective for code portability but has the potential to use more gates. If you are trying to save FPGA resources, you may want to change the execution properties to nonreentrant execution. However, for small subVIs, you could potentially use more FPGA fabric in this case because the compiler adds more logic to the FPGA fabric to handle arbitration. The section below describes techniques for avoiding arbitration.

Avoiding Arbitration

Arbitration occurs when your LabVIEW FPGA code contains a shared resource, such as an I/O node or a nonreentrant subVI, that has multiple readers or writers. In these situations, the compiler adds more logic to the FPGA fabric to handle this arbitration. If you are concerned about minimizing the amount of FPGA fabric you are using, try to avoid arbitration. Find more information on how arbitration works in the [LabVIEW FPGA Help Document: Managing Shared Resources](#).

You can avoid arbitration by changing the arbitration settings within LabVIEW as described in the help document above, but a more reliable technique is designing your application in a way that prevents arbitration. Some of these techniques are listed below:

- Minimize shared resources
- Use double-sided resources in time-critical code (such as FIFOs)
- Avoid multiple readers and/or writers to double-sided resources

Another technique is to create multiple global or local variables when you are in a situation that requires multiple writers. For example, consider having a LabVIEW FPGA VI that uses one loop for reading from an analog input node and two additional loops for processing the analog data. In this situation, you can create two separate global or local variables, say A and B, to write to inside your analog input loop. In your first processing loop, you can read from variable A, and in your second processing loop, you can read from variable B.

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 5.15. 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.

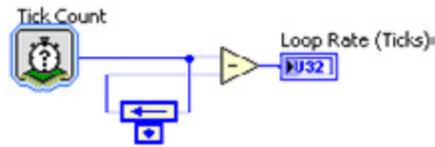


Figure 5.15. Use the Tick Count function to measure elapsed time.

Because you use 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.

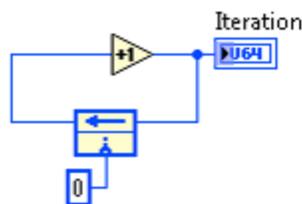


Figure 5.16. Unsigned integers are typically preferred for iteration counters because they give the largest range before rolling over.

Data Communication

Data communication in LabVIEW FPGA falls into two categories: interprocess and intertarget. Interprocess communication generally corresponds to sharing data between two or more loops on the FPGA target. Intertarget data communication is sharing data between the FPGA target and host processor. For both cases, you should consider whether you are communicating current value data, messages, or commands or are streaming data before deciding which mechanism to use. These data communication types are described in more detail in the [Introduction and Basic Architectures](#) chapter.

Interprocess Data Communication

If you have multiple loops on your FPGA target, you might want to share data between them like you would in a real-time or Windows-based program. LabVIEW FPGA includes several mechanisms for sharing data between loops.

Variables

LabVIEW FPGA features local and global variables for sharing current values or tags between two or more loops. With variables, you can access or store data in the flip-flops of the FPGA. Variables store only the latest data that is written. They are a good choice if you do not need to use every data value you acquire.

Memory Items

Another method for sharing the latest values is to use available memory items. Taking advantage of memory items consumes few logic blocks but uses the onboard memory. LabVIEW FPGA devices have two types of memory

items: target scoped and VI defined. You can access target-scoped memory items through all VIs under the FPGA target. VI-defined memory items are scoped to the VI that defines them. Figure 5.17 shows a block diagram using a Memory Method Node configured for a target-scoped memory item. This VI reads data from memory, increments the data, and then overwrites the same memory location with the new data.

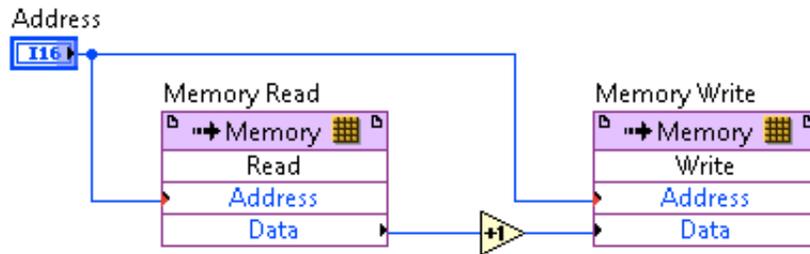


Figure 5.17. Read and write to a memory item using a Memory Method Node.

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. For more information on using memory items on an FPGA target, see the LabVIEW Help document [Storing Data on an FPGA Target](#).

FIFOs

If you are communicating messages or updates, or streaming data between two or more loops, consider using FIFOs for data transfer. A FIFO is a data structure that holds elements in the order they are received and provides access to those elements using a first-in-first-out access policy.

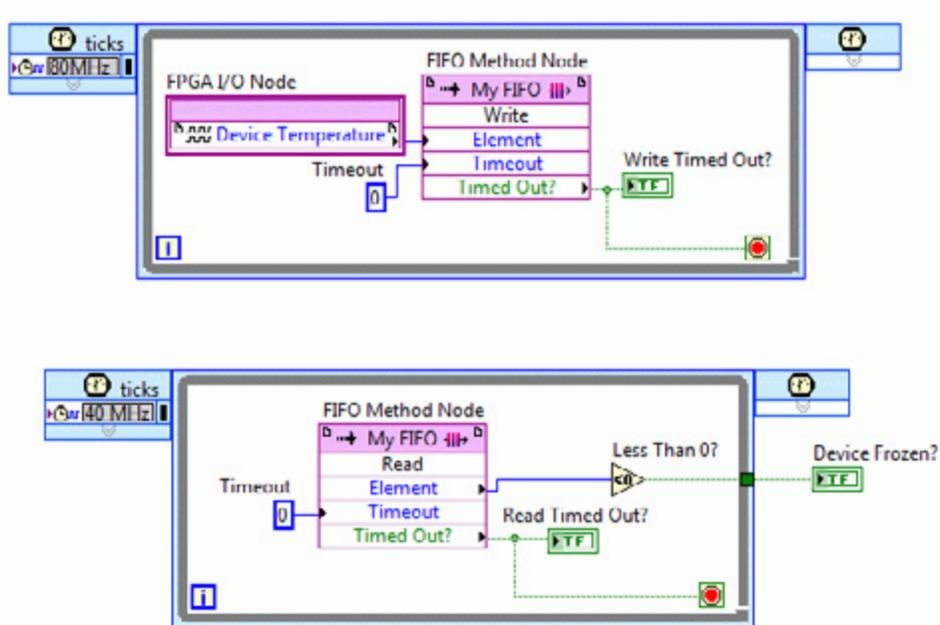


Figure 5.18. Use FIFOs to share buffered data between two parallel loops.

Similar to memory items, you have two types of FIFOs to choose from: target scoped or VI defined. You can access target-scoped FIFOs through all VIs under the FPGA target, and VI-defined FIFOs are scoped to the VI that defines them.

When you configure a FIFO, you must specify the implementation that determines which FPGA resources hold and transfer data in the FIFO. The following recommendations can help you choose one implementation over another.

- **Flip-flops**—Flip-flops use gates on the FPGA to provide the fastest performance. They are recommended only for very small FIFOs (fewer than 100 bytes).
- **Lookup table**—You can store data to two lookup tables per slice on the FPGA. Lookup tables are recommended only for small FIFOs (fewer than 300 bytes).
- **Block memory**—If you are trying to preserve FPGA gates and lookup tables for other parts of your application, you can store data in block memory.

For more information on using FIFOs on an FPGA target, see the LabVIEW Help document [Transferring Data Between Devices or Structures Using FIFOs](#).

Intertarget Communication

You can choose from two methods for communicating data between an FPGA VI and a VI running on the real-time processor: front panel controls and indicators or DMA FIFOs. You can use front panel controls and indicators to transfer the latest values or tags and DMA FIFOs to stream data or send messages and commands. Both methods require using the FPGA Interface functions on the host VI to interact with the FPGA.

Front Panel Controls and Indicators

If you only need to transfer the latest values of data to or from a host VI, you can store the data to controls or indicators on the FPGA VI and access this data using the Read/Write Control function in a host VI. Figure 5.19 shows a simple host VI that is reading from a digital I/O indicator on the FPGA and writing to a digital I/O control on the FPGA.

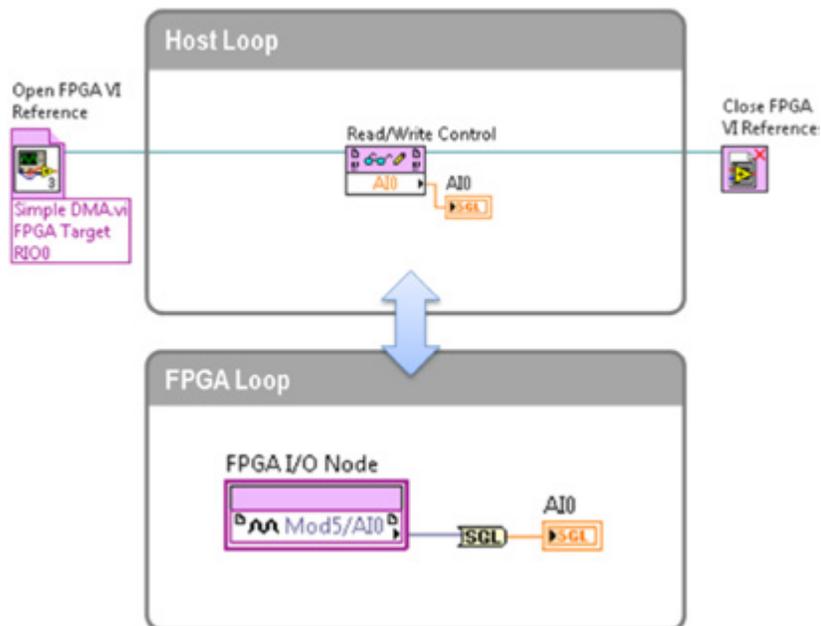


Figure 5.19. Read and write to front panel controls and indicators to share the latest values between targets.

Programmatic front panel communication has low overhead relative to other methods for transferring data between the FPGA and host processor. Read/Write nodes are good candidates for transferring multiple pieces of information between the FPGA and host processor given their relatively low overhead. DMA FIFOs, shown in Figure 5.20, can provide better throughput when streaming large amounts of data, but they are not as efficient for smaller and infrequent data transfers.

Transferring data between the FPGA and host processor using front panel controls and indicators requires involving the host processor more than the DMA FIFOs. As a result, the speed of data transfer is highly dependent on the speed and availability of the host processor. A slower processor or the lack of processor availability results in slower data transfer from the FPGA target to the host. A disadvantage of programmatic front panel communication is that this method transfers only the most current data stored on the control or indicator of the FPGA VI. For example, data could be lost if the FPGA VI writes data to an indicator faster than the host VI can read the data. Also, each control or indicator on the FPGA VI uses resources on the FPGA. Best practices in FPGA programming recommend limiting the number of front panel objects in FPGA VIs.

Additional LabVIEW Help Reference

- [Read/Write Control Function](#)

DMA FIFOs

If you need to stream large amounts of data between the FPGA and real-time processor, or if you require a small buffer for command- or message-based communication, consider using DMA FIFOs for data transfer. DMA does not involve the host processor when reading data off the FPGA; therefore, it is the fastest method for transferring large amounts of data between the FPGA target and the host.

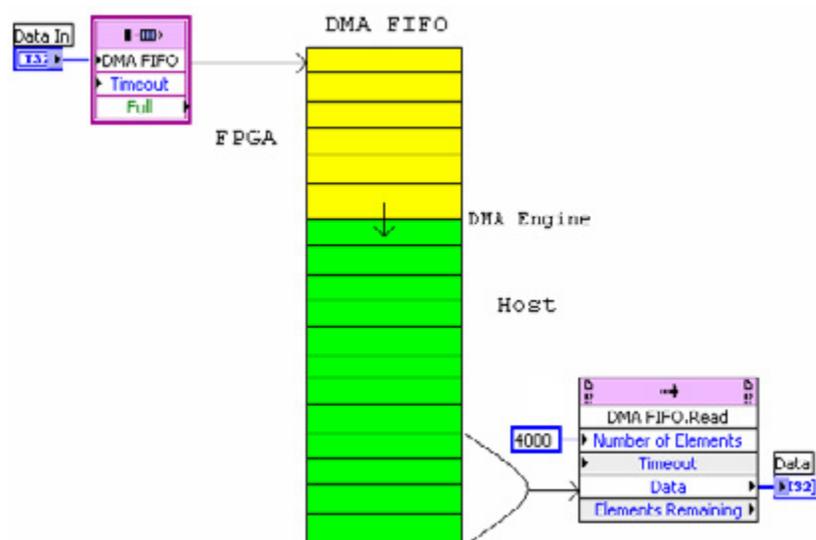


Figure 5.20. DMA uses FPGA memory to store data and then transfer it at a high speed to host processor memory with very little processor involvement.

The following list highlights the benefits of using DMA communication to transfer data between an FPGA target and a host computer:

- Frees the host processor to perform other calculations during data transfer
- Reduces the use of front panel controls and indicators, which helps save FPGA resources, especially when transferring arrays of data
- Automatically synchronizes data transfers between the host and the FPGA target

Though DMA FIFOs are a great mechanism for streaming data, they quickly become complicated when streaming data from more than one channel, optimizing your throughput, or scaling your data on the real-time host VI. If your application requires streaming analog data from one or more analog channels, you should use the [NI CompactRIO Waveform Acquisition Reference Library](#) discussed at the end of this section as a starting point. This library

features an FPGA template and an NI-DAQmx-like API for streaming data and offers many benefits including optimized performance and built-in scaling.

Using DMA FIFOs in LabVIEW FPGA

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. Clicking OK puts this new FIFO into your project, which you can drag and drop to the FPGA block diagram.

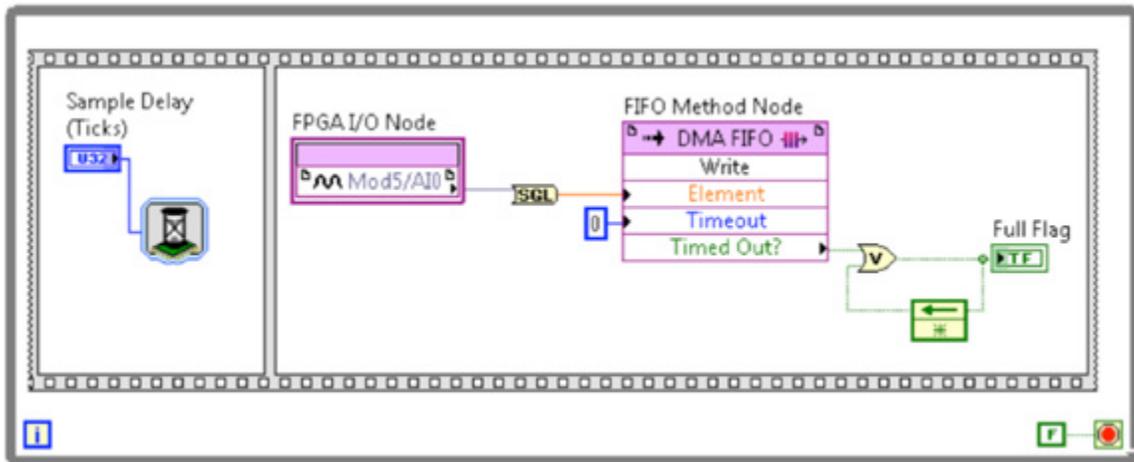


Figure 5.21. Simple DMA Transfer on One Channel (Simple DMA.vi)

Tip: In LabVIEW 2012 FPGA and later, you can convert fixed-point data to single precision floating-point data. Offloading this conversion to the FPGA can save significant processor resources. For earlier LabVIEW versions, you can obtain a similar conversion function through the downloadable example or the NI Developer Zone document [Fixed-Point \(FXP\) to Single \(SGL\) Conversion on LabVIEW FPGA](#).

You can use the same DMA channel to transfer multiple streams of data, such as that collected from I/O channels. Most CompactRIO systems have three DMA channels. In Hybrid Mode, CompactRIO systems have only one DMA channel. To pack multiple data streams or I/O channels into one DMA FIFO, use the interleaving technique shown in Figure 5.22, and unpack using decimation on the host.

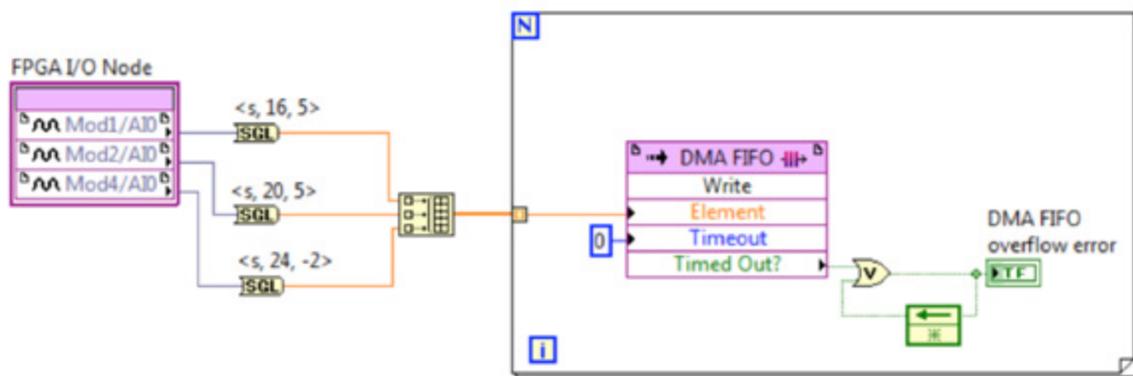


Figure 5.22. You can use build array and indexing on a For Loop to implement an interleaved multichannel data stream.

When passing multiple analog input channels into one DMA FIFO, the channels are stored in an arrangement similar to that shown in Table 5.1. This table assumes that four analog input channels are being interleaved into one DMA

FIFO. The unpacking algorithm on the host VI is expecting the elements to arrive in this specific order. If the FIFO overflows and elements are lost, then the unpacking algorithm on the host VI fails to assign data points to their correct analog input channels. Therefore, it is extremely important to ensure lossless data transfer when reading from multiple analog input channels.

Array Index	Element
0	AI 0
1	AI 1
2	AI 2
3	AI 3
4	AI 0
5	AI 1
6	AI 2

Table 5.1. When writing multiple channels to one DMA FIFO, the host VI expects the elements to arrive in a specific order.

Using DMA FIFOs on the Host

Typically, you dedicate a separate loop on the host VI to retrieve the data from the DMA buffer using the host interface nodes. When reading data from a DMA FIFO on a CompactRIO system, follow these three steps to achieve optimal performance:

1. Set the DMA Read timeout to zero.
2. Read a fixed-size number of elements (a multiple of the number of data channels).
3. Wait until the buffer is full before reading elements.

Figure 5.23 provides a good example of using the DMA FIFO Read functions efficiently to acquire a continuous stream of I/O data. The first DMA FIFO Read function computes the number of elements remaining in the buffer and checks it against the fixed size of 3000 elements. 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 (such as values from eight I/O channels), the Number of Elements should be an integer multiple of eight (such as 80, which gives 10 samples for each I/O channel).

As soon as the buffer reaches 3000 elements, the second DMA FIFO Read function reads the data or sleeps in the False case and checks again on the next iteration. Each DMA transaction has overhead, so reading larger blocks of data is typically better.

Finally, use a Decimate 1D Array function to organize the data by channel.

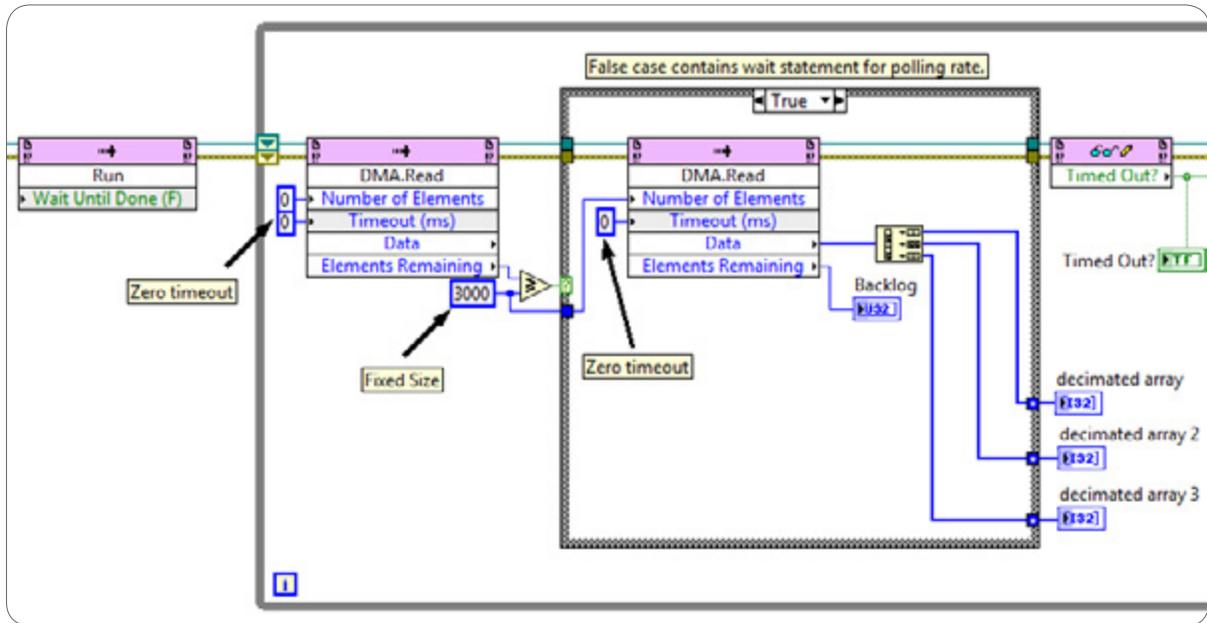


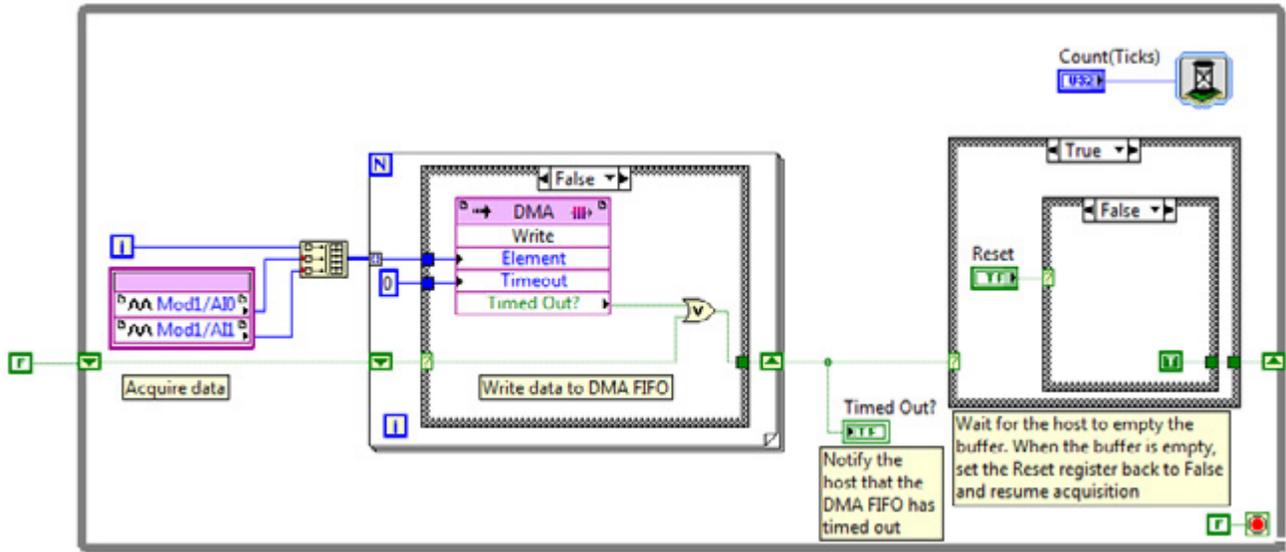
Figure 5.23. Reading Data Off a DMA FIFO (DMA RT.vi)

Whenever you are reading data from a DMA FIFO, you must be able to detect and recover from buffer overflows, or timeout cases, to maintain data correctness. The next section provides a recommended architecture for ensuring lossless data transfer when using DMA FIFOs.

Ensuring Lossless Data Transfer

If lossless data transfer is important, or if you are transferring data from multiple I/O channels into the same DMA FIFO, you must be able to monitor the state of the DMA mechanism and react to any faults that occur. On the FPGA DMA Write Node, a timeout usually indicates that the DMA buffer is full. You should monitor and latch it when it becomes true. You must do this from the FPGA side because simply sampling this register from the host is not sufficient to catch quick transitions. Once you detect a timeout event, you also need to recover from it. Figure 5.24 provides an example of detecting and handling a DMA FIFO timeout case.

FPGA VI



RT Host VI

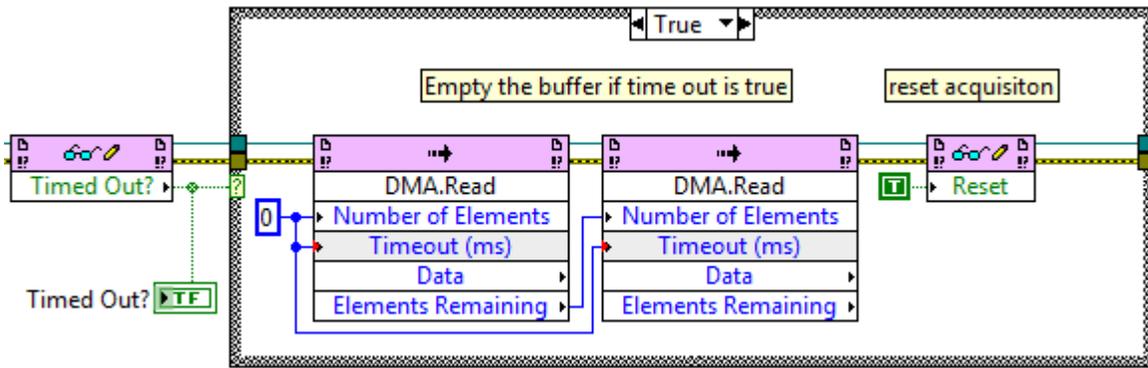


Figure 5.24. Example of Detecting and Recovering From Overflow (DMA Overflow Protection.vi and DMA RT.vi)

In this figure, the timeout event of the DMA FIFO is being monitored on the FPGA VI. When a timeout event occurs, the Timed Out? register is set to True. Once the Timed Out? register is set to True, it is latched until the host VI clears the buffer and sets the Reset register to True. The timeout case on the RT Host VI flushes the buffer by reading the remaining elements. Once the buffer is cleared, the Reset register is set to true and the acquisition on the FPGA VI resumes. By using this method of dealing with the DMA FIFO timeout event, you can detect and recover from buffer overflow so that you can avoid bugs created from missing data points.

Avoiding Buffer Overflows

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

If you are experiencing buffer overflows, you can increase the host FIFO buffer by using a DMA Configure call as shown in Figure 5.25. For FPGA to RT transfers, an overflow often occurs because the host FIFO buffer, not the FPGA FIFO buffer, is not large enough. The size of the DMA FIFO set through the FIFO properties page determines

only the size of the FPGA FIFO buffer that uses up FPGA memory. By default, the host FIFO buffer size is 10,000 elements or twice the size of the FPGA FIFO buffer, whichever is greater.

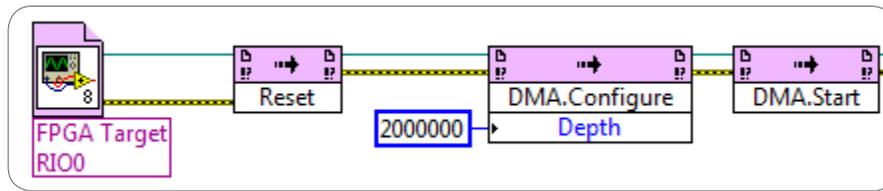


Figure 5.25. Increasing the DMA FIFO size on the host is one way of eliminating buffer overflow issues (DMA RT.vi).

Increasing the size of the buffers can help reduce buffer overflow conditions caused by sporadic events such as contention for buses and the host processor. But if the average transfer rate is higher than what the system can sustain, the data eventually overflows the buffer regardless of the buffer's size. One way to reduce buffer overflow conditions with appropriately sized buffers is to try to read larger amounts of data at a time, which leads to lower overhead per data unit and therefore increases overall throughput.

The CompactRIO Waveform Reference Library

If your application requires the use of DMA FIFOs for streaming analog data from the FPGA, you can jump-start your application development by taking advantage of the CompactRIO Waveform Reference Library developed by NI Systems Engineering. This reference application presents CompactRIO waveform data acquisition VIs and example source code, and supports both delta-sigma and SAR (scanned) modules.

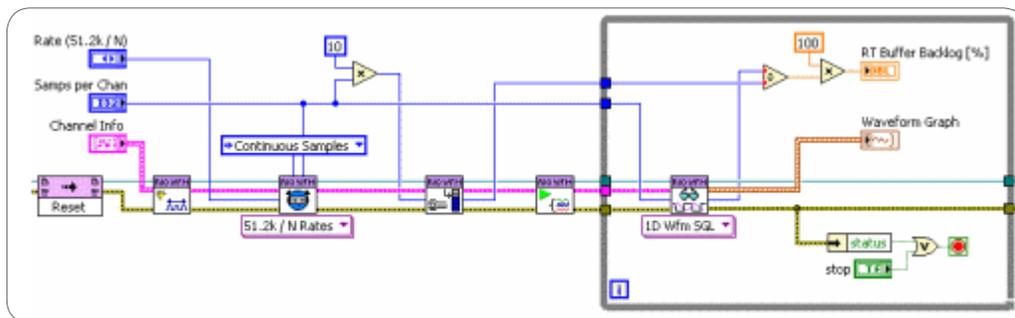


Figure 5.26. Continuous Acquisition Example Using the CompactRIO Waveform Reference Library API

The CompactRIO Waveform Reference Library includes an NI-DAQmx-like API that executes in LabVIEW Real-Time and converts raw analog data into the waveform data type. The waveform data type makes it easy to display multiple channels of I/O to a UI as well as to interface with common data-logging APIs such as Technical Data Management Streaming (TDMS). The API also converts your data to the appropriate engineering units depending on the scale that you configure.

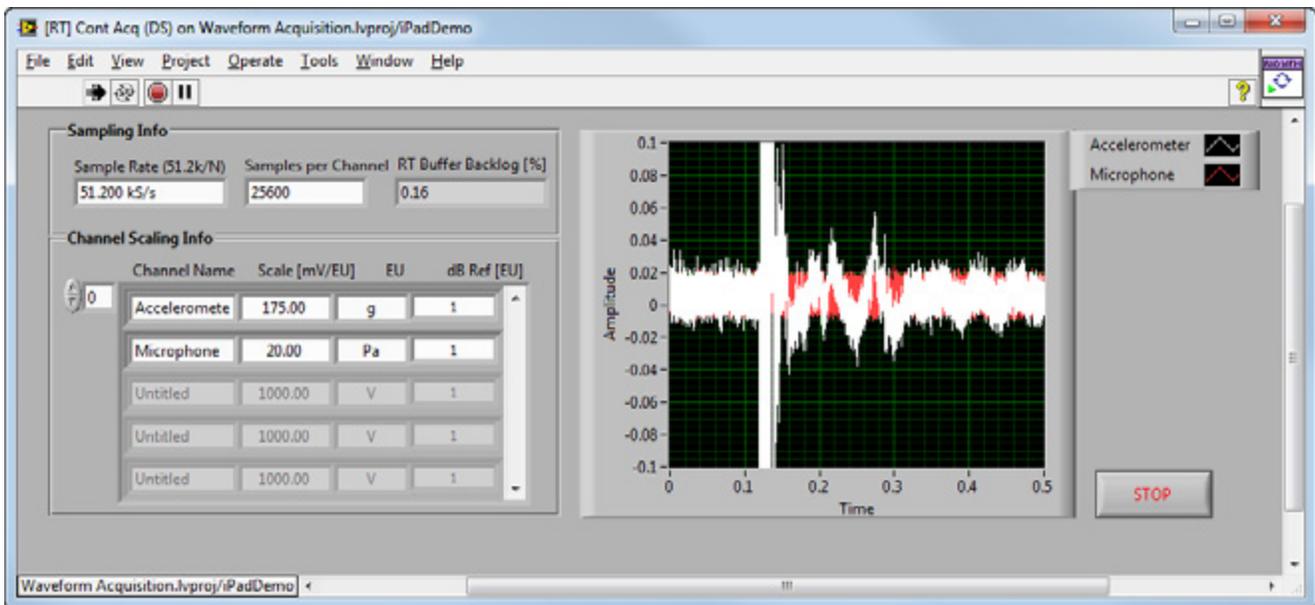


Figure 5.27. When using the CompactRIO Waveform Reference API, you can quickly stream data and display multiple channels of I/O to a user interface.

This library includes an FPGA VI that you need to slightly modify for the VI to reference the I/O modules used in your system. For most applications, you only need to modify the FPGA VI. This VI, designed to offer optimal streaming performance when using DMA FIFOs, includes mechanisms for monitoring and handling buffer overflow conditions. Use this FPGA VI template as a starting point whenever you are streaming data from one or more analog channels using DMA FIFOs.

You can find more information including installation files in the NI Developer Zone white paper [NI CompactRIO Waveform Reference Library](#).



Delta Sigma Acquisition.lvproj and SAR Acquisition.lvproj are available at `\Program Files\National Instruments\LabVIEW[Version]\user.lib\cRIO Wfm_exampleProjects` after installing the library.

Additional References

- [How DMA Transfers Work](#)
- [Best Practices for DMA Applications](#)

For more information on using DMA FIFOs on an FPGA target, see the LabVIEW Help document [Transferring Data Using Direct Memory Access](#).

Synchronizing FPGA and Host VIs Through Interrupts

LabVIEW FPGA features interrupts, which synchronize the FPGA and host VIs by allowing the host VI to wait until a specified interrupt is raised by the FPGA. The FPGA VI can block until the interrupt is acknowledged by the host. This eliminates the need to continually check the status of an FPGA register to see if a given flag has been set, and it lowers CPU use while the interrupt is not raised. On single-core hosts, you can achieve relatively good latency and decreased CPU load when compared to the poll-and-sleep approach. The example in Figure 5.28 shows how you

might implement an interrupt on the FPGA. This example is called the Interrupt Method for Synchronization and can be found in the NI Example Finder.

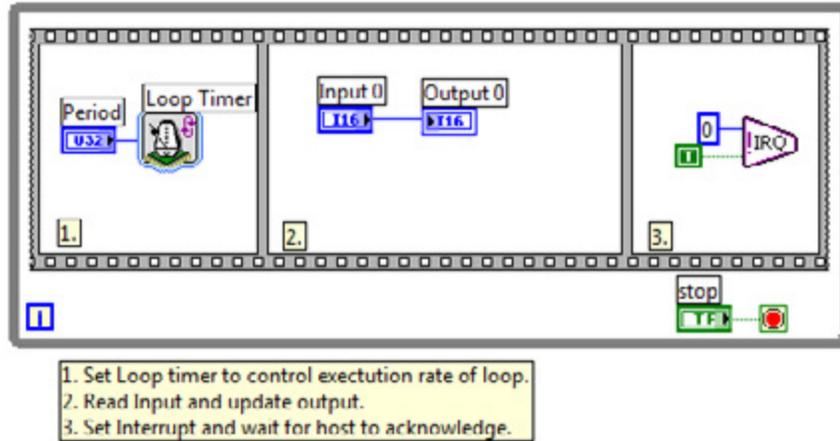


Figure 5.28. Basic LabVIEW FPGA Programming Structure for Sending an Interrupt

Interrupts offer simple code with a straightforward API, but you should be aware of the following caveats:

- Handling the interrupt on the host side involves overhead usually on the order of 5 μ s to 50 μ s. This response latency may be acceptable depending on the application or the frequency of the event.
- If you do not want the FPGA VI to block until the interrupt is acknowledged, you should add a separate FPGA loop for sending the interrupt.
- Interrupts are not the lowest latency option on multicore targets. The fastest response time is obtained by dedicating a CPU core to continuously monitor an FPGA register (no sleep).

Additional References

- [Using Interrupts on the FPGA to Synchronize the FPGA and Host](#)
- [Synchronizing FPGA VIs and Host VIs Using Interrupts](#)

Test and Debug LabVIEW FPGA Code

As described in the [Best Practices for Implementing LabVIEW FPGA Code](#) section, you should develop your LabVIEW FPGA VIs in simulation mode to quickly iterate on your designs and avoid long compile times. When you need to test and debug your VIs, you can stay in simulation mode or take advantage of several other options. You should select an execution mode depending on your requirements for functional verification versus performance and on the type of code you are testing: unit, component, or system. Each type of code has different attributes and verification requirements as described below.

Unit

“Unit” is the most fundamental level of IP you can build in the sense that it maps to a specific processing function or algorithm. You would not split it and test it as a set of smaller functional units.

You want to keep these simple, so you would call it a unit if:

- It can be encapsulated as a subVI you may want to reuse in other parts of your design
- It does not include I/O, data communication, or any target resources
- It does not have multiple loops running in parallel or at different rates
- It is functional in nature—you can provide some known inputs and test for expected outputs
- It may hold state, in which case you may need to call it multiple times to verify it, but its behavior should not rely on the explicit specific passing or control of time

Component

Components are more complex pieces of logic that include elements that can exhibit side effects or focus more on the timing in the system. They are composable by definition, and usually have a clear task or objective to accomplish. An FPGA application can often be broken down into multiple components, and verification at this level ensures that the components interact as expected when integrated into a larger component. You may also want to make sure that a subcomponent is interacting properly with the I/O or host (through the host interface) without waiting until the whole system is assembled.

System

You can think of the system level as the top-most component. It is represented by your top-level FPGA VI plus any additional HDL IP imported through the Component-Level IP (CLIP) Node. It is somewhat different from other components in that its interface is exposed to the host application, so verification tests are either similar to running your host app or they are your host app. Verification therefore requires using the host interface API as well as connecting real I/O signals to your system. A system usually contains multiple While Loops or SCTLs.

Table 5.2 provides guidance on which execution mode you should use for verification and debugging. Keep in mind that if you can perform extensive debugging and verification at the unit and component levels, then you decrease your verification work at the system level.

Execution Mode	Verify Functional Performance	Verify Timing	Verify Integration of HDL IP	Good for Unit Testing	Good for Component Testing	Good for System Testing
Windows PC	X			X		
FPGA Simulation Mode	X	X (only code contained in SCTL)			X	
FPGA Hardware	X	X	X		X	X
Advanced: Third-Party Simulator	X	X	X		X	

Table 5.2. This table provides a comparison of different execution modes for verifying and debugging your LabVIEW FPGA code.

Execute LabVIEW FPGA Code on Your Windows PC

You can execute your FPGA VI on your Windows PC by dragging the FPGA VI to the “My Computer” target in the LabVIEW project. This is the quickest and easiest approach for debugging and/or testing unit-level code. All functions contained in the LabVIEW FPGA palette, excluding target resources, can be executed in the desktop context.

There are several benefits to this approach. For debugging purposes, you have access to standard LabVIEW debugging features and visualization options (graphs, charts, and so on). For testing purposes, you have access to the hundreds of libraries in the LabVIEW for the Desktop programming palette.

Do **not** make changes to your LabVIEW FPGA code while in this context since you are no longer confined to the limited LabVIEW FPGA palette and you might introduce constructs that are supported only on the desktop. The My Computer execution context is ideal for developing tests that involve writing code around your LabVIEW FPGA VI.

Execute in Simulation Mode

Another option is to use LabVIEW FPGA simulation. Simulation executes your FPGA code on the host computer using a built-in high-fidelity, bit-accurate simulator. The simulator is cycle-accurate when simulating code contained within an SCTL, assuming that the design can compile at the desired rate. The simulator supports FPGA target resources such as I/O, memory items, and DMA FIFOs, so it can be used for code at the unit and component levels. You can configure your LabVIEW FPGA VI to run in the simulation mode by right-clicking FPGA Target in the LabVIEW project and selecting Execute VI on»Development Computer with Simulated I/O.

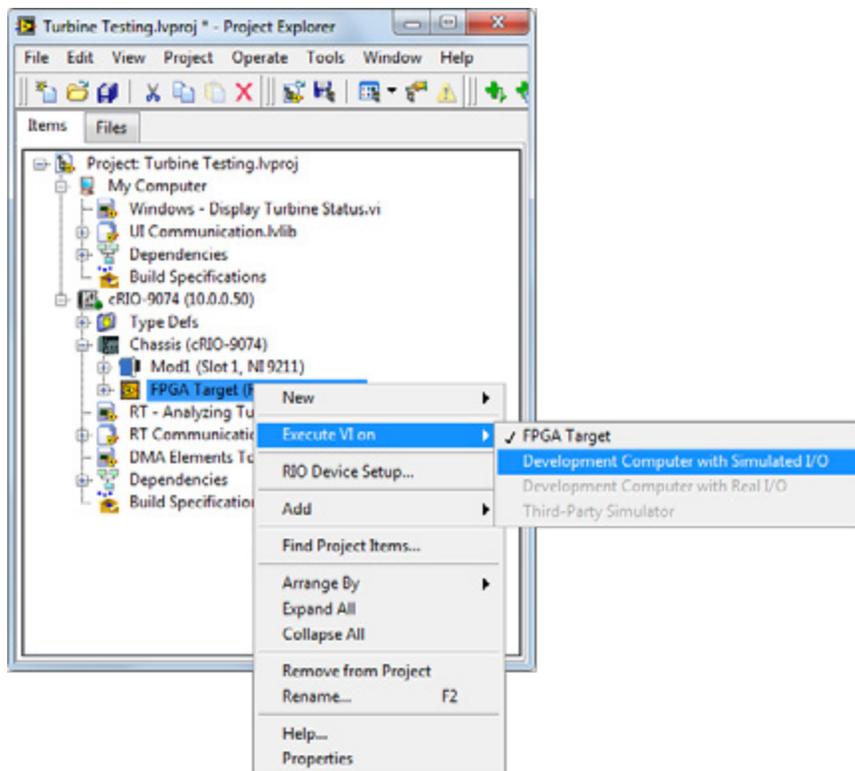


Figure 5.29. You can change the execution mode of your LabVIEW FPGA VI by right-clicking the FPGA target in the LabVIEW project.

Debugging in Simulation Mode

When executing your LabVIEW FPGA VI in simulation mode, you have access to standard LabVIEW debugging features including highlight execution, probes, and breakpoints. LabVIEW 2013 and later includes an additional debugging tool called the Sampling Probe. When inserted into your FPGA design while running in simulation, these probes provide a unique probe window of the different digital signals in relationship to one another with respect to time (or tick/clock iteration count in this case).

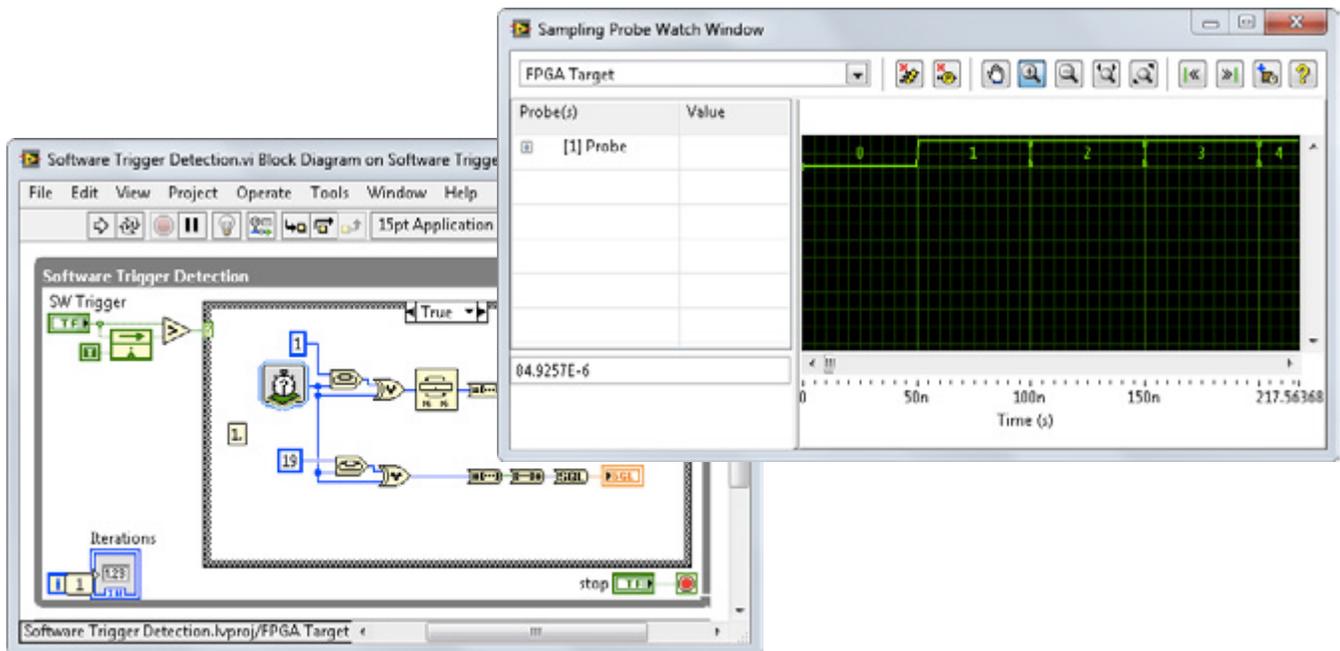


Figure 5.30. View signals in relation to each other with respect to time using the new LabVIEW FPGA sampling probe.

The window also includes functions for locating the next rising edge or previous rising edge on a signal, and zoom capabilities to navigate through the data. For information on how to use the Sampling Probe, see the [LabVIEW FPGA Help Document: Debugging Using Sampling Probes](#).

Testing Your LabVIEW FPGA Code in Simulation

You can create tests in LabVIEW that exercise your LabVIEW FPGA components. The recommended approach is to have a VI that performs testing (or testbench VI) running in the host (My Computer or Real-Time) context, and the LabVIEW FPGA VI running in the FPGA context in simulation mode or in actual hardware. Since the testbench VI runs in the host context, you have access to hundreds of functions that you can use to create test vectors, cases, and stimuli in addition to the checking, analysis, display, and reporting of test results.

You can also use the same library of functions to build a reference implementation of your component. You can import implementations built in C/C++ or any other language that can generate a DLL or shared library as well as take advantage of support for other languages such as .m script or The MathWorks, Inc. Simulink® software. Figure 5.31 provides a diagram of a basic test, all of which can be implemented within the LabVIEW graphical development environment.

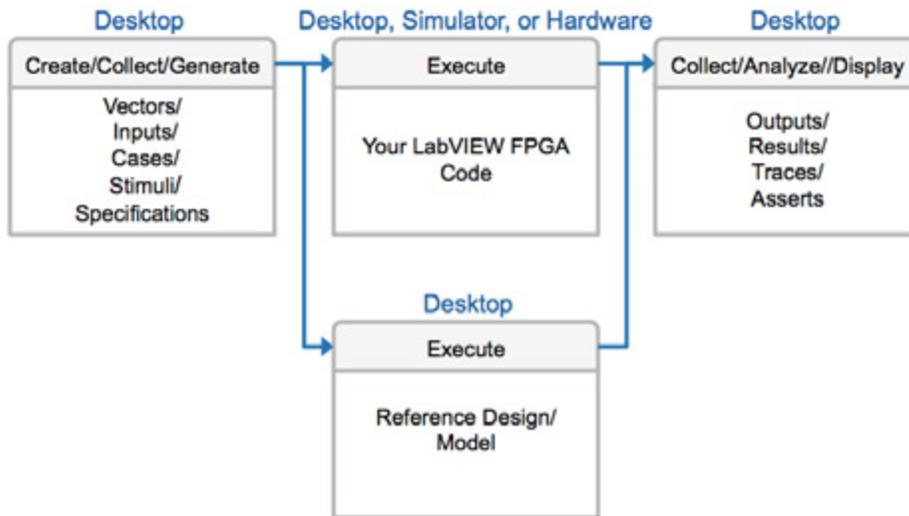


Figure 5.31. A typical test bench consists of several components, all of which can be implemented in the LabVIEW graphical development environment.

In the LabVIEW FPGA Module Version 2013 and later, you can test LabVIEW FPGA components using the Desktop Execution Node. With the Desktop Execution Node, you can perform verification on a LabVIEW FPGA VI without making changes to your LabVIEW FPGA code to accommodate the testing. The Desktop Execution Node uses simulated time to reflect timing in hardware. You can provide stimuli to controls, indicators, and I/O before time advances; force time to advance a set number of ticks; and then read the values of the controls, indicators, and I/O before providing additional stimuli to the VI.

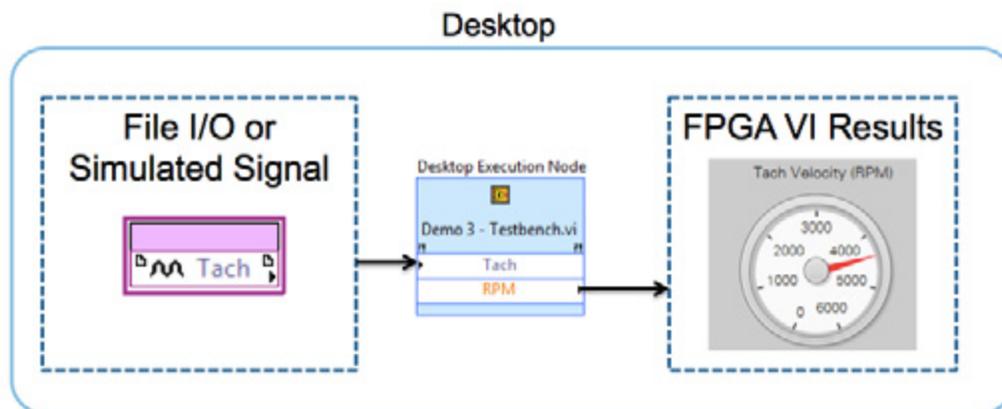


Figure 5.32. The Desktop Execution Node can be used to create a test bench for your LabVIEW FPGA VI

Understanding Simulated Time

To be successful with the Desktop Execution Node, you need to understand the concept of simulated time. Starting with LabVIEW 2013 FPGA, a developer may consider two timing paradigms when creating an FPGA design: wall clock time and simulated time. Wall clock time is the actual, real-world time a slice of logic may take to execute. An FPGA device that has been programmed with a bitfile runs in wall clock time. Simulated time is an event-driven model of wall clock time. During execution, some nodes may announce that a time step is required at a particular time. When LabVIEW detects that execution is idle, simulated time advances to the next earliest time step.

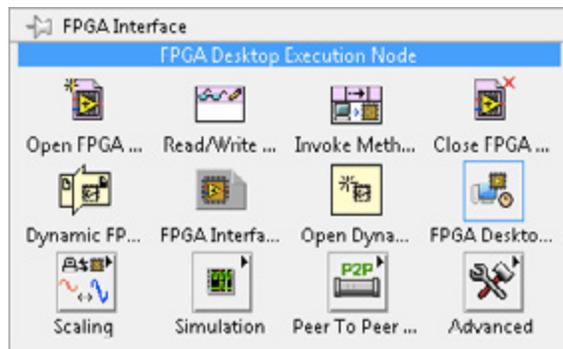


Figure 5.34. The Desktop Execution Node is on the FPGA Interface palette.

Step 2: Configure the Desktop Execution Node

Select Your VI

Each component you want to verify using the Desktop Execution Node must be saved as a VI prior to testing. You can then point to your FPGA VI from within the Desktop Execution Node configuration window. Once you select your FPGA VI, all of the controls, indicators, and FPGA resources that you have access to populate under Available Resources.

Select Terminals

You can configure the Desktop Execution Node terminals by selecting your resources of interest and using the blue arrows to copy them over into the Selected Resources window. For this example, you want to write to the analog input node named **Tach** and read from the indicator named **RPM**.

Select a Reference Clock

To configure the Reference Clock, select the clock that the loop within your FPGA VI is referencing. The default clock on most RIO hardware targets is 40 MHz. If you are using an SCTL, the reference clock could be the top-level clock or a derived clock.

Notes

- Most components contain a single loop. If your component contains multiple loops, select the fastest clock referenced in your VI.
- If you are performing unit testing and your code is not contained within a loop, encompass your unit code within a While Loop for the purpose of testing. Otherwise, the Desktop Execution Node stops after completing the first iteration.

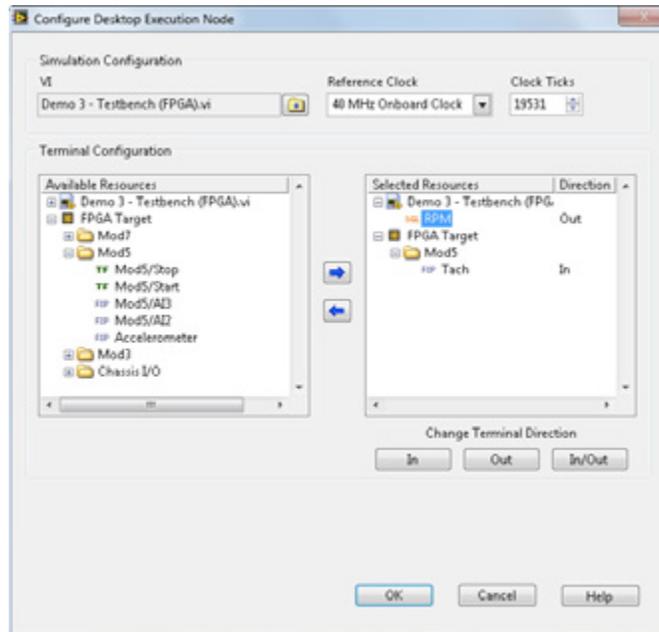


Figure 5.35. The first step to configuring the Desktop Execution Node is to select your FPGA VI.

Determine the “Clock Ticks” or Simulated Time

The Desktop Execution Node can control simulated time so developers can alter stimuli at key points during execution on the development computer with simulated I/O. To successfully use this feature, you need to measure the time your FPGA VI takes to complete or you need to design your VI in such a way that the time it takes to complete is intuitively known (for example, using a Loop Timer to guarantee timing). Some tips on measuring the time that your FPGA VI takes to complete are below.

Single-Cycle Timed Loop

If you are using an SCTL, the code contained within that loop always takes one tick of the reference clock to execute, so you can set the Clock Ticks to 1 tick. If you have multiple SCTLs, select the fastest clock as the reference clock.

While Loop With a Loop Timer

In this case, you can specify the value of the Loop Timer in ticks. If your loop timer is configured in milliseconds or microseconds, perform the conversion. For example, if you have a Loop Timer set to 10 μ s, perform the following calculation:

- Ticks = clocks (Hz) x Time (s)
- Ticks = 40,000,000 Hz x 0.00001 seconds
- Ticks = 400

Therefore, you would configure your Clock Ticks input in the Desktop Execution Node to 400 ticks.

While Loop With No Loop Timer

If you are using a While Loop that does not contain any analog I/O (it may contain digital I/O), the easiest way to measure the number of ticks is by using the Sampling Probe. To do this, create an indicator off the While Loop iteration terminal. Right-click the wire and select **Sampling Probe»FPGA**. Run the VI in simulation mode for a second or two, stop, and view the Sampling Probe window. In the example below, you can see that the While Loop

takes two ticks of the 40 MHz clock to execute, according to the Sampling Probe window (each tick equals 25 ns). For this case, you would set the Clock Ticks input to two ticks.

Note: Because this is a While Loop, the number of ticks measured in simulation does **not** necessarily equal the number of ticks that result when executing the same code in hardware. Any code inside an SCTL, however, is guaranteed to be cycle accurate.

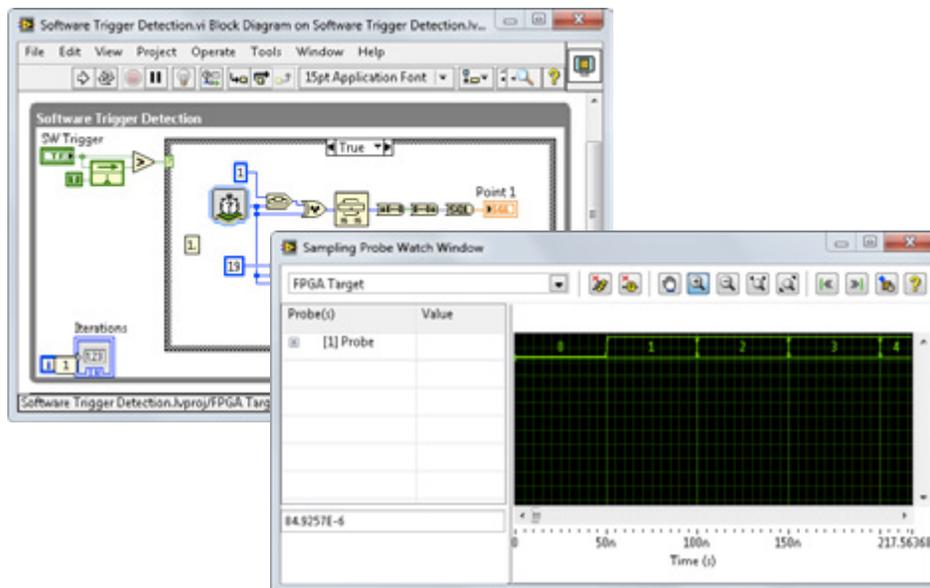


Figure 5.36. If your While Loop does not contain a timing source, you can measure the number of clock ticks per iteration by using a Sampling Probe.

While Loop Executing at the Rate of a Delta-Sigma Module Clock

You may have an FPGA VI that is executing at a rate defined by a delta-sigma analog input module, such as the VI used in the example shown in Figure 5.37. In simulation mode, LabVIEW ignores the timing input from these property nodes. Therefore, these situations require you to perform two steps:

1. Control the simulation timing of the VI by adding a Loop Timer that is equivalent to the scan rate
2. Set the Clock Ticks input in the Desktop Execution Node to the same value that is input to the Loop Timer

In this specific example where you are reading from a tachometer, the analog input module is configured to run at a rate of 2.048 kS/s, or 488 μ s. For the first step, you need to add a Loop Timer to the While Loop and configure it to a loop rate of 488 μ s.

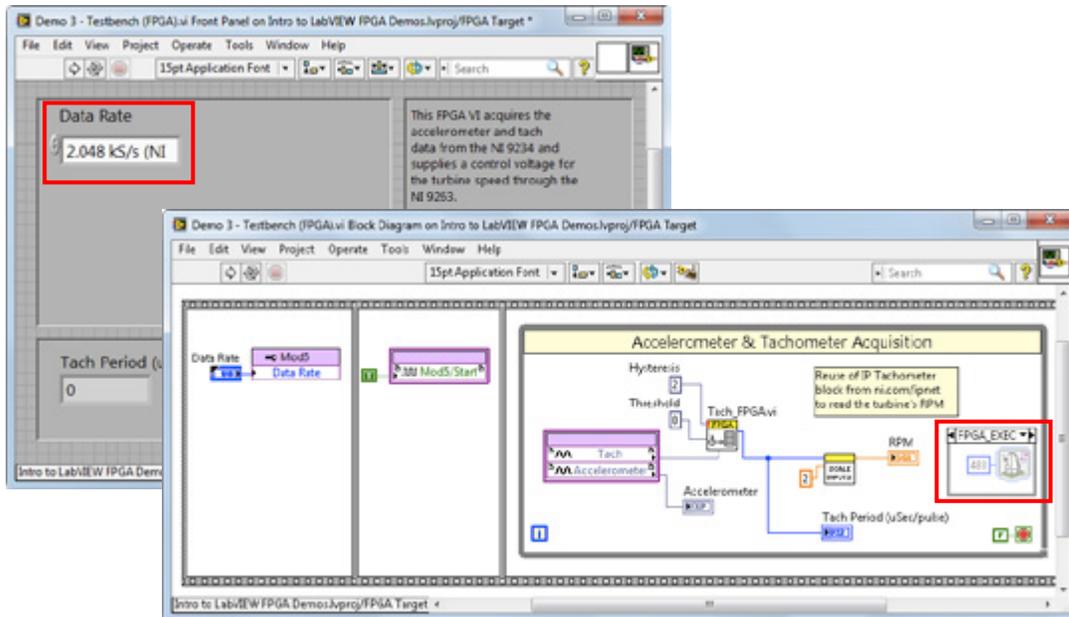


Figure 5.37 When the timing source of your loop is the onboard clock of a delta-sigma analog input module, you need to specify the simulated time by adding a conditional Loop Timer.

You can put the Loop Timer inside a Conditional Disable structure so that it is called only when the VI is executing in simulation mode. Configure the Conditional Disable structure with the following values:

- `FPGA_EXECUTION_MODE == DEV_COMPUTER_SIM_IO`

The Loop Timer should be placed inside this conditional case, and the default case should be empty.

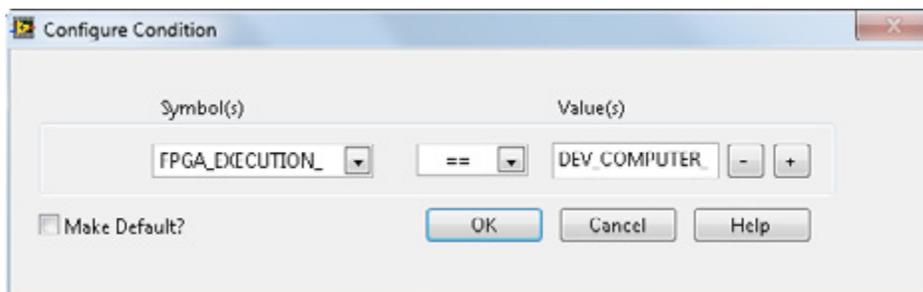


Figure 5.38. Configure a Conditional Disable structure to execute the Loop Timer only when in simulation mode.

Finally, the last step is to set the Clock Ticks input in the Desktop Configuration Node. In this example, 488 μ s is equal to 19,531 ticks, so you set the Clock Ticks input to 19,531 ticks.

To learn more about using the Desktop Execution Node, see the NI white paper [Using the LabVIEW FPGA Desktop Execution Node](#).

Step 3: Build Your Test Bench

Once you have successfully configured the Desktop Execution Node, you can begin building your test bench. In the example shown below, logged tachometer data is being read from a TDMS file and written to the analog input I/O node called Tach. The FPGA VI is executed in simulation mode, and the resulting rpm is displayed on a chart. To verify the tachometer IP, you can view the tachometer data read from the file and the rpm data plotted to the chart.

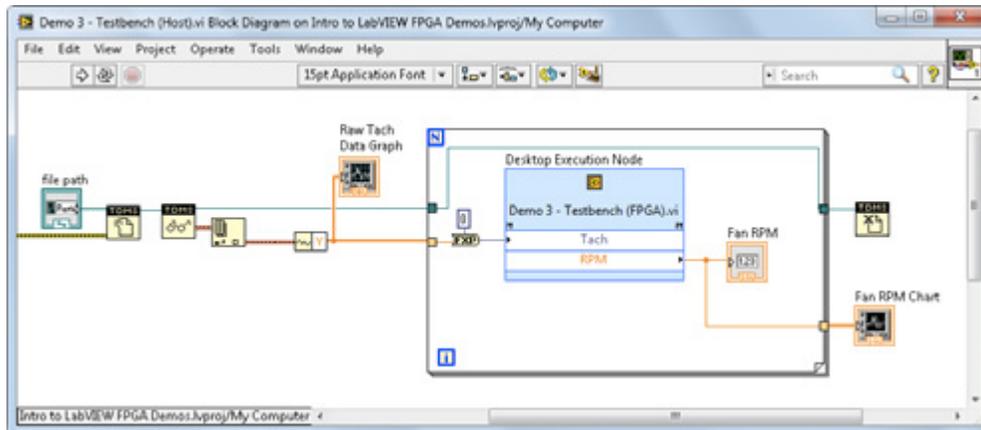


Figure 5.39. Once the Desktop Execution Node is configured, you can create a test VI and verify your FPGA code.

Execute Within a Third-Party Simulator

If you are required to verify timing in addition to functionality, before you compile your LabVIEW FPGA VI to hardware, you can interface with three third-party simulators: Mentor Graphics ModelSim (LabVIEW 2013 and earlier), Mentor Graphics Questa, and Xilinx ISim. You can use the design verification and debugging capabilities of these simulators to debug functional behavior and timing-based errors. To use ISim with the LabVIEW FPGA Module, you should be familiar with HDL simulators and VHDL, which is required for writing a test bench. You now have the option to write test benches for ModelSim using either VHDL or LabVIEW. The white paper [Cycle-Accurate Co Simulation with Mentor Graphics ModelSim](#) provides a tutorial for interfacing to ModelSim.

Execute in Hardware

Debugging in simulation mode or in the desktop context can save you time, but situations do arise that require the code to be compiled to the FPGA target and then run and debugged in real time. In these situations, you have to write additional code into the application to test and validate the application's core functionality. This code is usually removed or disabled when debugging is complete. The [NI FPGA Debug Library](#) component offers a set of simple functions to debug LabVIEW FPGA applications in real time. In addition to performing common simple tasks, these functions provide a modular programming interface to help you rapidly construct advanced debugging structures.

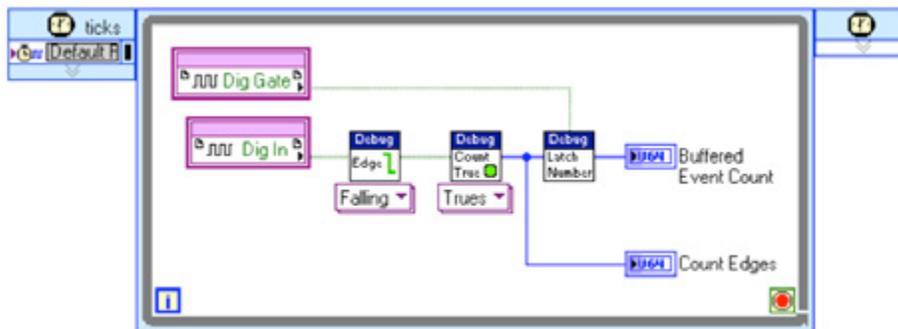


Figure 5.40. The NI FPGA Debug Library contains VIs that you can use for common simple tasks, or advanced debugging structures like the advanced I/O counters shown here.

Optimizing LabVIEW FPGA Code

When you use regular LabVIEW programming techniques in LabVIEW FPGA, you immediately reap the benefits of the FPGA-based approach. Sometimes you may need to push your system even further in one or more of these related dimensions: throughput, timing, resources, and numerical precision.

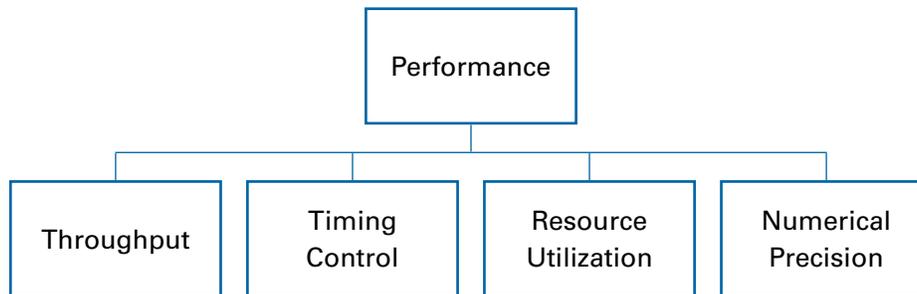


Figure 5.41. With LabVIEW FPGA programming techniques, you can push your system further for throughput, timing control, resource utilization, or numerical precision.

These dimensions are often interconnected, so improving your design with one of them affects the others sometimes positively but more often at their expense. It is important to understand these dimensions and how they relate to each other, so this guide lists some basic definitions below and explores related techniques in later chapters.

Throughput

Throughput is a key concern for DSP and data processing applications. It is measured as work per unit of time. In the majority of applications using NI RIO hardware, work refers to the processing or transfer of samples, so their throughput is usually measured in samples per second or some equivalent form such as bytes, pixels, images, frames, or operations per second. The fast Fourier transform (FFT) is an example of a processing function with throughput that can be measured in FFTs, frames, or sample per second.

The [High-Performance RIO Developer's Guide](#) offers a more in-depth discussion of the factors that affect throughput as well as a set of techniques that can help you achieve higher throughput when creating LabVIEW FPGA applications.

Timing Control

Timing control refers to the ability to prescribe and measure the amount of time between events of interest in the system. When you use LabVIEW FPGA, your designs are translated to a hardware circuit so you can create designs that have fast timing responses with little jitter. Control applications usually require a guaranteed maximum response time between system sampling and control signal updating. This amount of time is referred to as latency. In digital protocol applications, timing specifications may refer to the target, minimum, or maximum time between events related to the data or signals being transmitted. Precise timing control is important to both the control and digital protocol application areas.

The [High-Performance RIO Developer's Guide](#) for a more in-depth discussion of latency as well as a set of techniques that can help you achieve lower or more precise timing responses when using LabVIEW FPGA.

Resource Utilization

An FPGA has a finite number of resources and is usually much more constrained in storage and memory elements than a processor or microcontroller. Being able to fit your design into the FPGA is a hard constraint on the whole development process. FPGAs are also made up of different types of resources, so running out of one type of resource can keep you from progressing in your applications.

More importantly, resource utilization can have a dramatic impact on the other performance dimensions, especially throughput and meeting timing constraints. Refer to the [High-Performance RIO Developer's Guide](#) for descriptions of the different resources that compose the FPGA and how you can balance them to make your design fit and increase its performance.

Numerical Precision

Numerical precision concerns revolve around having enough digits or bits so your application can work correctly. Inadequate numerical precision is considered a functional problem that must be avoided and tested against. The number of bits used to represent system variables, including the number of bits used for integers, the integer and fractional parts of fixed-point numbers, and the dynamic range of floating-point numbers, can have a substantial impact on the performance and resource utilization of your application, so you should consider this.

Compiling LabVIEW FPGA Code

The LabVIEW FPGA compilation process is described in the first section of this chapter, “[FPGA Technology](#).” This process can take up to several hours depending on how intricate your design is. This section provides some tips for reducing compile times and understanding the compile report.

Reduce Compile Times

NI provides several options for compiling your LabVIEW FPGA code. When you create your LabVIEW FPGA build specification in the LabVIEW project, you see the options in a dialog similar to the one shown in Figure 5.42.

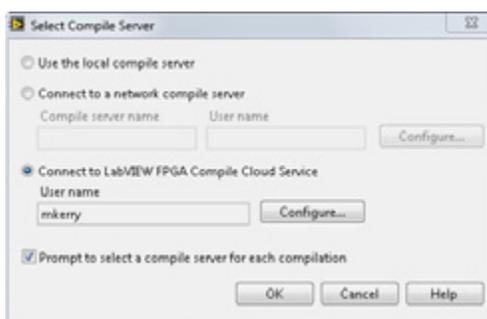


Figure 5.42. You have several options for selecting a compile server when compiling your LabVIEW FPGA VI.

Use the local compile server—By default you can compile on your local development PC. This is a viable solution for compiling small VIs, but if you are concerned about long compile times, you should consider the other options below.

Connect to a network compile server (single server)—You can choose to offload your compiles to a single Windows- or Linux-based machine on the network. Compilation times are proven to be reduced when compiling on a Linux machine versus a Windows machine.

Connect to a network compile server (farm)—If you are working on a team, you can use the LabVIEW FPGA Compile Farm Toolkit to set up a compile farm with multiple workers. This toolkit helps you create an on-site server to manage FPGA compilations. You can connect as many worker computers as you need, and the central server software manages the farming out of parallel compilations and queuing. This is an effective way to reduce compile times if you cannot use cloud technology for your project.

Connect to the LabVIEW FPGA Compile Cloud Service—This service helps you compile your FPGA VIs on Linux with the latest dedicated high-RAM high-end computers. Depending on the size of your LabVIEW FPGA VI, you may notice substantial reductions in your compile times compared to compiling on your Windows desktop. Compiling in

the cloud also adds the capability to compile many VIs in parallel. To try a 30-day trial of the LabVIEW FPGA Compile Cloud Service, visit ni.com/trycompilecloud.

Reading the Compile Report

After compiling an FPGA VI, LabVIEW displays a Compile Report that contains information about the overall size and speed of the application. This information can help you decide how to optimize your code if necessary.

Note that the compiler algorithm is not deterministic, and you might get different results from one compile to the next even if you do not change the VI or the compiler settings. You can change the compiler settings if the compile fails by right-clicking the build specification in the LabVIEW project and selecting **Properties** from the context menu. In the Properties page, select the category **Xilinx Options**.

VI Size

The Final device utilization (map) report shown in the dialog of a successful compile offers information about the number of slice registers, slice lookup tables, and multipliers used as well as the amount of block RAM used. A recommended best practice is to keep your overall FPGA usage below 90 percent in your final application. If you upgrade your software in the future and need to recompile your VI, you may find that a different version of the Xilinx compilation tool chain uses more or less fabric. In this case, you need some extra room to work with.

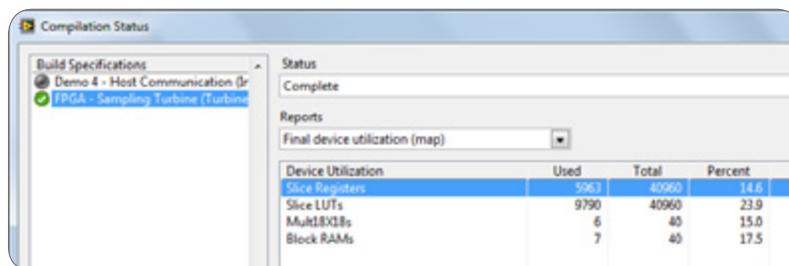


Figure 5.43. Final Device Utilization (Map) Report

VI Speed

The Successful Compile Report dialog box also contains information about the clock rates of the application.

- **Requested**—Displays the clock rate at which the compiled FPGA VI runs. The default setting is 40 MHz.
- **Maximum**—Displays the theoretical maximum compile rate for the FPGA VI.

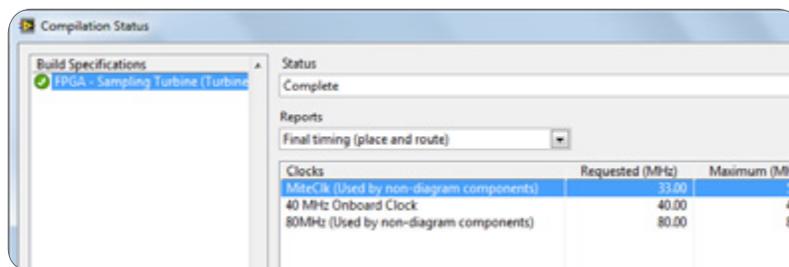


Figure 5.44. Final Timing (Place and Route) Report

If the maximum rate is slower than the requested rate, an error results and the compile process stops. You must modify the application to the point where the maximum rate is equal to or greater than the requested rate. A common problem when you use SCTLs is that the Requested Rate exceeds the Theoretical Maximum. Refer to the “Single-Cycle Timed Loop” section of this chapter for more information about optimizing FPGA applications using SCTLs.

CHAPTER 6

Timing and Synchronization of I/O

Understanding how data travels between module hardware components and the LabVIEW FPGA block diagram can help you develop better programs and debug faster. This section introduces you to the different hardware architectures of analog and digital C Series I/O modules and how to communicate with each one. These modules are typically used for measurement or control signals and feature model numbers that follow this style: NI 92xx, NI 93xx, or NI 94xx.

Some basic terminology used in this section is listed below.

- **ADC**—Analog-to-digital converter. Discrete component that converts an input analog signal (usually voltage) into a digital representation. Front-end circuitry, also known as signal conditioning, is used to convert real-world analog signals into voltage levels within the set range of the ADC.
- **DAC**—Digital-to-analog converter. Discrete component that converts a digital value into an analog value. Analog output is usually a voltage, but, if you add circuitry, you can convert it into a current value.
- **Arbitration**—The process of providing one request priority while causing all other requests to wait.
- **Jitter**—Inconsistent periods between multiple iterations of a looping program structure. Measured as the difference between the longest period experienced and the nominal period requested.

LabVIEW FPGA Communication Nodes

You can use three graphical function blocks to communicate with modules from a LabVIEW block diagram. At a lower level, these programmatic interfaces vary due to hardware architecture differences. For example, the graphical function blocks that retrieve data from AI Channel 0 all look the same even though at a lower level, the raw FPGA communication differs between modules. This abstraction reduces development time and provides the open environment that supports several chassis and module combinations.

The three main function blocks that communicate with C Series modules are the I/O node, method node, and property node.

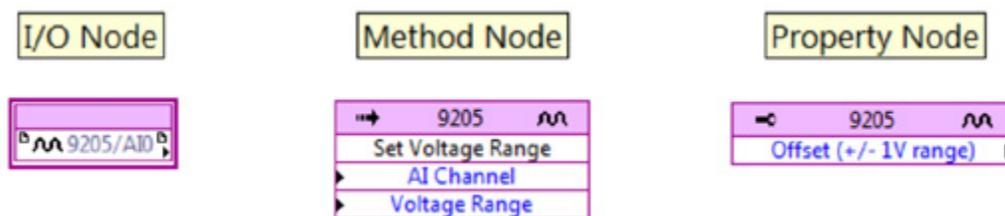


Figure 6.1. The I/O node, method node, and property node for LabVIEW FPGA visually have subtle differences.

I/O Nodes

- Pull data from hardware channels
- Read calibration information
- Are designed to be “thin” interfaces to the module (in other words, minimal data or timing manipulation)
- Block loops until data is available
 - Cannot be used in SCTLs (except I/O nodes for parallel digital lines)

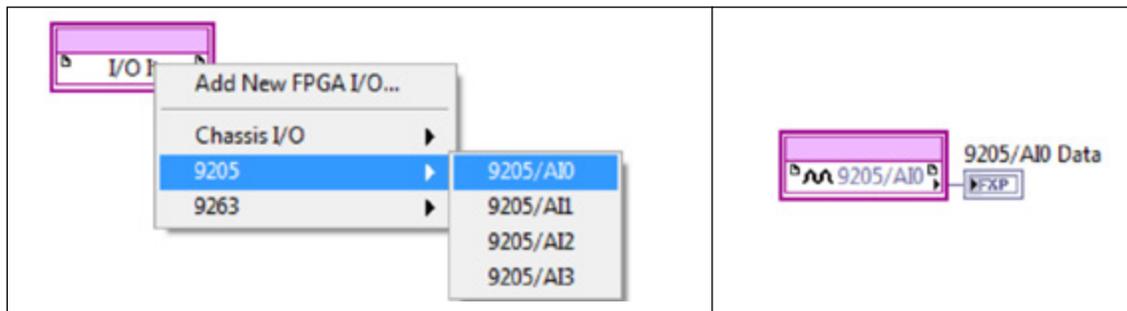


Figure 6.2. I/O Node Selection Menu (left) and an I/O Node Set to Channel A10 (right)

Method Nodes

- Invoke features that are special to a particular set of modules
- Used when the method involves multiple parameters
- Examples include
 - Wait for change on a digital line
 - Triggering on the NI 9205 C Series analog input module

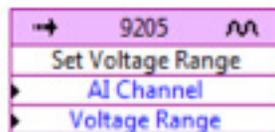


Figure 6.3. Method Node to Set Voltage Range on the NI 9205

Property Nodes

- Access module properties
- Feature some properties that are available during run time
- Examples include
 - Data rate setting for delta-sigma modules
 - Calibration constants
 - Serial number
 - Module ID
 - Vendor ID

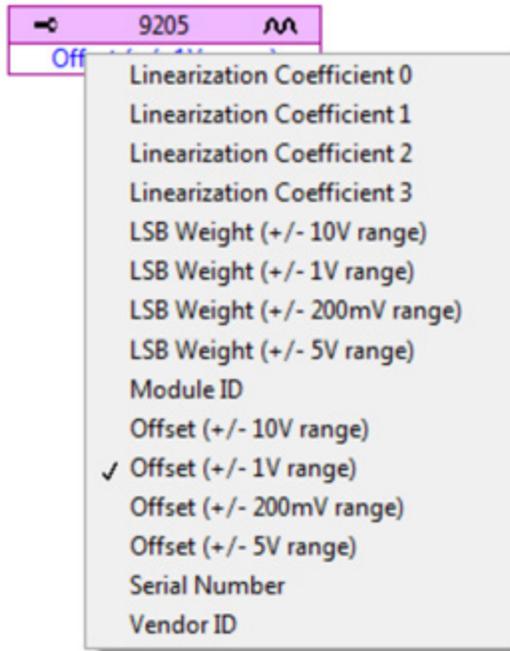


Figure 6.4. Property nodes for some modules feature several options.

Specialty C Series Modules

Some specialty C Series modules such as motor drivers, CAN, serial, and SD card modules do not have a generalized API for the FPGA like the analog and digital modules do. These NI modules have examples in the NI Example Finder that include LabVIEW FPGA and LabVIEW Real-Time host code, and most third-party modules are shipped with examples. Use these examples as a starting point if you are using a specialty module in your application.

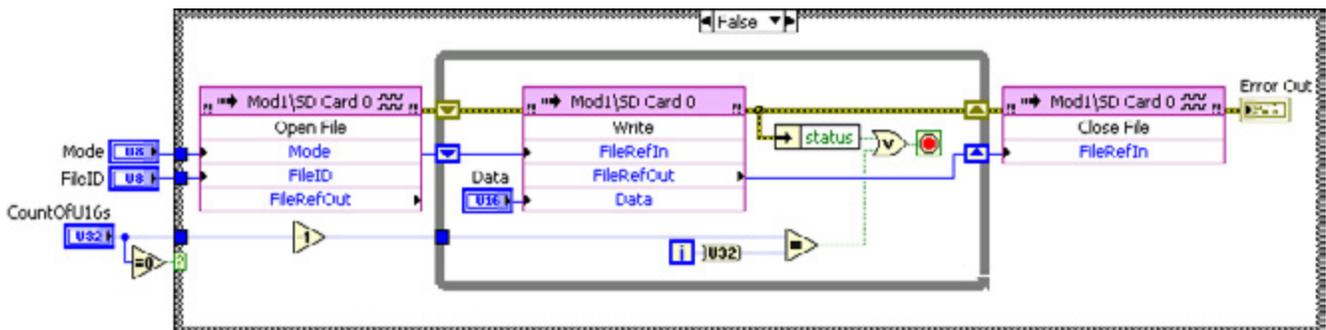


Figure 6.5. Write SD Card Example for the NI 9802

Module Classifications

This section describes the different types of C Series I/O modules. You need to understand how these modules are designed to properly implement timing and synchronization. The basic types of module classifications are presented in Figure 6.6.

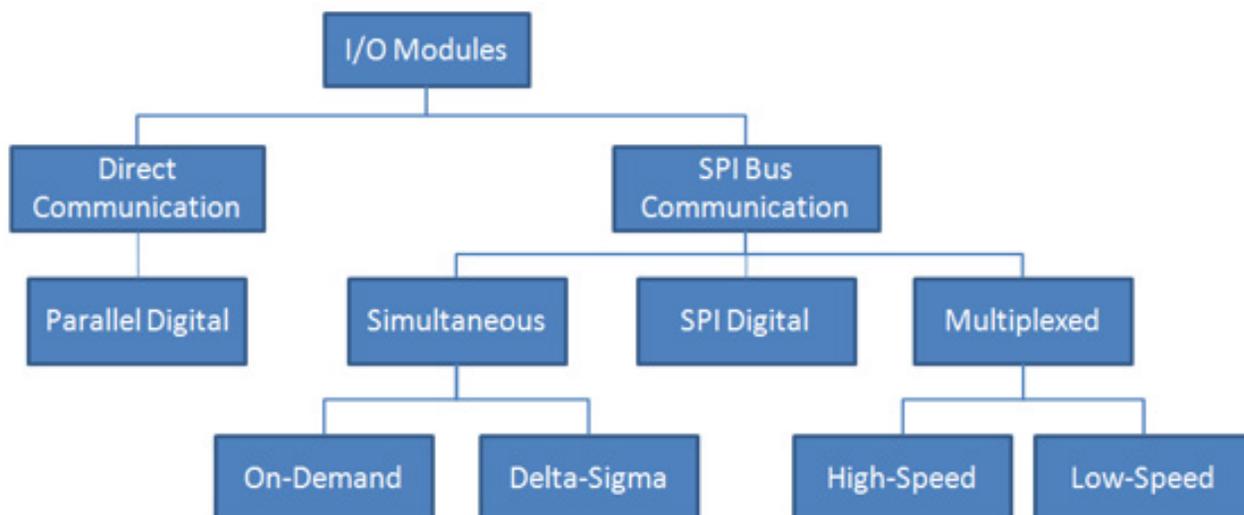


Figure 6.6. C Series Module Classification Organizational Tree

Direct FPGA Communication

Modules that feature direct communication with the FPGA in a CompactRIO chassis route signals from the I/O connector on the front of the module through the 15-pin D-SUB on the back of the module to the FPGA. This architecture makes programming with direct FPGA communication modules easier because you can think of each line as a unique programming entity. The only modules that use direct FPGA communication are digital I/O (DIO) modules with eight or fewer lines.

Parallel Digital

Because each of the digital lines on a direct FPGA communication module is programmed as a separate entity, digital modules with eight or fewer channels are also referred to as parallel digital modules. The parallel design of these modules makes them the easiest to program in LabVIEW. I/O nodes for parallel digital modules can be called from SCTLs, which make programming more efficient, and individual channel lines can be called simultaneously from separate loops, as seen in Figure 6.7, with no concern for arbitration.

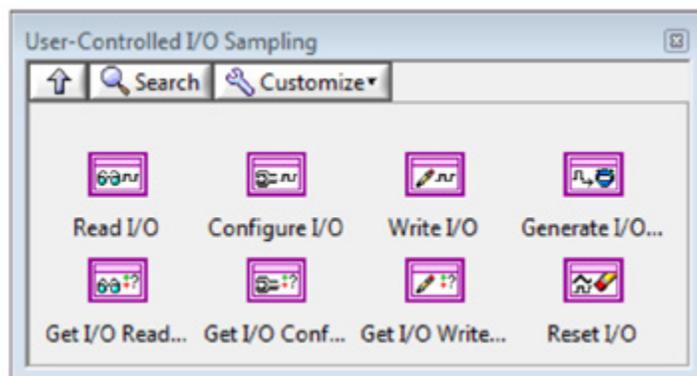


Figure 6.7. You can call lines on the same parallel digital module from separate loops.

Be aware that a call for direction change, at any time, on the DIO lines of an 8-channel parallel module causes an interrupt that you must handle appropriately on the LabVIEW block diagram to prevent a glitch or jitter on other lines in use at the time of direction change. The NI 9402 does not have this problem. Four lines are used for DIO and the other four lines are used to control the direction. Direction changes can happen independently of data transfer. The NI 9401, one of the most popular DIO modules, uses all eight DIO lines and must switch modes to communicate line states. This mode change affects all DIO lines regardless of which ones are changing state.

SPI Bus Communication

The Serial Peripheral Interface bus, or SPI, is a standard 4-wire communication protocol set up for master/slave, full-duplex (two-way simultaneous) communication. The bus clock rates typically range from 1 MHz to 70 MHz. The SPI clock rate that CompactRIO hardware uses to communicate with C Series modules varies but usually operates around 10 MHz.

The common architecture of SPI bus modules contains a complex programmable logic device (CPLD) that, on the module side, controls the timing to and data collection from the ADC/DAC chips. On the chassis side, the CPLD communicates via SPI bus back to the FPGA. Based on the specific module used and slot location, the LabVIEW API knows how and where to communicate to the individual CPLD on the module. This is why new module support is added in new versions of LabVIEW.

SPI Bus Challenges

Regardless of front-end circuitry, each SPI module communicates to the FPGA in a CompactRIO chassis over a single dedicated SPI bus. This means that even though commands to the module (from multiple loops on a LabVIEW block diagram) or data retrieval (from modules with multiple ADCs) can happen in parallel, those commands must be interlaced through a single SPI bus. LabVIEW and the NI-RIO driver can properly prioritize multiple calls that come in at the same time, but if a call comes in while the previous call is still being handled, you may introduce unwanted jitter into your control or data acquisition loop. Another way to convey this concept is to say that I/O node calls to a single module from different parts of a LabVIEW FPGA program create a “hardware race condition.” This in no appreciable way limits the abilities of the module or a CompactRIO system, but it does require that you arbitrate I/O node calls within your program. The easiest way to do this is to keep all of the I/O node calls in the same loop, but you can also use sequence structures or semaphores. You can find a reference design for semaphore implementation in LabVIEW FPGA in the NI Developer Zone document [Semaphore Reference Design for LabVIEW FPGA](#).

As you continue to read about the different classifications of modules, note that all modules that use SPI communication must adhere to the same caveat of hardware arbitration. Simultaneous Modules

Modules that are simultaneous have one ADC per channel and acquire data with no appreciable skew between channels. The two subcategories of simultaneous modules, on-demand and delta-sigma, transfer data via the SPI bus and are subject to all of the specifications and challenges of other SPI bus modules.

On-Demand Conversion

NI C Series On-Demand Simultaneous Modules	
Model Number	Description
NI 9215	Analog Input, ± 10 V, 100 kS/s/ch
NI 9263	Analog Output, ± 10 V, 100 kS/s/ch
NI 9265	Analog Output, 0 to 20 mA, 100 kS/s/ch
NI 9219	Analog Input, Universal, 100 S/s/ch
NI 9222 ¹	Analog Input, ± 10 V, 500 kS/s/ch
NI 9223 ¹	Analog Input, ± 10 V, 1 MS/s/ch

¹These modules are simultaneous modules and can be programmed with FPGA I/O nodes, similar to other modules listed in this table. However, to run at higher speeds, you need to program them with the user-controlled I/O sampling API as outlined in the Data on Demand section below. The maximum sampling rates achievable when using both FPGA I/O nodes and the user-controlled I/O sampling API can be found in the "Maximum Sampling Rate" section of the user manual.

Table 6.1. Examples of Simultaneous Modules With On-Demand Conversion

On-demand modules, like the ones listed in Table 6.1, have few specific challenges when it comes to programming with LabVIEW FPGA. This makes them some of the easiest modules to program. The biggest caveat involves arbitration, which is shared by all modules that use SPI communication.

Data on Demand

Data on demand is less of a caveat and more of a feature. On-demand simultaneous C Series modules have the ability to return data when the I/O node is called at any interval down to the minimum conversion time as listed in the manual. This means that the acquisition can be clocked by external, irregular clocks. The Δt does not need to be a constant for an acquisition with an on-demand simultaneous module.

Pipelined Simultaneous Data or User-Controlled I/O Sampling



The NI 9223 User-Controlled IO Sampling.lvproj is in the NI Example Finder.

Some modules designed for high-speed measurements exceed the data throughput capabilities of the FPGA I/O node. In these situations, you can apply user-controlled I/O sampling functions to communicate with a module. These add complexity to the program but dramatically increase the bandwidth from the module.

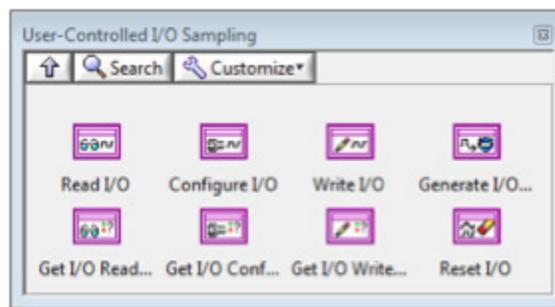


Figure 6.8. You can use the User-Controlled I/O Sampling palette for higher bandwidth communication to some modules.

When programming with the user-controlled sample method, you may find it easier to start with an existing example program that is shipped with LabVIEW. Figure 6.9 is the block diagram from the NI 9223 User-Controlled IO Sampling.lvproj, which you can find in the NI Example Finder.

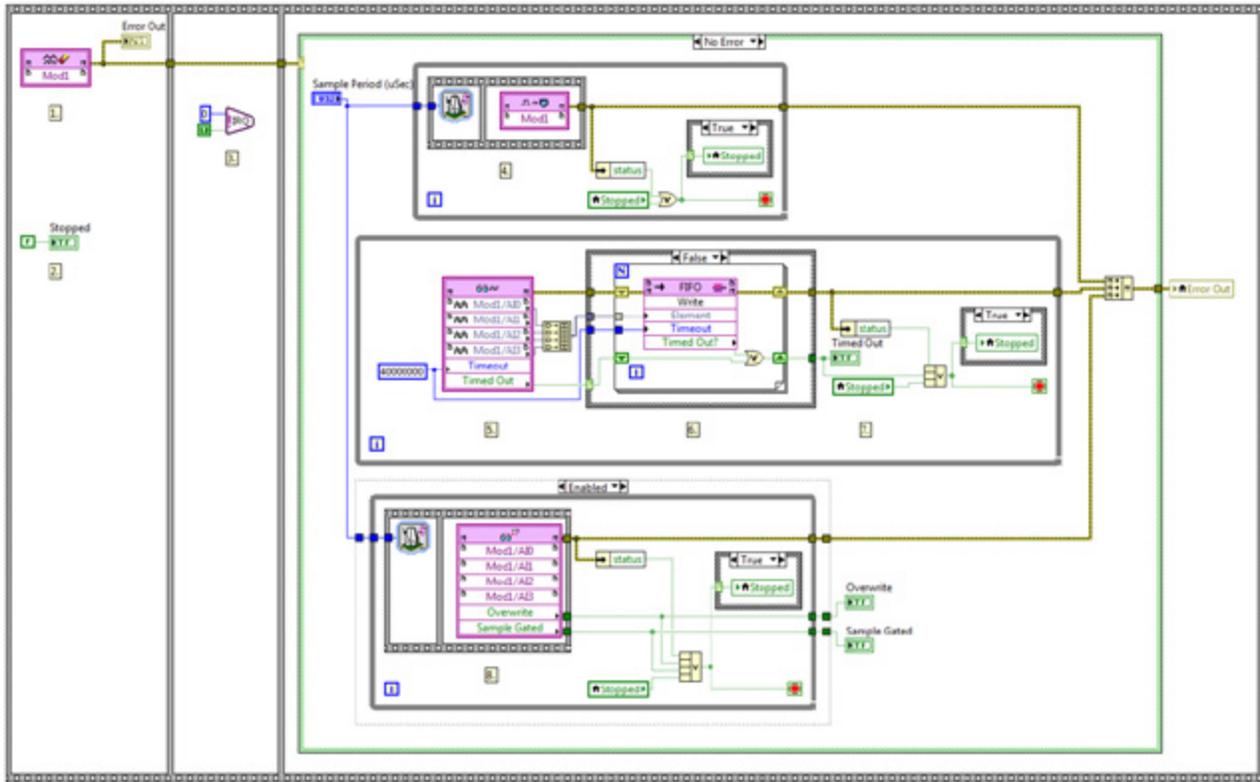


Figure 6.9. NI 9223 User-Controlled IO Sampling.lvproj Example Program

To program with the user-controlled I/O sample method, follow these steps that reference the example program in Figure 6.9.

Initialize the Process

1. Call the Reset I/O function. When this call completes, the module is ready to perform an acquisition using the user-controlled I/O sampling functions. You must call the Reset I/O function first to prepare the NI 9223 to use the other user-controlled I/O sampling functions.
2. Set the Stopped Boolean to false. This Boolean provides synchronization between the While Loops in the last sequence frame. If any loop stops, it causes the others to stop as well.
3. Use an interrupt to signal the host that the FPGA is ready to begin acquiring data and wait to start the acquisition until the host acknowledges it. This is necessary to ensure that the DMA FIFO has been started prior to acquiring data.

Loop 1

4. Call the Generate I/O Sample Pulse function to begin acquiring data. The rate at which this function is called determines the sample rate for the acquisition, so a loop timer is used to enforce the desired sample period.

Loop 2

5. Call the Read I/O function to read data acquired from the module. This function is configured to read a single sample from each channel on the module. Because this function waits for data to become available, a generous but noninfinite timeout is provided. In the case of the default 40 MHz top-level clock, the timeout is 1 second.
6. Write the acquired data into the DMA FIFO.
7. If a timeout occurred either while waiting for data from the module or waiting to write the data to the DMA FIFO, then report the timeout to the host and stop the VI.

Loop 3

8. Call the Get Read I/O Status function at the same rate you call the Generate I/O Sample Pulse function. This checks the status of every sample you acquire. If an “overwrite” or “sample gated” occur (output = true), then report the status to the host and stop the VI. In the example, this loop is encased in a diagram disable structure to make it easily removable from the application. Calling the Get Read I/O Status function is useful for development and debugging but not strictly necessary for deployment if the application does not feature variable timing in the application. In the Figure 6.9 example, you cannot obtain a sample gated status unless the top loop’s sample period is less than the minimum sample period supported by the module. However, this application could produce an overwrite status if the host VI cannot read from the DMA FIFO fast enough.

Delta-Sigma Modulation



The NI 9234 Getting Started.lvproj is in the NI Example Finder.

NI C Series Simultaneous Modules With Delta-Sigma Modulation	
Model Number	Description
NI 9229	Analog Input, ± 60 V, 50 kS/s/ch
NI 9239	Analog Input, ± 10 V, 50 kS/s/ch
NI 9233	Analog Input, IEPE, 50 kS/s/ch
NI 9234	Analog Input, IEPE, 51.2 kS/s/ch
NI 9235	Analog Input, 120 Ω $\frac{1}{4}$ bridge, 50 kS/s/ch
NI 9236	Analog Input, 350 Ω $\frac{1}{4}$ bridge, 50 kS/s/ch
NI 9237	Analog Input, $\frac{1}{4}$, $\frac{1}{2}$, full bridge, 50 kS/s/ch
NI 9225	Analog Input, 300 V _{rms} , 50 kS/s/ch
NI 9227	Analog Input, 5 A _{rms} , 50 kS/s/ch

Table 6.2. Examples of Simultaneous Modules With Delta-Sigma Modulation

Many C Series modules designed for high-speed, dynamic measurements use delta-sigma ($\Delta\Sigma$) converters. To better understand how these modules work, you must first know the fundamentals of delta-sigma modulation. The Greek letters delta and sigma are mathematical symbols for difference and sum, respectively. The modulation circuit of a delta-sigma converter compares the running sum of differences between the desired voltage input, V_{in} , and a known reference voltage, V_{ref} . The output from the comparator becomes a bitstream that is sent into a digital filter and a 1-bit DAC. Because of this negative feedback, the differences oscillate around 0 V, or ground. The digital filter effectively keeps track of how many times the difference is above 0 V, and, based on this count along with the reference voltage, you can determine the input voltage. This modulation loop runs at a much higher frequency than the actual output frequency of the converter.

C Series modules with delta-sigma converters feature an oversample clock that runs the modulation circuitry. Oversample clocks, which run at 12 MHz or faster, affect timing, synchronization, and programming paradigms. The following list provides insight into the specific challenges of C Series modules that use delta-sigma modulation.

- **Need a Sync Pulse to Reset**—Oversample clocks need to be “reset” before they are used. This is why there is a LabVIEW FPGA I/O node to send a “start” event to the module.
- **Time to Data Ready Is Non-Zero**—The time between the “start” event and data availability is specified as “time to first data.” This time may vary slightly between different delta-sigma-based modules and greatly between modules of other types. On-demand modules have a “time to first data” of zero. You can find documentation on how to align the data sets in [KnowledgeBase 4DAEUNNQ: How Do I Compensate for Different Group Delays With C Series Modules in LabVIEW FPGA?](#) and [KnowledgeBase 53CHLD6C: What Is the Best Method to Synchronize Two Different DSA Modules in LabVIEW FPGA?](#)
- **Sample Rates Are Discrete and Specific**—Because of the oversample clock and the digital filter, delta-sigma modules can run only at discrete sample rates. These sample rates are a function of a divisor and the oversample clock. This is why the “rate” input for a delta-sigma module is an enumerated data type of predetermined sample rates. If you try to input a sample rate that is not supported, it is rounded to the next highest available sample rate.
- **Minimum Sample Rates Are Greater Than 1 kHz**—The minimum sample rate for delta-sigma modules is often over 1 kHz. Use averaging, filtering, or some form of decimation to further reduce your data set beyond the rate the digital filter on the module outputs.
- **No Irregular or External Clocks**—Delta-sigma modules cannot report data “on demand” and thus do not work with irregularly timed I/O node calls because the iterative process must complete a large number of loops before returning accurate data. The I/O node for a delta-sigma module always blocks for the exact (Δt) for which the module has been set to acquire. An I/O node call at an interval less than Δt must wait until the full sample period is complete. This gives the oversample circuitry enough time to calculate an accurate value. To compensate for this, implement a resampling algorithm on the FPGA before processing your data or using it in a control loop.
- **Physically Share Oversample Clock to Synchronize $\Delta\Sigma$ Modules**—Two delta-sigma modules that need to be synchronized must share the same oversample clock. To synchronize modules, the oversample clock from one module must be exported from the right-click properties menu as the “source” module in the LabVIEW project. It must be imported from the same menu of the other “client” modules. Any module can be a “source” or a “client” module—it is up to the programmer’s discretion. Remember, changes made to a module property window from the project view cause a recompile and cannot be changed at run time.

You can see many of these caveats in the block diagram of the example program NI 9234 Getting Started.lvproj shown in Figure 6.10.

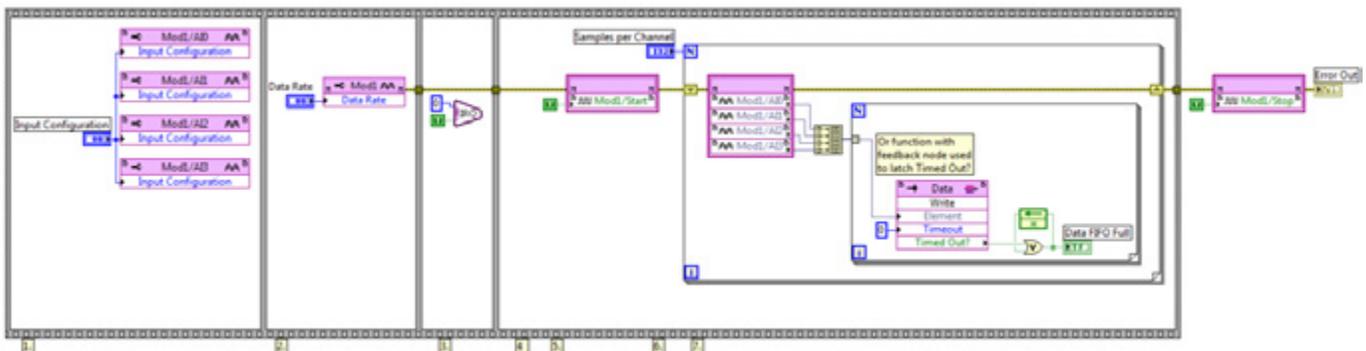


Figure 6.10. Block Diagram From NI 9234 Getting Started.lvproj

You can find example programs for all of the delta-sigma modules in the NI Example Finder.

Multiplexed

One of the more expensive components of a module is the ADC. By using a multiplexer, also known as a mux, to route multiple channels through a single ADC, multiplexed modules offer higher channel counts at lower per channel prices than simultaneous modules.

Before learning how to program these modules, you need to know some specification-level details. First, the sample rates are often listed as the total rate of all channels put together, also known as an aggregate rate. From the module hardware standpoint, all channels selected must run at the same rate (aggregate rate divided by number of channels), but from the program standpoint, you can remove samples from select channels with FPGA processing. Second, it is important to note that there is an interchannel delay, or skew, between all channels in a multiplexed module. You can implement processing in the FPGA to compensate for this skew via shifting or data resampling, but most systems that incorporate multiplexed modules are not impacted by this small offset. If hardware-based phase alignment is important to your system, you should select a module with multiple ADCs.

You can choose from two main subsets of multiplexed modules: high speed and low speed.

High Speed

High-speed multiplexed modules implement a double pipeline to increase the throughput of data to the chassis. With a double pipeline, the first valid data cannot be returned until the process has run two complete iterations. Once the first two iterations have run to “prime” the pipeline, the subsequent iterations begin to yield valid data. When using an I/O node to sample channels, the pipeline is automatically managed by the FPGA I/O node, and the channels within the FPGA I/O node are sampled in numerical order regardless of the order they appear in the node. If the first two channel requests in the FPGA I/O node do not match the two channel requests stored in the module pipeline, there is a delay before the first channel sample occurs. This delay is caused by the FPGA I/O node automatically updating the module channel sample pipeline, which takes two channel sample cycles. This situation could happen if you have I/O nodes addressing the same module from different parts of the block diagram. For example, if in an “init” case of a Case structure, you read channels 0, 1, and 2 from an I/O node and then in the “acquire” case, you read from channels 5, 6, and 8, you incur the pipeline updating penalty because the module is already primed for channels 0, 1, and 2 and needs to flush the pipeline. If this delay causes problems, the workaround is to acquire all of the channels in the same step and place data from channels 5, 6, and 8 into a FIFO to be called upon later.

NI C Series High-Speed Multiplexed Modules	
Model Number	Description
NI 9205	Analog Input, ± 10 V, 250 kS/s (aggregate rate)
NI 9206	Analog Input, ± 10 V, 250 kS/s (aggregate rate)
NI 9201	Analog Input, ± 10 V, 500 kS/s (aggregate rate)
NI 9221	Analog Input, ± 60 V, 800 kS/s (aggregate rate)
NI 9203	Analog Input, ± 20 mA, 200 kS/s (aggregate rate)

Table 6.3. Examples of High-Speed Multiplexed Modules

I/O Sample Method for High-Speed Multiplexed Modules



The NI 9205 Basic I/O Sample Mode.lvproj is in the NI Example Finder.

Some high-speed multiplexed modules, such as the NI 9205 and NI 9206, have an alternate method for programming. This method, referred to as the “I/O sample method,” is more difficult to implement but takes up less FPGA space and provides an easier input into DMA nodes on a LabVIEW block diagram. This lower-level access to the module does not account for the double-pipeline architecture, so you must explicitly discard the first two data points returned from the I/O sample method because they are invalid. The step that adds the most difficulty is building the channel array to feed into the sample method. You can find an example program for this method, NI 9205 Basic I/O Sample Mode, in the NI Example Finder.

Low Speed

Low-speed modules do not require the same amount of bandwidth as high-speed modules and do not implement a pipeline. This makes the LabVIEW implementation straightforward. You face SPI bus module caveats when you work with low-speed multiplexed modules.

NI C Series Low-Speed Multiplexed Modules	
Model Number	Description
NI 9211	Thermocouple, 14 S/s (aggregate rate)
NI 9213	Thermocouple, 1200 S/s (aggregate rate)
NI 9214	Thermocouple, 1088 S/s (aggregate rate)
NI 9217	RTD, 400 S/s (aggregate rate)
NI 9207	Analog Input Combo V/I, 500 S/s (aggregate rate)
NI 9208	Analog Input, ± 20 mA, 500 S/s (aggregate rate)

Table 6.4. Examples of Low-Speed Multiplexed Modules

SPI Digital

Digital modules that feature more than eight lines exceed the number of pins allowed for direct FPGA communication and, thus, communicate over the SPI bus. These modules operate with a convert pulse, and they latch inputs and update outputs simultaneously with each convert. As with any other SPI bus module, all of the lines on a 32-channel digital I/O module are routed through the same communication lines to the backplane. You should control the I/O node calls to SPI bus modules with dataflow to prevent simultaneous calls resulting in jitter. These calls include normal I/O calls as well as commands to change line direction from input to output.

Synchronizing Modules

Some applications, such as vibration or sound measurement, require a high level (sub 100 nS) of synchronization between channels. This section discusses timing and synchronization of both delta-sigma-based modules and scanned (SAR) modules. Any NI C Series I/O module that is not classified as delta sigma is classified as SAR.

Synchronizing Delta-Sigma Modules



The NI 9205 Basic I/O Sample Mode.lvproj is in the NI Example Finder.

To synchronize delta-sigma modules in CompactRIO hardware, you need to physically share the oversample clock and start triggers between all of the modules. You can find the phase match specification between channels and between modules in the specifications section of the manual for many modules. Delta-sigma-based modules may be synchronized even if they are not the same model number.

1. Select one of the modules, “the master,” to export the clock to the backplane. The other modules are set to import this clock from the backplane. Whichever module you select as the master overrides both the timebase and available rates. For example, if you want 51.2k rates in the system, select the NI 9234 or NI 9232. If you want 50k rates, select the NI 9237. You modify import/export properties from the properties window accessed from the right-click menu of the module in the LabVIEW Project Explorer. Import/export settings cannot be set programmatically because this information is compiled.

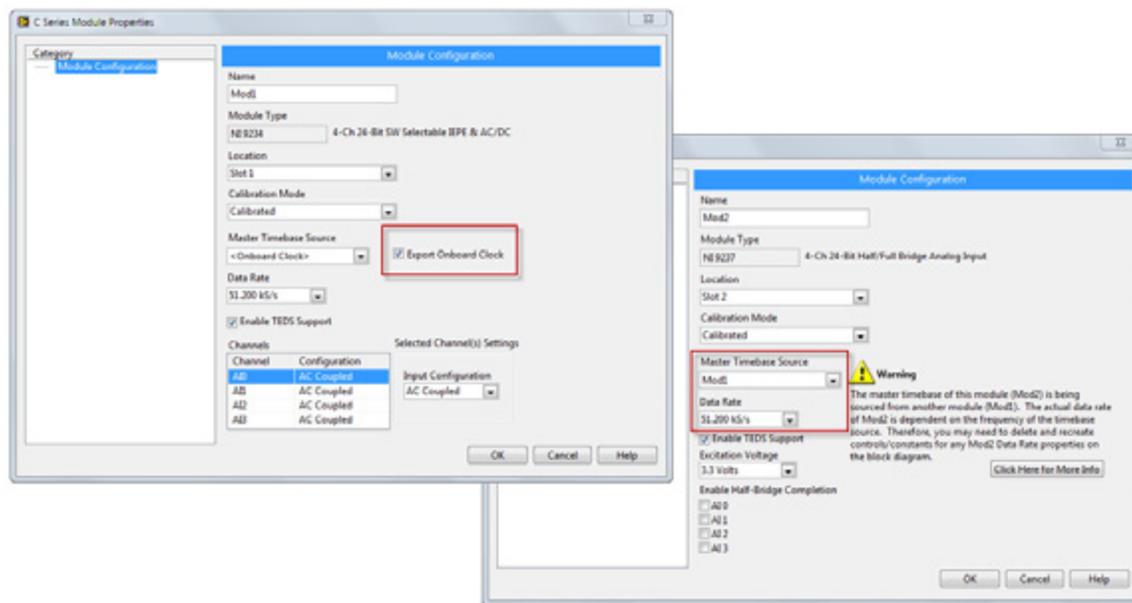


Figure 6.11. Export the clock from the chosen master module (left) and then select it from all subsequent modules to be synchronized (right).

2. On the block diagram, create a property node for each I/O module and use the Data Rate enum to specify the rate, as shown in Figure 6.12. Note that even though the I/O modules share the same sampling rate, you must create a unique Data Rate enum for each property node (right-click on the property node for each module and select “create constant”). This ensures that the enum integer is properly matched to the expected rate for the specific I/O module.
3. Create a Start Trigger for each I/O module and place them into the same I/O node. This ensures the start triggers are properly routed.

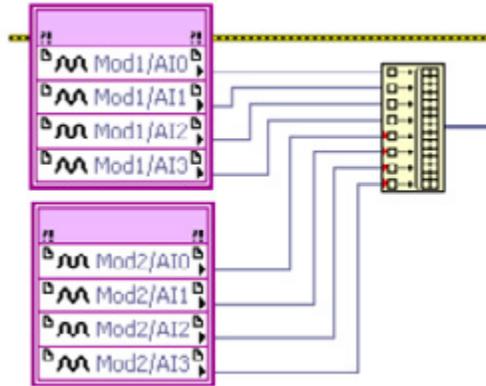


Figure 6.13. FPGA I/O Nodes for Hybrid Applications
(Delta-Sigma and SAR Modules)

An application becomes more complicated when it requires multirate synchronization between delta-sigma and SAR modules. Since these modules have different timebases, they cannot share a clock when they are separated into two or more loops executing at multiple rates. The best option is to separate them into multiple loops and then expect some drift between the timebases over time, in addition to a phase offset caused by the varying startup times.



LabVIEW example code
is provided for this section.

CHAPTER 7

Adding I/O to the CompactRIO System

Adding I/O to CompactRIO

NI products based on the LabVIEW reconfigurable I/O (RIO) architecture are increasingly adopted for system-level applications that require high-channel counts, intensive processing, and distributed I/O. Adding RIO expansion I/O to the NI RIO product offering enables a 1:N system topology with one controller and many FPGA I/O nodes for flexible, high-channel-count systems that can perform both distributed control and centralized processing.

NI offers three types of expansion chassis systems for expanding your I/O. Each expansion chassis features an FPGA and is based on the same C Series module architecture. The main differences between expansion I/O options stem from the choice of bus, with each bus being best suited for a subset of expansion I/O applications. The NI I/O expansion chassis systems include the following:

- MXI-Express RIO
- Ethernet RIO
- EtherCAT RIO



Figure 7.1. Expand your CompactRIO I/O with NI expansion chassis.

Each expansion chassis has unique capabilities. Choose an expansion chassis depending on your application requirements. Table 7.1 provides a quick comparison of the three different types of expansion chassis.

	MXI-Express RIO	Ethernet RIO	EtherCAT RIO
	NI 9157, 9159	NI 9148	NI 9144
Slots	14	8	8
FPGA	Virtex-5 (LX85 or LX110)	Spartan 2M	Spartan 2M
Network Topology	Daisy chain	Same as Ethernet	Daisy chain
Distance	7 m between nodes	100 m before repeater	100 m before repeater
Multichassis Synchronization	FPGA-based DIO	FPGA-based DIO	Implicit in bus operation
Communication Jitter	<10 μ s	No Spec	<1 μ s
Bus Throughput	250 MB/s	100 Mbit/s	100 Mbit/s
API Support	FPGA Host Interface	FPGA Host Interface/Scan Engine	FPGA Host Interface/Scan Engine
Host	Windows/Real-Time	Windows/Real-Time	Real-Time Only

Table 7.1. Comparison of RIO Expansion Chassis

MXI-Express RIO

Featuring a high-performance Xilinx Virtex-5 FPGA and a PCI Express x1 cabled interface, an NI 9157/9159 MXI-Express RIO 14-slot chassis for C Series I/O expands the RIO platform for large applications requiring high-channel counts and a variety of signal conditioning and custom processing and control algorithms. You can use these MXI-Express RIO expansion chassis with high-performance NI cRIO-9081/9082 CompactRIO systems in addition to industrial controller and PXI systems. When using a MXI-Express RIO expansion chassis, you interface to the expansion chassis I/O within your main controller VI using LabVIEW FPGA Host Interface functions.

NI cRIO-9081/9082 Systems



NI 9157/9159 MXI-Express RIO Chassis



MXI-Express Bus

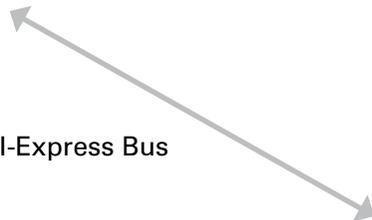


Figure 7.2. A MXI-Express RIO Chassis Paired With an NI cRIO-9081/9082 High-Performance CompactRIO System

Ethernet RIO

When expanding your I/O using the standard Ethernet protocol, you can use the NI 9148 Ethernet RIO expansion chassis. Programmers can take advantage of the existing network infrastructure such as switches and routers. Although full duplex switch networks eliminate packet collisions, switches introduce jitter, and you should use general Ethernet only in applications that do not require deterministic communication. If you need synchronization between the local I/O and expansion I/O, see the EtherCAT RIO section for more information.



Figure 7.3. The NI 9148 Ethernet RIO Expansion Chassis

When using the Ethernet RIO expansion chassis, the main controller is responsible for running the real-time control loop using I/O from its own chassis in addition to the I/O from one or more Ethernet RIO chassis. The expansion chassis provides expansion or distributed I/O for the main controller.

The Ethernet RIO expansion chassis works with both LabVIEW FPGA and the Scan Engine. If you use LabVIEW FPGA with the expansion chassis, you can embed decision-making capabilities to quickly react to the environment without host interaction. The FPGA can also offload processing from the main controller by conducting inline analysis, custom triggering, and signal manipulation.

When using LabVIEW FPGA, you should create a regular While Loop or a Timed Loop with a lower priority to handle the communication since Ethernet is nondeterministic (see Figure 7.4). This allows your control task to run deterministically and reliably because it is not affected by the possibly high-jitter I/O device. When using LabVIEW FPGA, you interface to the I/O within your real-time VI using the FPGA Host Interface functions.

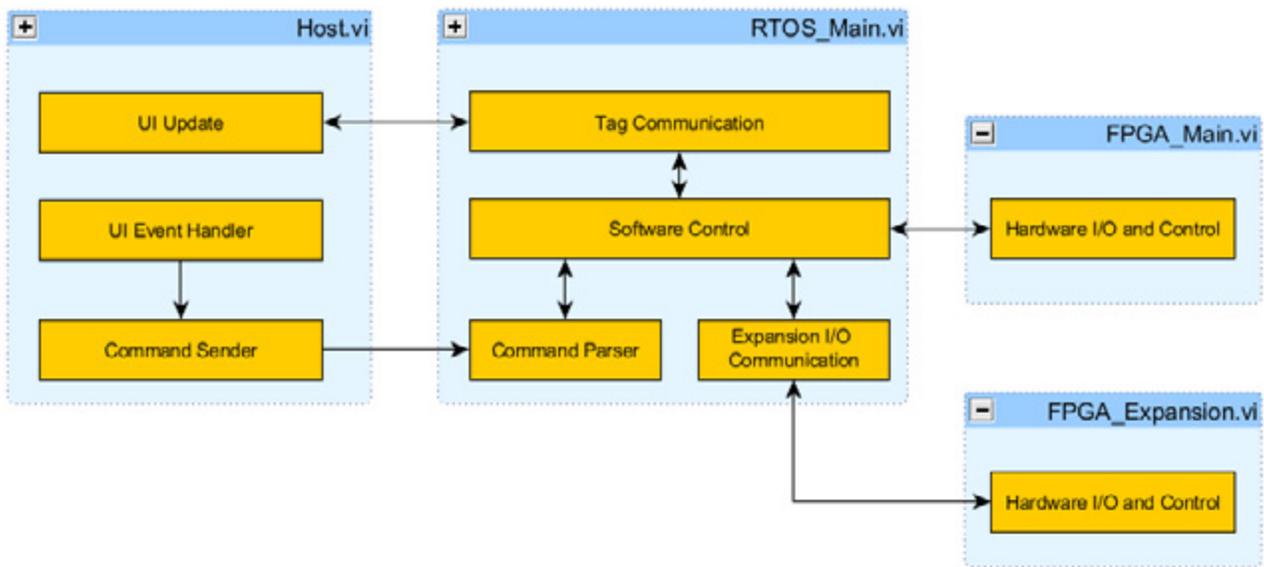


Figure 7.4. Add a new process to handle the I/O expansion task when using LabVIEW FPGA Interface Mode.

The Ethernet RIO expansion chassis also works with the Scan Engine. You have the option to choose Scan Mode or FPGA Interface Mode when adding the Ethernet RIO chassis to your LabVIEW project. When using Scan Mode, your design diagram might look like Figure 7.5, where you have all of your system I/O accessible from the Scan Engine. When using Scan Mode, you interface to the I/O within your real-time VI using the Scan Engine I/O variables.

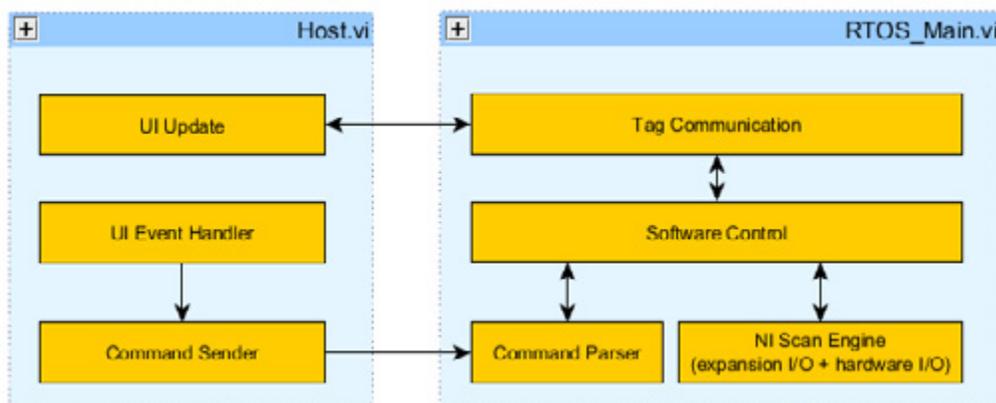


Figure 7.5. You can use the NI Scan Engine to handle I/O with the NI 9148 Ethernet RIO chassis.

To get started with the NI 9148 Ethernet RIO expansion chassis, see the NI Developer Zone tutorial [Getting Started With the NI 9148 Ethernet RIO Expansion Chassis](#).

EtherCAT RIO

In certain applications, the main I/O and expansion I/O systems require tight synchronization—all of the inputs and outputs must be 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 systems with deterministic Ethernet technology using the NI 9144 expansion 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.

Host Computer

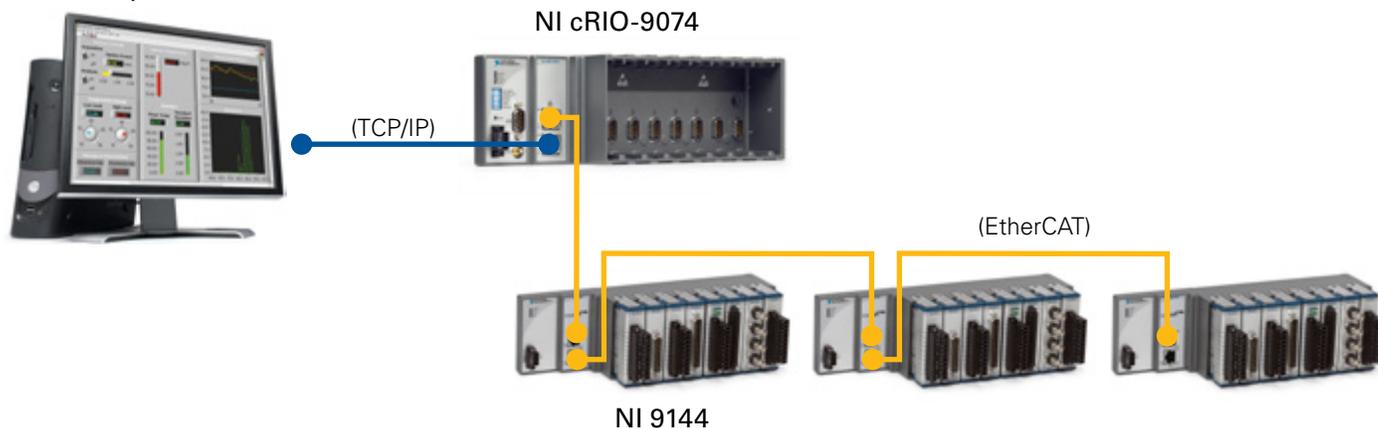


Figure 7.6. The CompactRIO hardware architecture expands time-critical systems using the NI 9144 deterministic Ethernet chassis.

Tutorial: Accessing I/O Using the EtherCAT RIO Chassis

The following step-by-step tutorial provides instructions for configuring your EtherCAT system and communicating I/O with both the Scan Engine and LabVIEW FPGA. This example uses an NI cRIO-9074 integrated system, but any CompactRIO system with two Ethernet ports can work with EtherCAT.

Step 1: Configure the Deterministic Expansion Chassis

1. To enable EtherCAT support on the CompactRIO controller, you must install the NI-Industrial Communications for EtherCAT driver on the host computer. This driver is on the CD included with the NI 9144 chassis and on ni.com for free download.
2. 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 using standard Ethernet cables. 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.

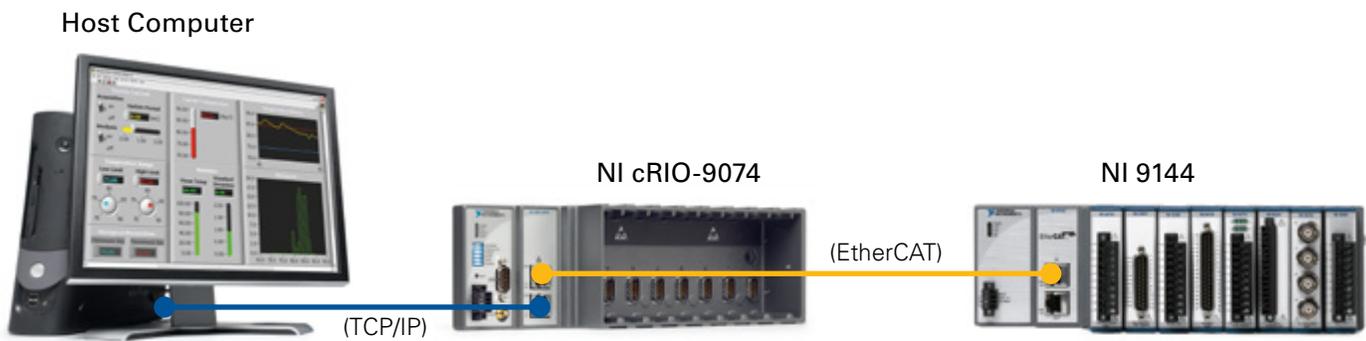


Figure 7.7. Hardware Setup for Ethernet RIO Tutorial

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

3. Launch MAX and connect to the CompactRIO controller.
4. In MAX, expand the controller under **Remote Systems** in the **Configuration** pane. Right-click **Software** and select **Add/Remove Software**.

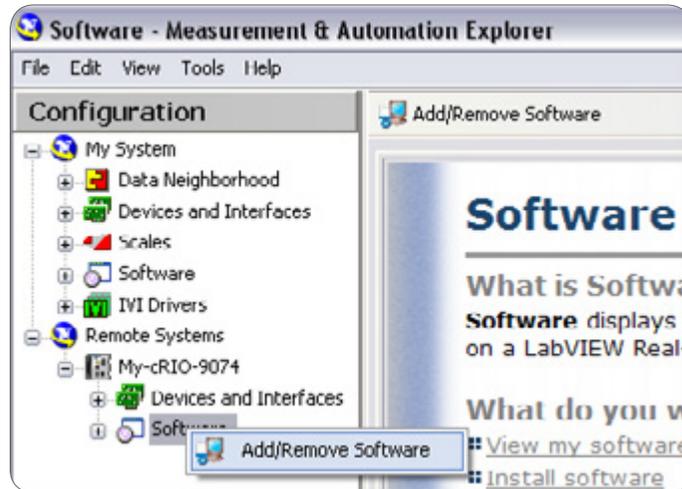


Figure 7.8. Install the appropriate software in MAX.

5. 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 **NI-Industrial Communications for EtherCAT**. The required dependencies are automatically checked. Click **Next** to continue installing software on the controller.
6. Once the software installation is finished, select the controller under Remote Systems in the Configuration pane. Navigate to the **Network Settings** tab in the bottom right portion of the window. In the Network Adapters window, select the secondary Ethernet Adapter (the one that does not say primary). Next to Adapter Mode, select **EtherCAT** in the pull-down box and hit the **Save** button at the top of the window.

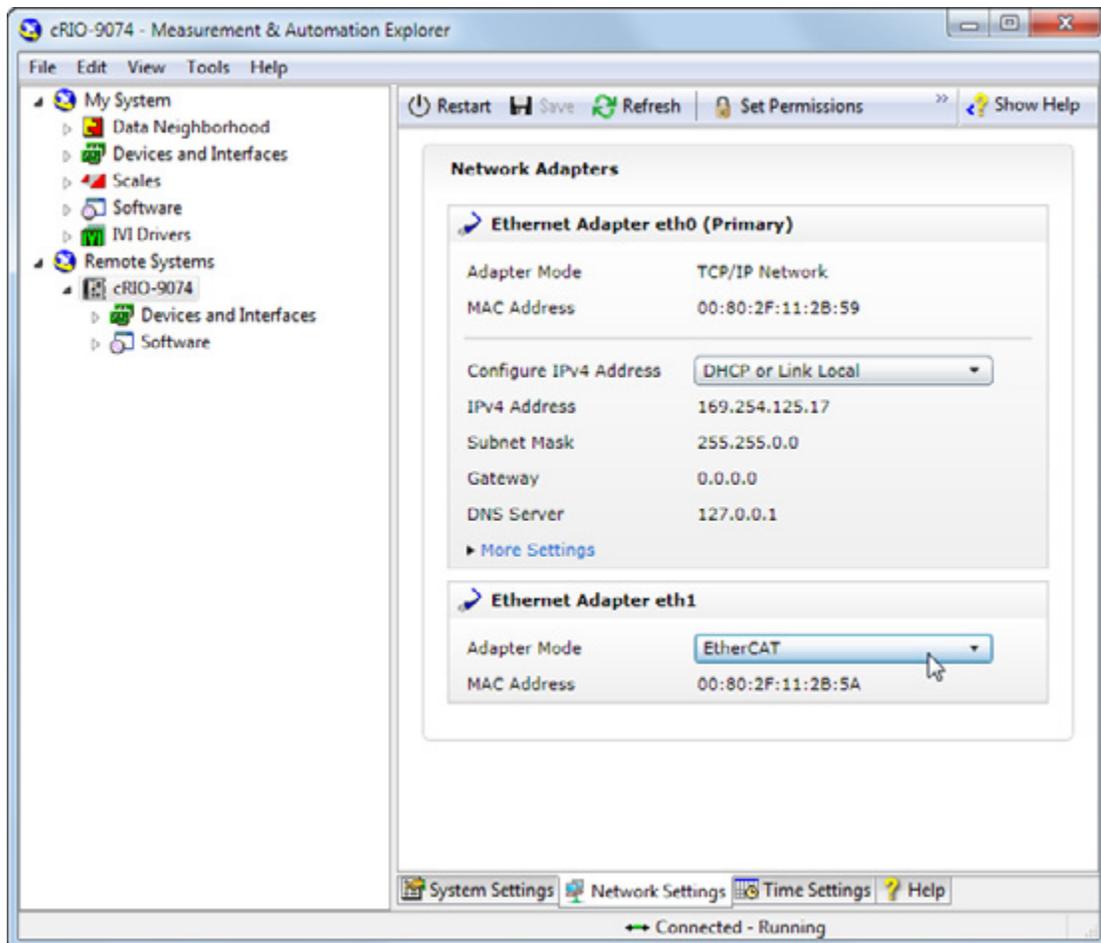


Figure 79. Select EtherCAT as the mode for the second Ethernet port of the CompactRIO controller.

Step 2: Add Deterministic I/O (Option 1—Scan Engine)

1. In the LabVIEW Project Explorer window, right-click on the CompactRIO controller and select **New»Targets and Devices**.
2. In the Add Targets and Devices dialog window, expand the category **EtherCAT Master Device** to autodiscover the EtherCAT port on the master controller. Select your EtherCAT device and 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.

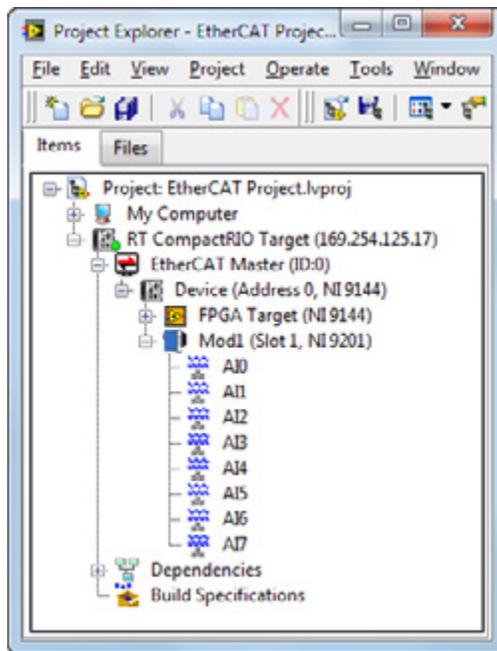


Figure 7.10. The LabVIEW project lists the master controller, NI 9144 chassis, and I/O modules.

3. By default, the I/O channels show up in the project as Scan Engine I/O variables. The Scan Engine automatically manages I/O synchronization so that all modules are read and updated at the same time once each **scan cycle**.

Note: 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.

Step 2: Add Deterministic I/O (Option 2—LabVIEW FPGA)

1. In the LabVIEW Project Explorer window, right-click on the CompactRIO controller and select **New»Targets and Devices**.
2. In the Add Targets and Devices dialog window, expand the category **EtherCAT Master Device** to autodiscover the EtherCAT port on the master controller. Select your EtherCAT device and 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. By default the I/O channels show up in the project as Scan Engine I/O variables.
3. Right-click the EtherCAT device in the LabVIEW project and select **New»FPGA Target**. You can now create a LabVIEW FPGA VI to run on the EtherCAT target. By default the chassis I/O is added to the FPGA target but not the I/O modules. To program the C Series module I/O using LabVIEW FPGA, drag the C Series modules from the EtherCAT device target onto the FPGA target in the LabVIEW Project Explorer window.

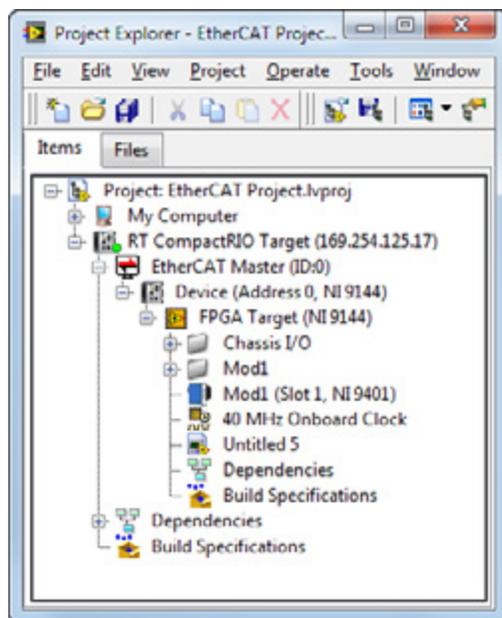


Figure 7.11. Drag C Series I/O modules under the FPGA target to access them with LabVIEW FPGA.

4. Create an FPGA VI under the EtherCAT device target and program using the EtherCAT I/O. The Figure 7.12 example uses the FPGA on an EtherCAT chassis to output a PWM signal.

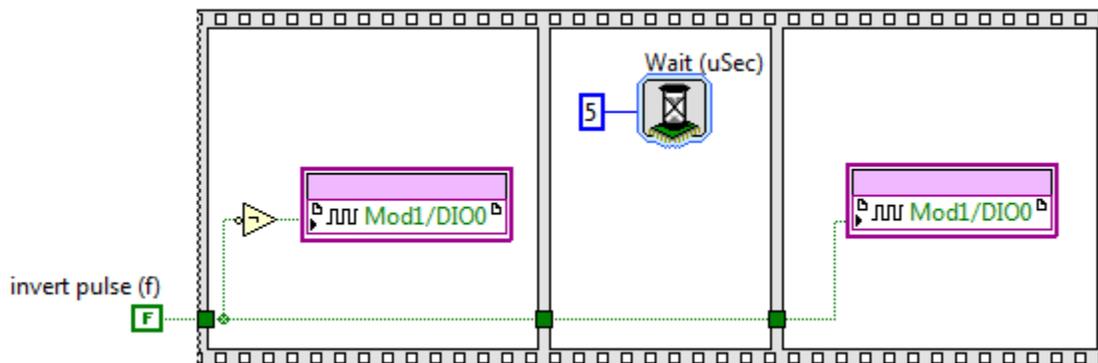


Figure 7.12. Program the EtherCAT FPGA using LabVIEW FPGA.

Note: There is a limit to the number of user-defined I/O variables that you can create in FPGA Interface Mode. The NI 9144 can hold a total of 512 bytes of input data and 512 bytes of output data for both I/O variables in Scan Mode and user-defined I/O variables in FPGA Interface Mode. For example, if you are using four 32-channel modules in Scan Mode and each channel takes up 32 bits of data, then Scan Mode I/O variables are using 256 bytes of input data. With the remaining 256 bytes of input data, you can create up to 64 input user-defined I/O variables (also of 32-bit length) in FPGA Interface Mode.

5. The next step is creating an interface that allows the Real-Time VI to communicate to the FPGA VI on the EtherCAT expansion chassis. For example, you need the user to be able to control the pulse width of the PWM signal, along with the invert pulse input. To transfer data to or from an FPGA on an EtherCAT chassis to a Real-Time VI, you use a new mechanism called user-defined I/O variables. To create a user-defined I/O variable, right-click the EtherCAT device target and select **New»User-Defined Variable**.

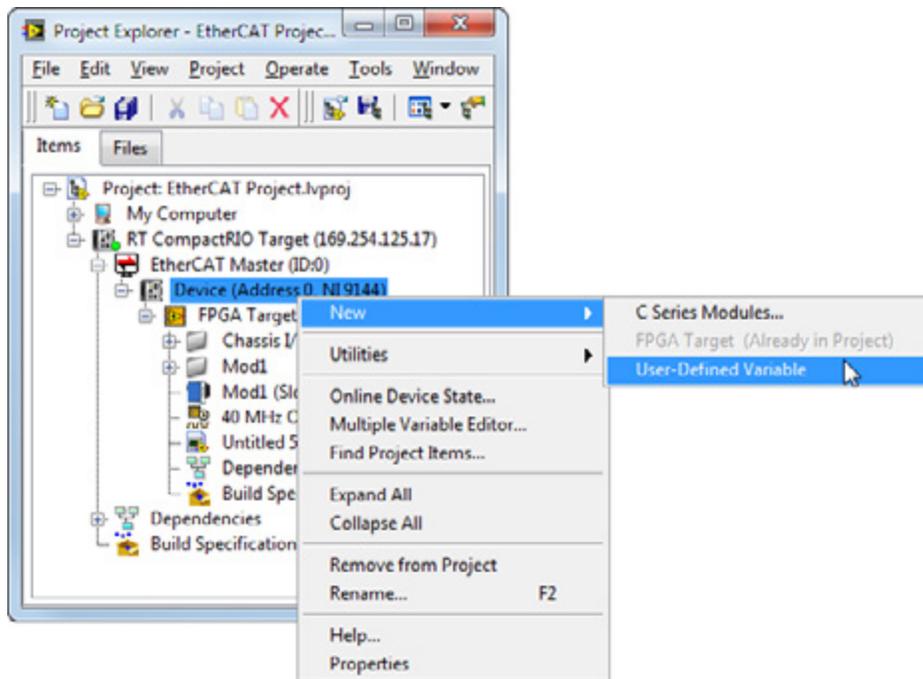


Figure 7.13. Create user-defined I/O variables to communicate between a Real-Time VI and the EtherCAT FPGA VI.

6. In the Properties dialog box, specify the variable name, data type, and direction (FPGA to host or host to FPGA). Drag and drop the user-defined I/O variables onto your FPGA VI.

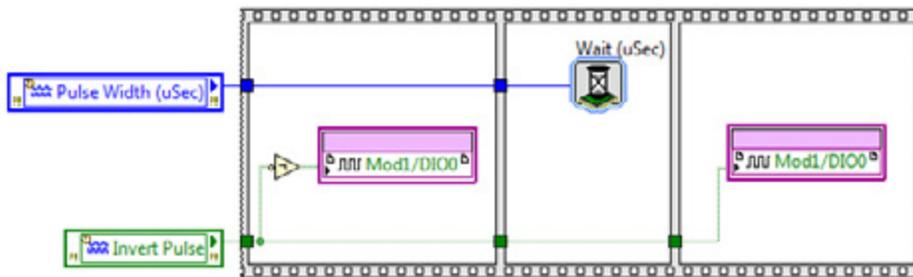


Figure 7.14. Drag and drop user-defined I/O variables onto the FPGA VI.

Note: These user-defined I/O variables transfer single-point data to and from the master controller at each scan cycle, so they are best used for passing processed data from the NI 9144 FPGA.

7. Drag and drop the same variables onto your Real-Time VI to communicate between the two targets.

CHAPTER 8

Communicating With Third-Party Devices

This section examines different methods for communicating with third-party devices from a CompactRIO controller. Because CompactRIO has built-in serial and Ethernet ports, it can easily work with Modbus TCP, Modbus Serial, and EtherNet/IP protocols. You can find information on using the Ethernet port to communicate to an HMI in [Chapter 4: Best Practices for Network Communication](#).

When interfacing to a third-party device in LabVIEW Real-Time using RS232, USB, or Ethernet ports, you should consider adding a separate process for handling the communication since these protocols are nondeterministic. This allows the control task to run deterministically and reliably because it is not affected by the possibly high-jitter I/O device.

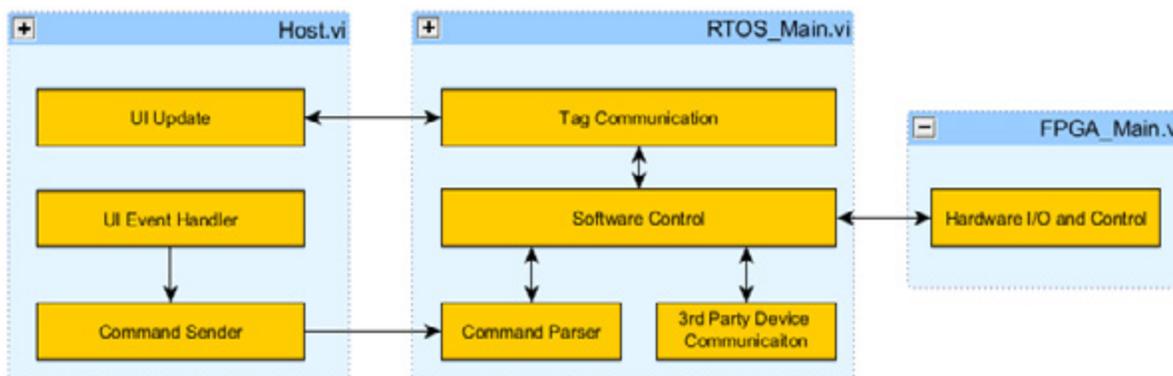


Figure 8.1. Add a separate process for handling any nondeterministic communication to a third-party device over serial or Ethernet.

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. Though 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. Several protocols 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 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.

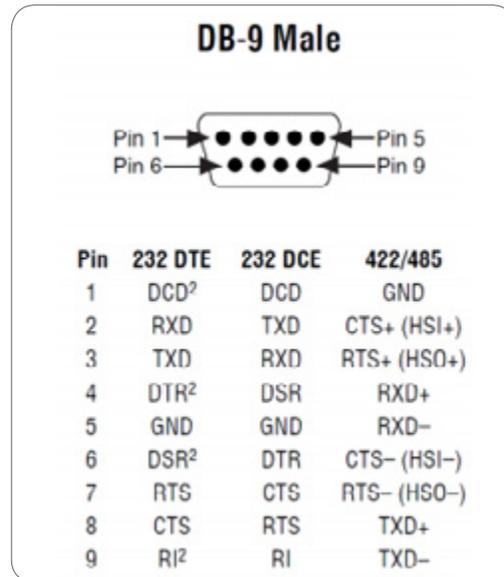


Figure 8.2. D-SUB 9-Pin Connector Pinout

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

RS232 ports feature two standard pinouts. 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 8.1. Selecting a Cable to Run Between Different RS232 Ports

Loopback Testing

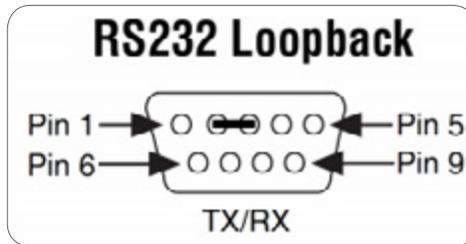


Figure 8.3. RS232 Loopback Wiring

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 at **Data Communication»Protocols»Serial**.

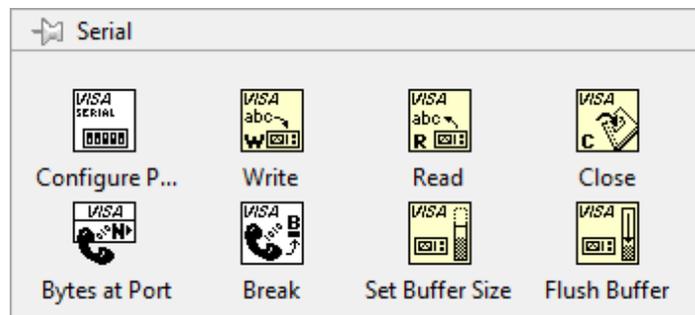


Figure 8.4. VISA Functions

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\smp1ser1.lib 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.

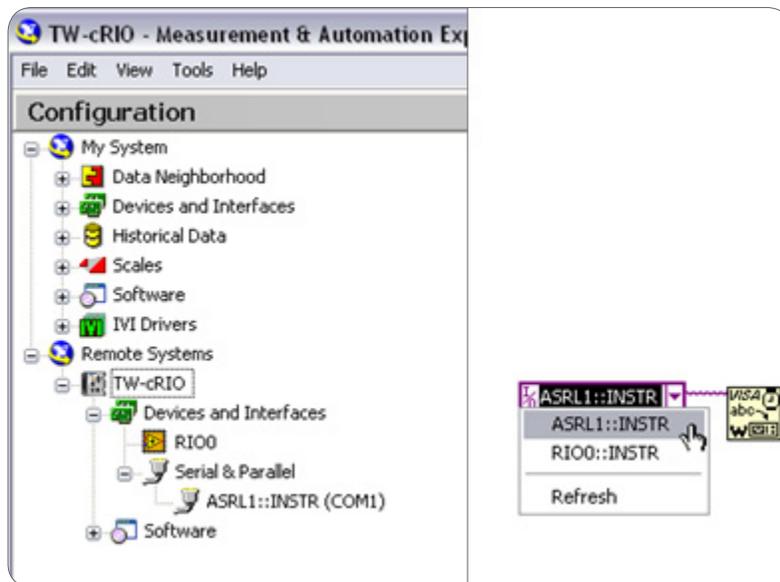


Figure 8.5. You can directly browse to the VISA resource name from the VISA resource control in LabVIEW or from MAX.

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 controller, you can directly browse to the VISA resource name from the VISA resource control in LabVIEW or from MAX.

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.

Though 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 nondeterministic. 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.

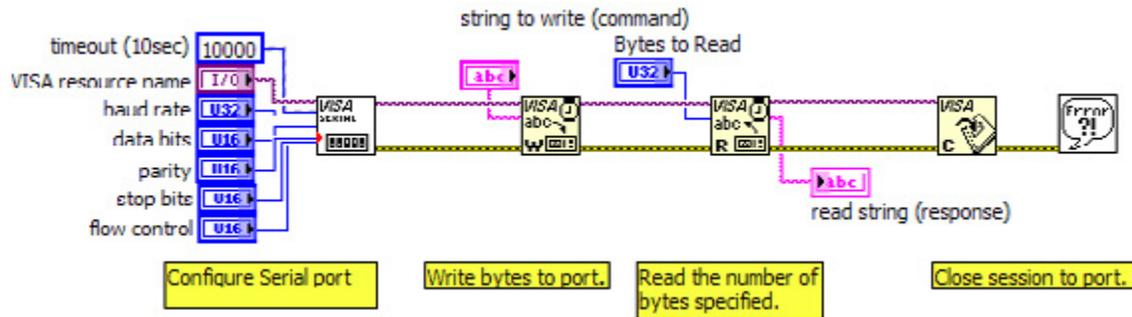


Figure 8.6. Simple Serial Communication Example

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 8.7 shows an example of using a property node to check the number of bytes available at the port before reading.

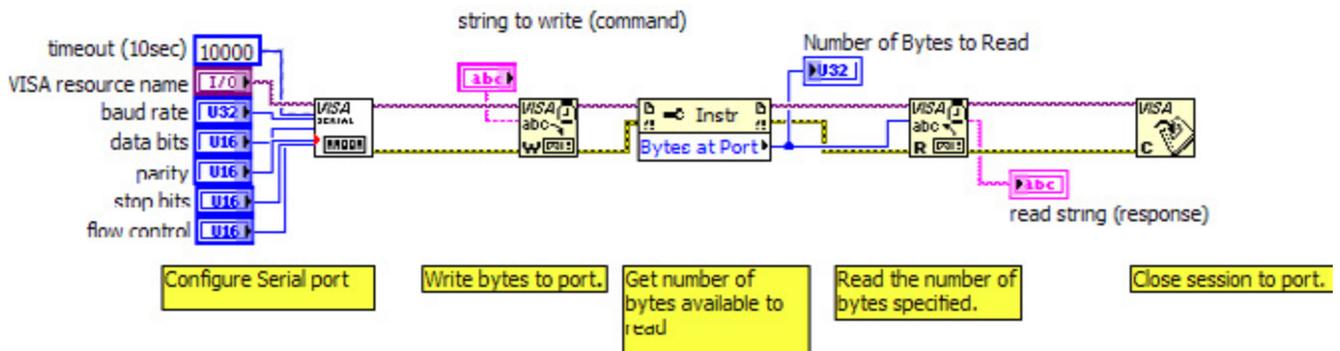


Figure 8.7. Using a VISA Property Node to Read the Number of Bytes

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, NI has worked with major measurement and control vendors to provide a library of ready-to-run instrument drivers for more than 10,000 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 in two 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 \(ni.com/idnet\)](http://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.

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.

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 modules are accessed directly from the FPGA using LabVIEW FPGA or from the Scan Engine.

Using NI 987x Serial Modules With Scan Mode

To enable support for these modules using the Scan Engine, add NI-Serial 9870 and 9871 Scan Engine Support to your CompactRIO target in Measurement & Automation Explorer (MAX). Once the **NI-Serial 9870 and 9871 Scan Engine Support** and your CompactRIO target are correctly configured, the ports of an NI 987x appear in MAX when the **Serial & Parallel** branch is expanded. ASRL1 is your built-in serial port, and ASRL 2 through 5 are the ports of an NI 987x.

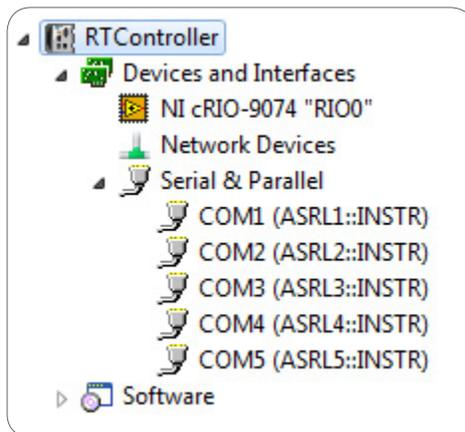


Figure 8.8. Access your NI 987x serial ports in MAX by installing Scan Engine support.

In LabVIEW, place a VISA Configure Serial Port VI or a Visa Property Node to configure the settings of your individual ports.

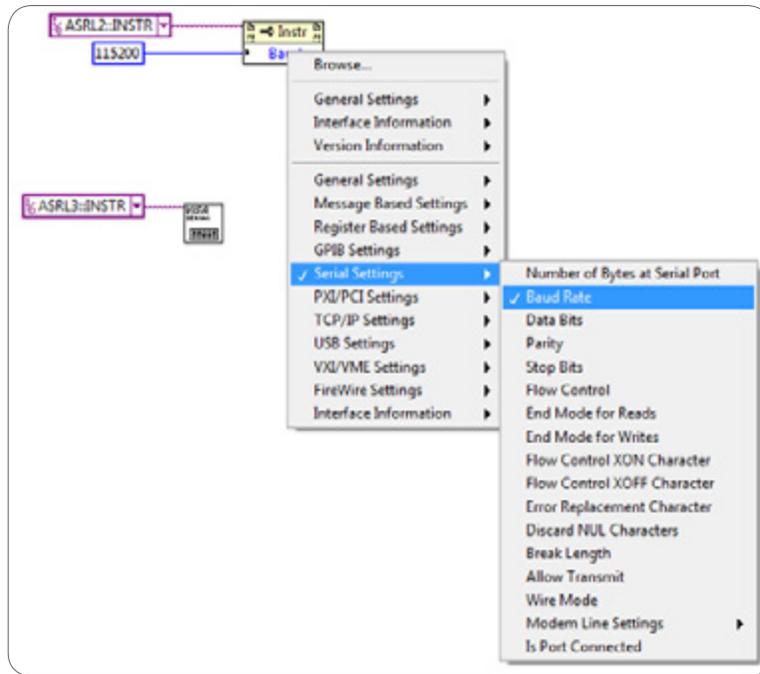


Figure 8.9. Program an NI 987x with the VISA API when using Scan Mode.

Once you have set the serial port to match that of the connected device, use the VISA API as you would when using any other VISA device.

Note: This maximum nonstandard baud rate for the NI 9871 module (3.6864 Mbit/s) can be achieved only while using the FPGA interface. The maximum baud rate while using either module with the RIO Scan Interface is 115.2 kbit/s.

Using NI 987x Serial Modules With LabVIEW FPGA

You also can access NI 987x serial modules with LabVIEW FPGA. See the Figure 8.10 example titled NI 987x Serial Loopback.lvproj in the NI Example Finder. This example demonstrates communication to the NI 987x serial port and streaming serial data to a real-time VI.

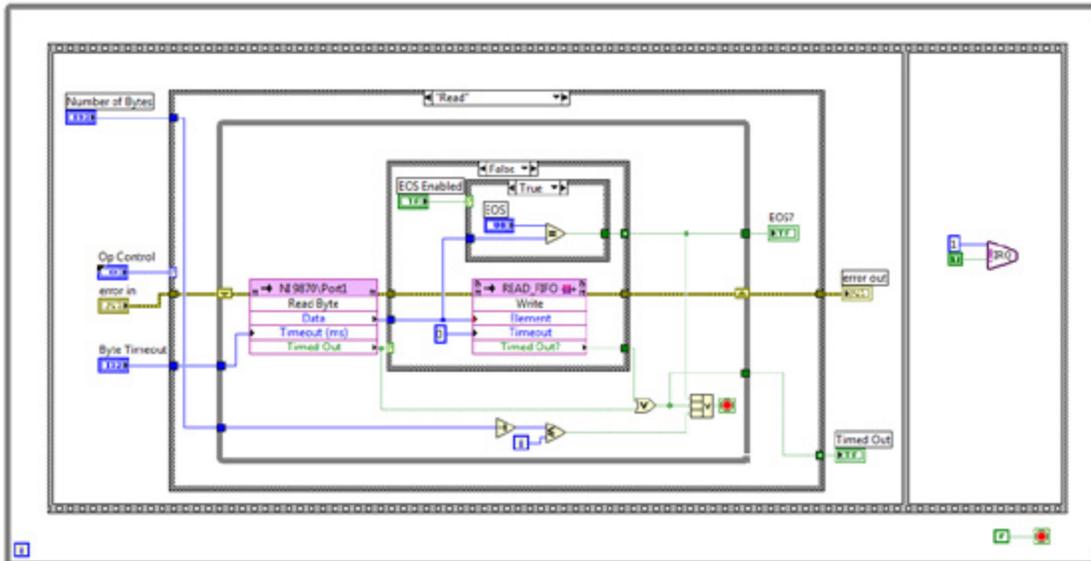


Figure 8.10. You can find the NI 987x serial loopback example for LabVIEW FPGA in the NI Example Finder.

Communicating With PLCs and Other Industrial Networked Devices

Often applications call for integrating NI programmable automation controllers (PACs) such as CompactRIO into an existing industrial system or with other industrial devices. These systems can consist of traditional PLCs, PACs, or a variety of specialty devices such as motor controllers, sensors, and HMIs. These devices feature a wide range of communication options from simple RS232 to specialized high-speed industrial networks.

You can choose from three common methods to connect the CompactRIO controller to an industrial system or PLC. You can directly wire the CompactRIO controller to the PLC using standard analog or digital I/O. This method is simple but scales for only small systems. For larger systems, you can use an industrial communications protocol to communicate between the CompactRIO controller and the PLC. For large Supervisory Control and Data Acquisition (SCADA) applications, OPC is the most commonly used protocol. OPC scales to high channels and can be used with either Windows-based PCs or real-time OSs depending on the OPC standard.

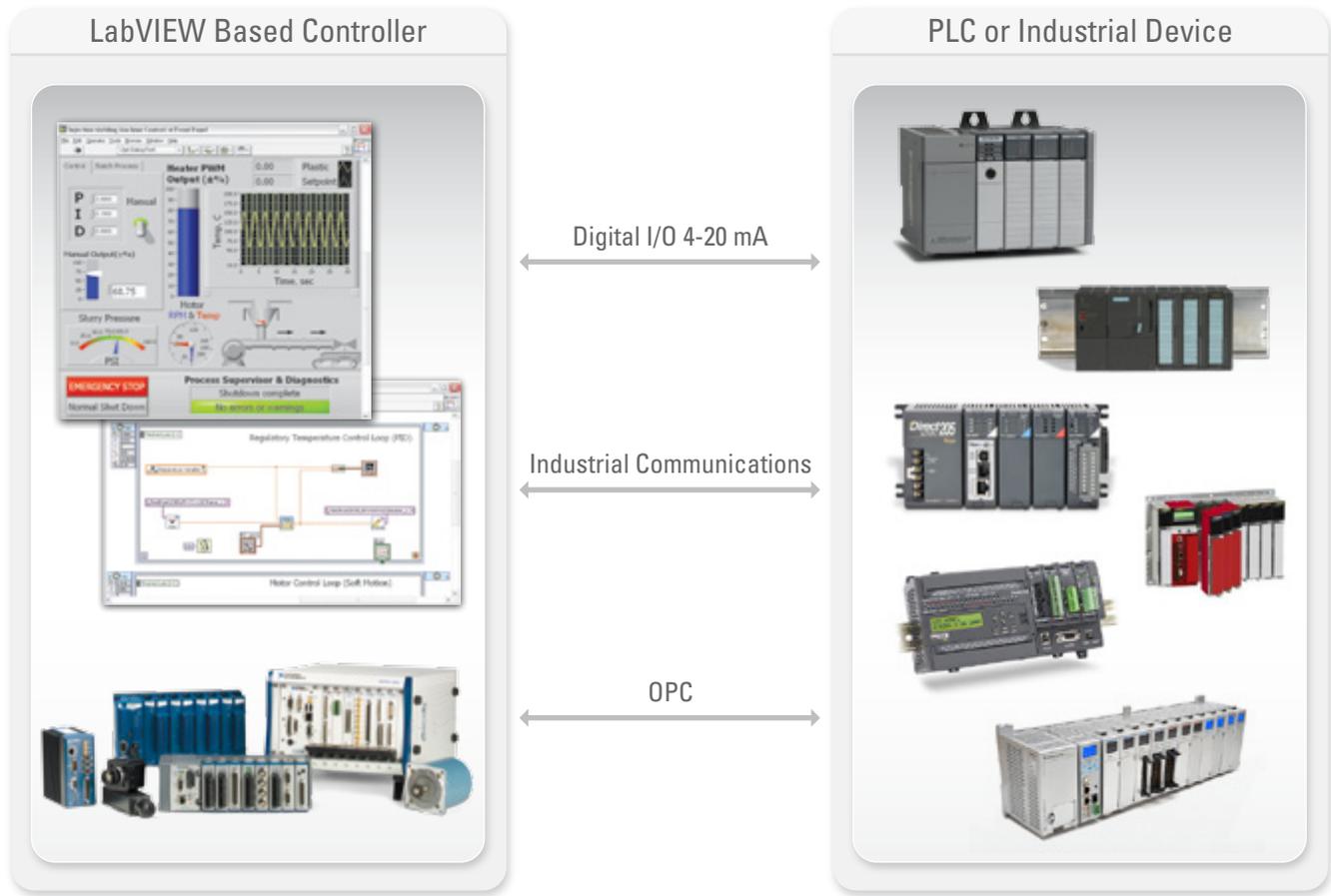


Figure 8.11. Three Methods for Connecting a CompactRIO Controller to an Industrial Device

Industrial Communications Protocols

Beyond using standard I/O, the most common way to connect a CompactRIO controller 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, so they offer 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 become tedious. Industrial communications protocols provide a framework for how data is communicated. They are fundamentally similar to the instrument drivers discussed earlier although they typically run a more sophisticated stack.

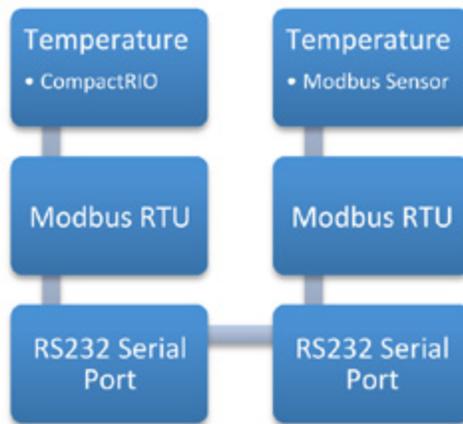


Figure 8.12. Typical Industrial Protocol Implementation

Industrial protocols take a low-level bus like RS232 or TCP/IP and add predefined techniques for communication on top. This is 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 amount of 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 to use for abstracting the low-level communication details from end users, so engineers can focus on the application.

Because CompactRIO has built-in serial and Ethernet ports, it can easily work with many protocols including Modbus TCP, Modbus Serial, and EtherNet/IP. You also can use plug-in modules for communication with most common industrial protocols such as PROFIBUS and CANopen.

When you cannot use native communication, another option is gateways. Gateways are protocol translators that can convert between different industrial protocols. For example, you can connect to a CC-Link network 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. Developed in 1979, it 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. You can implement it using asynchronous serial transmission to communicate with touch screens, PLCs, and gateways to support other types of industrial buses. Modbus works with a CompactRIO controller and its onboard serial or Ethernet modules. Note that the onboard serial port of CompactRIO hardware uses RS232 though some Modbus devices may use the RS485 electrical layer. In this case, you can use a common RS485-to-RS232 adapter.

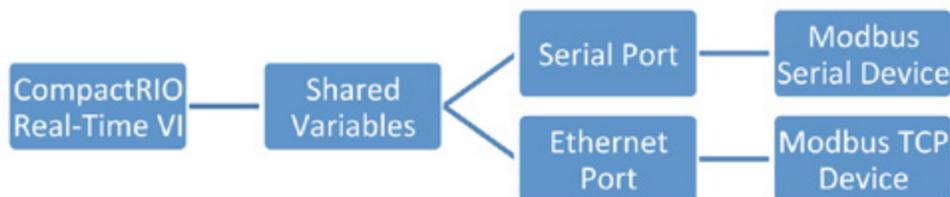


Figure 8.13. Software Architecture for Communicating With Modbus Devices

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	Only the master device can read. Only the slave device can change its register values.
Coils	Single-Bit	Read-Write	Both master and slave devices can read and write to these registers.
Input Registers	16-Bit Word	Read Only	Only the master device can read. Only the slave device can change its register values.
Holding Registers	16-Bit Word	Read-Write	Both master and slave devices can read and write to these registers.

Table 8.2. Information is passed between master and slave devices by reading and writing to registers located on the slave device.

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 controller as a master. This is the most common configuration. Likewise, connecting to a Modbus master (such as another PLC) requires configuring the CompactRIO controller 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-to-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 a raw binary form, 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 that is 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 Tutorial

The NI Developer Zone document [How to Turn an RTTarget Into a Modbus Slave Using I/O Servers](#) walks you through the steps of turning your CompactRIO controller into a Modbus slave. It also shows you how to read from this server with LabVIEW for Windows if you do not have a third-party Modbus server. Before starting the tutorial, install the Modbus I/O Server onto your CompactRIO target through MAX.

Additionally, with the LabVIEW Real-Time and LabVIEW Datalogging and Supervisory Control (DSC) modules, you can configure Modbus I/O servers on the development computer to communicate with Modbus devices. If you do not have the LabVIEW Real-Time or LabVIEW DSC modules, you can use the free [LabVIEW Modbus Library](#) to create Modbus master or slave applications with any Ethernet or serial ports on your development computer. This

library provides a set of lower-level VIs for greater flexibility, performance, and customization. Download the [LabVIEW Modbus Library](#) to begin programming your Modbus applications today.

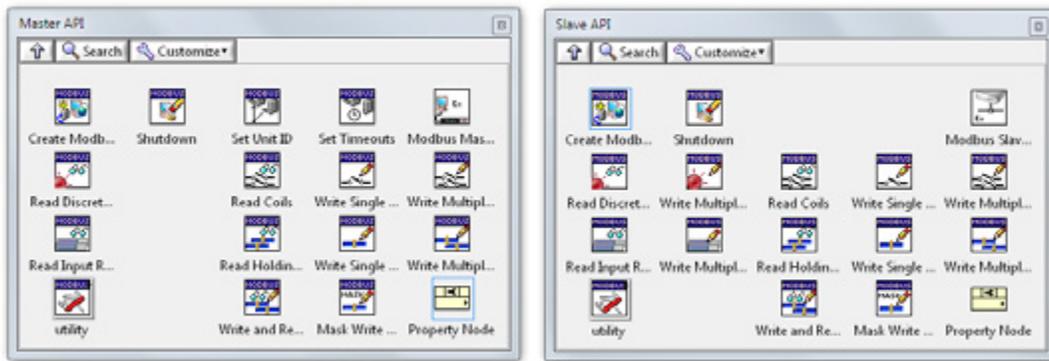


Figure 8.14. The LabVIEW Modbus Library provides low-level functions for greater flexibility and performance.

Modbus Serial- and TCP-based communications are also available as a third-party add-on on the LabVIEW Tools Network called [ModBusVIEW over TCP](#).

EtherNet/IP

EtherNet/IP is a protocol built on the Common Industrial Protocol (CIP) for communications with EtherNet/IP devices. This protocol, developed and commonly found on Rockwell Automation (Allen-Bradley) PLCs, uses standard Ethernet cabling for communications with industrial I/O. EtherNet/IP is an application layer protocol that uses TCP for general messages and User Datagram Protocol (UDP) for I/O messaging and control.

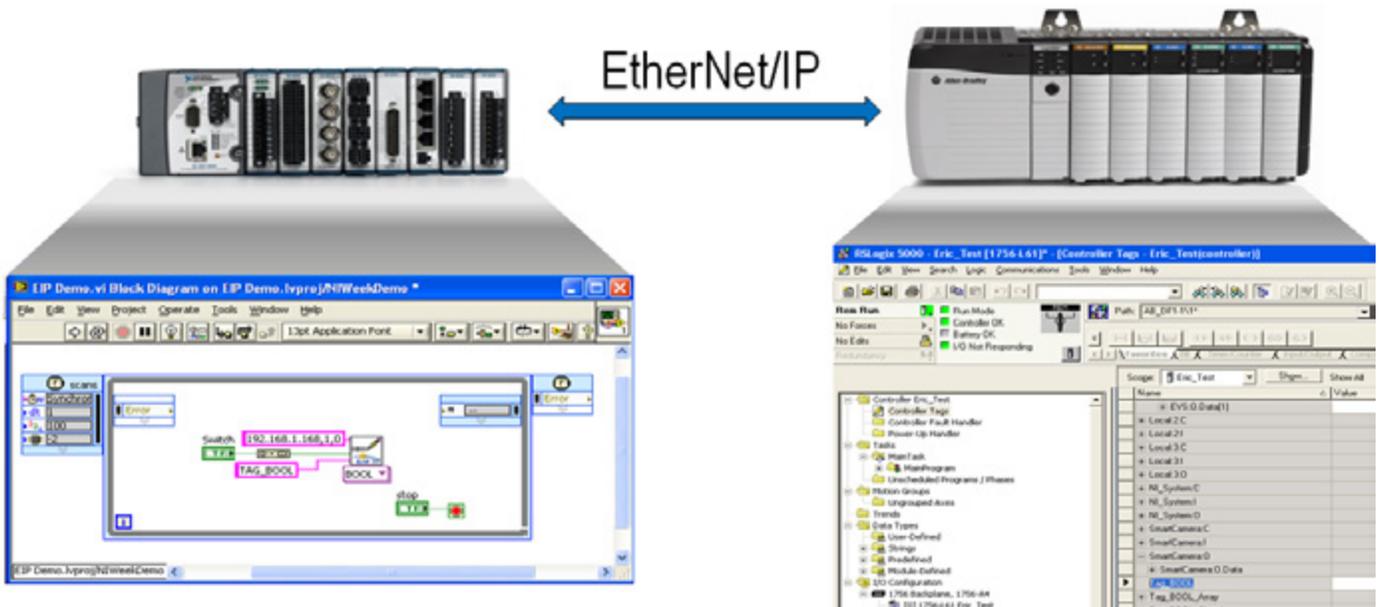


Figure 8.15. With EtherNet/IP, you can directly read and write to tags on a Rockwell PLC from CompactRIO.

The LabVIEW Driver for EtherNet/IP provides both explicit messaging API and implicit I/O data communication with a wide range of PLCs and other EtherNet/IP devices. With this driver, the CompactRIO controller supports direct communication to Rockwell Automation PLCs, allowing for easy integration into a new or an existing system.

Energy Protocol: DNP3

DNP3 (Distributed Network Protocol) is a protocol specification for vendors of power grid SCADA components. This protocol is commonly used in electrical and water utilities to communicate between a master station and other devices such as remote terminal units (RTUs) and other intelligent electronic devices (IEDs).

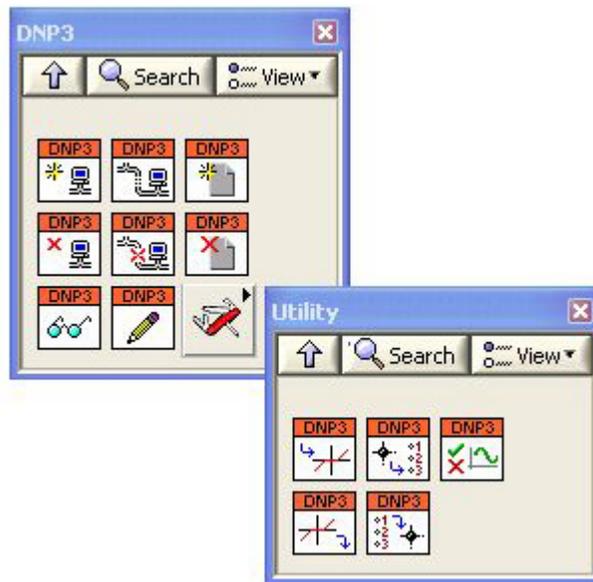


Figure 8.16. DNP3 functions provide capabilities to create outstations in LabVIEW.

You can use the LabVIEW Driver for DNP3 Outstation Support to program a real-time target, such as a CompactRIO system, as an outstation device with advanced functionality like power quality monitoring, phasor measurements, and other smart grid-related analysis. The DNP3 software driver supports Ethernet and serial communication, file transfer, and time synchronization between the master and outstation. Multiple communication channels per outstation and multiple sessions (logical devices) per channel also work with this driver.

OPC

OPC, or Open Platform Communications, 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 section covers OPC Data Access (DA), the core specification that defines how to universally move data around using common Windows technologies and OPC Unified Architecture (UA), a cross-platform standard optimized for security and with extended functionality.

OPC Data Access (DA)

OPC DA is a standard interface defined by the OPC Foundation that allows communication between numerous data sources, including devices on a factory floor, laboratory equipment, test system fixtures, and databases. The LabVIEW Real-Time and LabVIEW DSC modules provide OPC Client I/O servers that can communicate with any server implementing the OPC Foundation OPC server interface through the OPC DA standard. When LabVIEW acts as an OPC server, it uses the Shared Variable Engine to create OPC tags out of network-published shared variables that an OPC DA client can connect to. This allows LabVIEW VIs to easily communicate with other OPC DA client software. The network-published shared variable in LabVIEW provides a gateway to OPC.

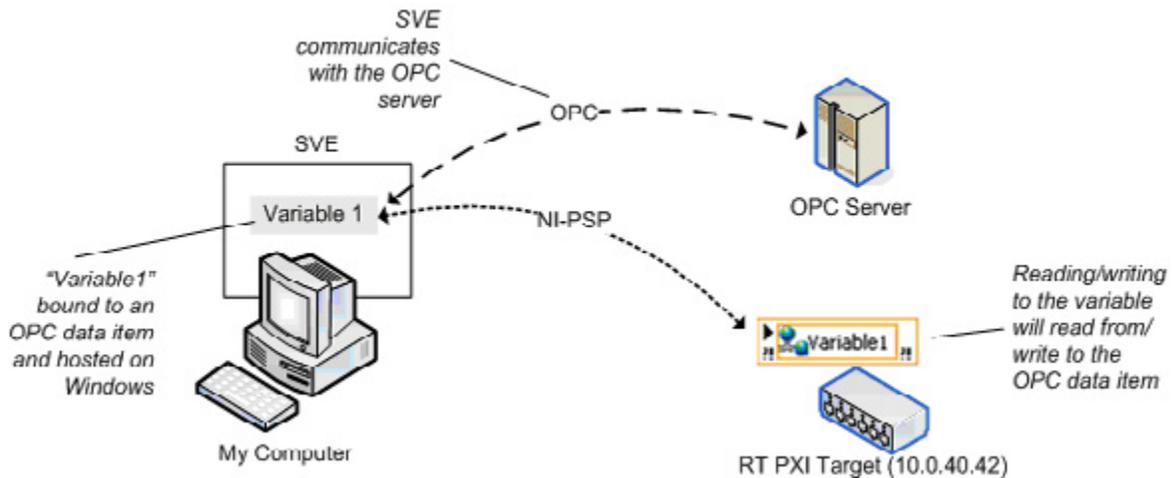


Figure 8.17. Communicate OPC data items with network-published shared variables.

OPC DA is a Windows-only technology and the CompactRIO controller cannot directly communicate using the OPC DA protocol. Instead, a Windows PC can communicate to the CompactRIO controller using the NI-PSP protocol and can translate and republish the information via OPC. When running on a Windows machine, the Shared Variable Engine functions as an OPC server and performs this translation and republishing automatically. This means other OPC DA 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 DA

The following example describes how to publish network shared variables as OPC DA items using the Shared Variable Engine as an OPC DA server. The example assumes you are already publishing data on the network via network-published shared variables hosted by the CompactRIO system. If the network-published shared variables are hosted on a Windows PC, they are published as OPC DA items by default. You can verify that the variable is working correctly by launching an OPC DA client such as the NI Distributed System Manager as described in step 6.

1. The CompactRIO controller publishes data onto the network using network-published shared variables hosted on the CompactRIO controller. In this example, the network shared variable is named SV_PID_SetPoint.

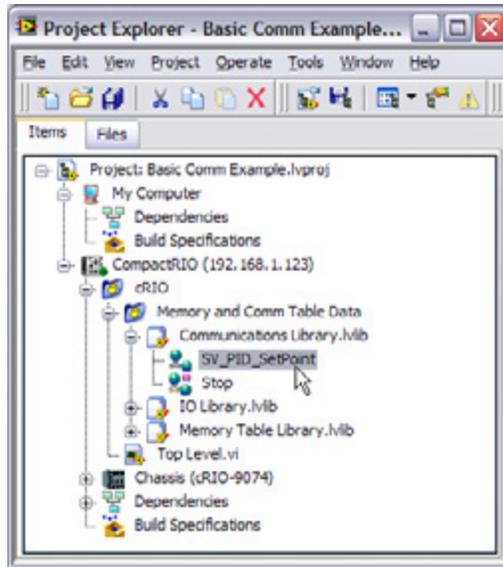


Figure 8.18. Network-Published Shared Variables Hosted on the CompactRIO System

2. To publish the variables as OPC DA data items, bind each variable hosted on the CompactRIO controller to a variable hosted on the Windows PC. To create variables hosted on a Windows PC, right-click My Computer in the LabVIEW Project and select **New»Variable**.

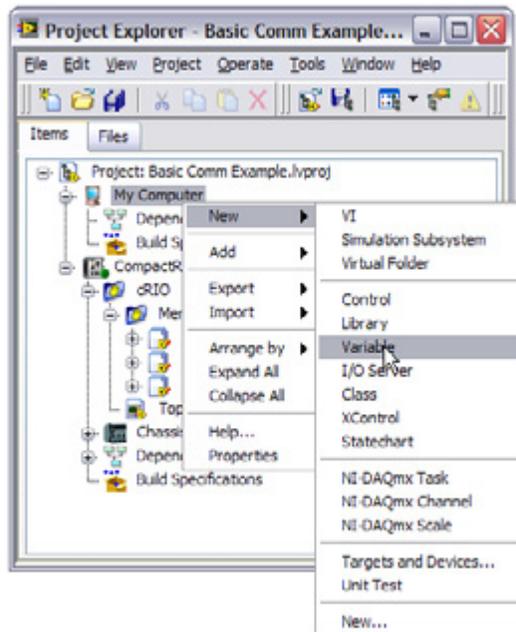


Figure 8.19. Network-published shared variables hosted on Windows are automatically published via OPC.

3. 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 controller.

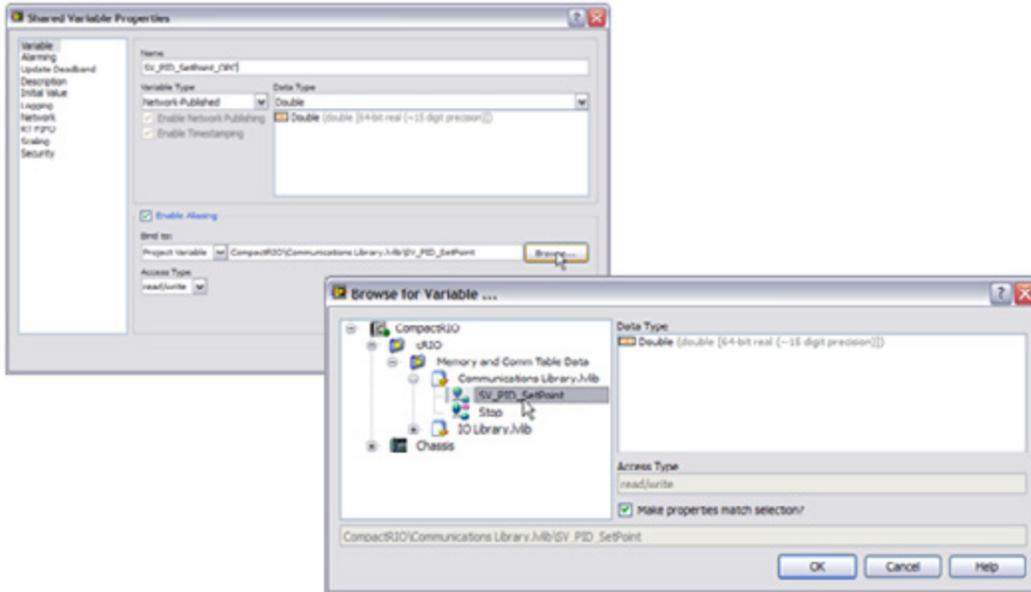


Figure 8.20. An aliased network-published shared variable provides a gateway between CompactRIO and OPC.

4. Save the new library you create with a useful name, such as OPC Library.lvlib.
5. Deploy the variables to the Shared Variable Engine by right-clicking the library and selecting Deploy.
6. The shared variable is now published as an OPC DA item. Other OPC DA clients can now connect to the Windows PC and use the data for display, HMI, and so on. You can verify that the variable is working correctly by launching an OPC DA client such as the NI Distributed System Manager (Start»Programs»National Instruments»NI Distributed System Manager).

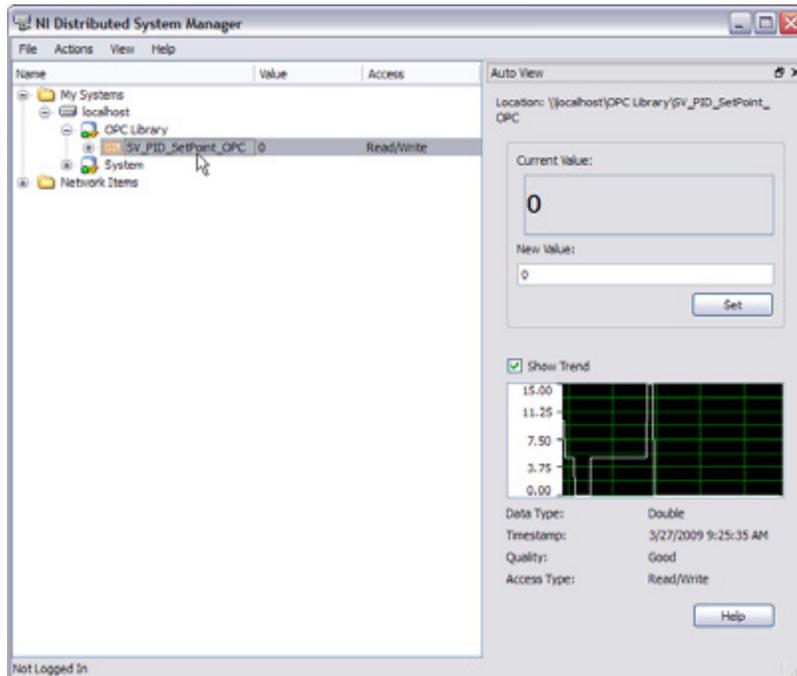


Figure 8.21. The OPC clients, such as the NI Distributed System Manager, can read and write to the OPC item.

7. If you have an OPC DA client, such as the LabVIEW DSC Module, you can verify that the variable is working from that client as well. NI shared variable OPC DA items are published under the National Instruments. Variable Engine.1 server name.
8. Once variables are deployed in the Shared Variable Engine, they stay deployed with subsequent reboots of the OPC DA server machine.
9. Find more information on the Shared Variable Engine and OPC DA in the LabVIEW Help file [Connecting to OPC Systems Using LabVIEW \(Windows Only\)](#).

OPC Unified Architecture (UA)

OPC UA is a new communication technology standard that was first released by the OPC Foundation in 2006 as an improvement on its predecessor, Classic OPC. OPC UA includes the functionality of Classic OPC to offer current data access, alarms, and events. Furthermore, OPC UA is based on a cross-platform, business-optimized service-oriented architecture (SOA), which expands on the security and functionality found in Classic OPC, instead of Microsoft-based COM/DCOM technology.

The [LabVIEW Datalogging and Supervisory Control \(DSC\) Module](#) and the [LabVIEW Real-Time Module](#) include the [OPC UA](#) VIs to exchange data between OPC UA servers and clients and create certificates that protect data. You need LabVIEW DSC to use the OPC UA VIs on Windows targets, and you need LabVIEW Real-Time to use the OPC UA VIs on LabVIEW Real-Time targets.

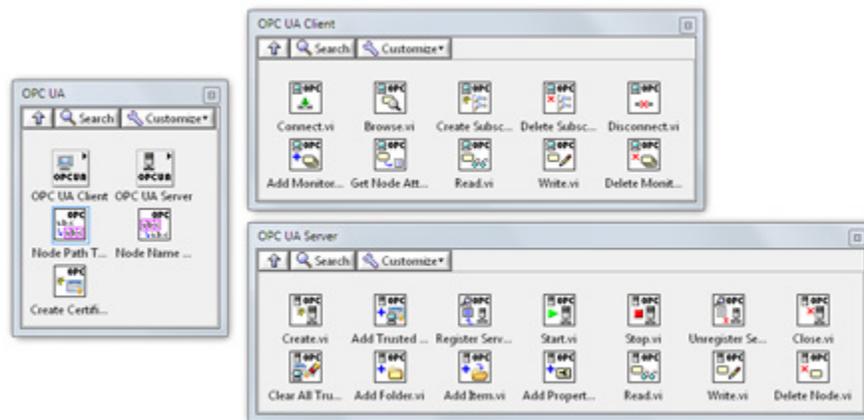


Figure 8.22. The OPC UA VIs support both nonsecure connections and secure connections between an OPC UA server and an OPC UA client.

To learn more about the OPC UA standard, read the white paper [Why OPC UA Matters](#). For example code on how to use the OPC UA API, review the project `C:\Program Files\National Instruments\LabVIEW XXXX\examples\comm\OPCUA\OPC UA Demo.lvproj`.

OPC Servers

In addition to OPC DA I/O servers and the OPC UA API, National Instruments provides an OPC Server solution that contains a list of drivers for many of the industry's PLCs. For a list of supported PLCs, refer to the NI Developer Zone article [Supported Device & Driver Plug-in List for NI-OPC Servers](#).

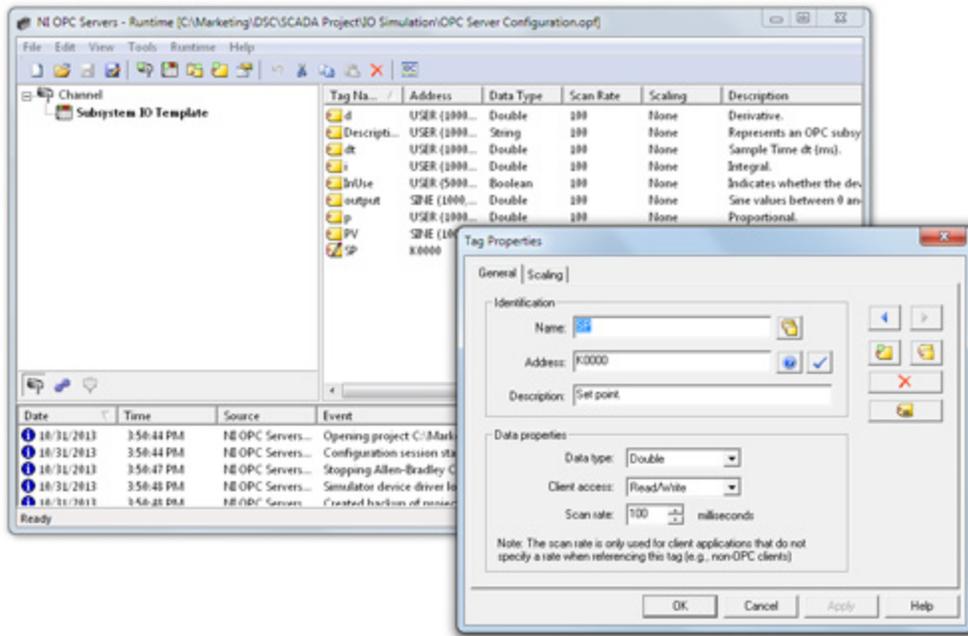


Figure 8.23. OPC servers allow connectivity to hundreds of third-party PLCs.

To learn more about OPC connectivity, visit ni.com/opc.

Additional Resources

For information on the other industrial communications protocols NI supports, visit ni.com/industrialcommunications.

CHAPTER 9

Designing a Touch Panel HMI

Building User Interfaces and HMIs Using LabVIEW

This chapter examines one software design architecture for building an operator interface with a scalable navigation engine for cycling through different human machine interface (HMI) pages.

You can use this architecture to build an HMI based on LabVIEW for any HMI hardware targets including the NI TPC-2212 touch panel computer running the Windows Embedded Standard 7 (WES7) OS or the NI TPC-2206 running the Windows XP Embedded (XPe) 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 an NI touch panel computer (TPC) or any HMIs running Windows Embedded Standard 7 or Windows XP Embedded.

Basic HMI Architecture Background

An HMI can be as simple or as complex as the functionality you require. The 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 loop
3. Navigation (user interface) loop

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. Initialization and shutdown tasks are not diagrammed since they are not processes (they execute only once).

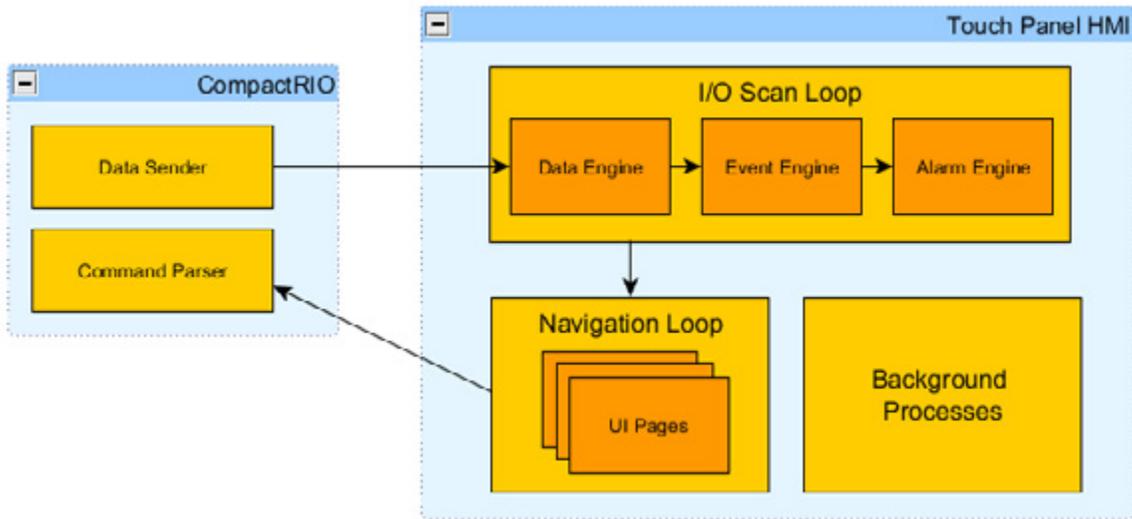


Figure 9.1. Basic Software Architecture of a Touch Panel HMI Communicating With a CompactRIO Controller

I/O Scan Loop

The I/O scan loop typically consists of three components: the Data Engine, the Event Engine, and the Alarm Displays Engine.

Data Engine

The Data Engine exchanges tag values with the controller via an Ethernet-based communication protocol or Modbus. It receives this data across the network and makes it available throughout the HMI application.

Event Engine

The I/O scan loop might also handle alarms and events. The Event Engine compares a subset of tag values to a set of predefined conditions (value equal to X, value in range or out of range, and so on) and logs an event when a tag value matches one of its event conditions.

Alarm Engine

Some events are simply logged while other events require operator intervention and are configured as alarms. The alarm event data is sent to the Alarm Displays Engine, which manages how the alarm is presented to the operator. When the tag value leaves the alarm state, the Event Engine sends an alarm canceling the event to the Alarm Displays Engine.

Navigation Loop

The navigation loop handles the management and organization of different UIs in your application. You can implement a navigation loop as a simple state machine built with a While Loop and a Case structure. Each case encloses an HMI page VI that, when called, is displayed on the HMI screen. Figure 9.2 is an example of a navigation loop.

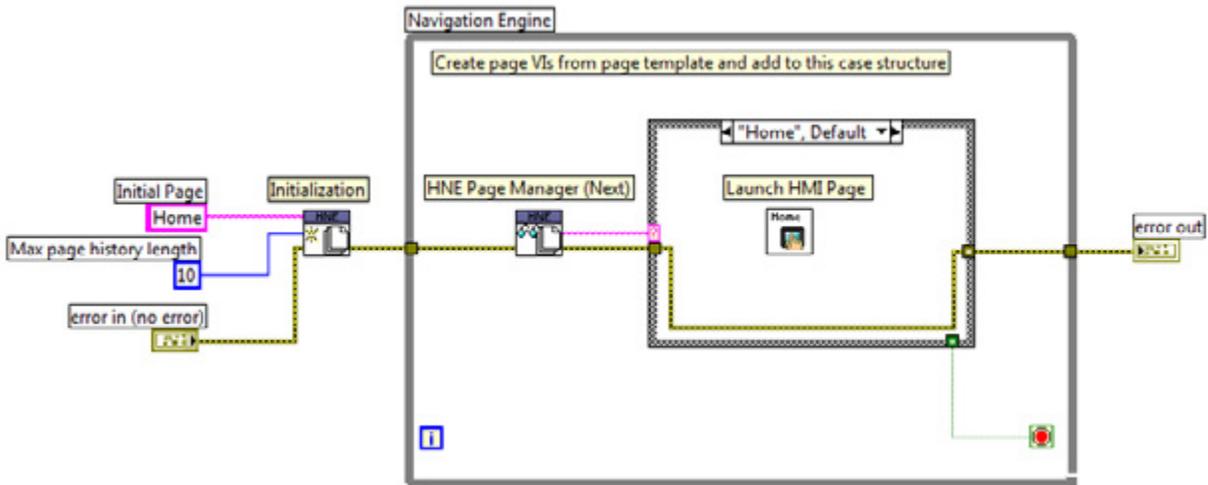


Figure 9.2. Example Navigation Loop Block Diagram

This example uses the [HMI Navigation Engine \(HNE\) Reference Library](#), which was created for HMI page management and navigation. The HNE is based on VIs included with the LabVIEW Touch Panel Module, but it features an additional history cache so the user can define a button to jump backward toward the previously viewed screen. The HNE also includes basic templates and examples for getting started. Refer to the NI Developer Zone document [HMI Navigation Engine \(HNE\) Reference Library](#) to download the HNE library.

The HNE installs a page manager API palette named HNE to the User Libraries palette in LabVIEW.

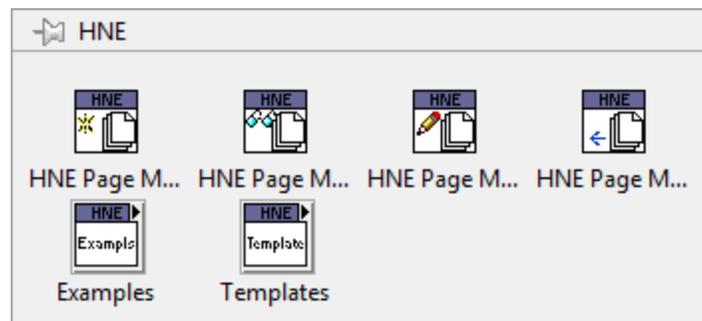


Figure 9.3. HNE (HMI Navigation Engine) Palette

- **HNE Page Manager (Init)**—This VI initializes the HNE by setting the navigation history depth and setting the name of the first page to be displayed.
- **HNE Page Manager (Next)**—This VI returns the name of the next page and passes it to the Case structure in the HNE.
- **HNE Page Manager (Set)**—This VI sets the name of the next page to be displayed. It is used within HMI pages to support navigation buttons.
- **HNE Page Manager (Back)**—This VI returns the name of the previous page from the page history. It is used within HMI pages to support the operation of a “back” navigation button.
- **Examples subpalette**—This is a subpalette that contains the example VI for the HNE API.
- **Templates subpalette**—This is a subpalette that contains two template VIs for the HNE API. The first is a template VI for the navigation loop called HMI_NavigationEngine VI and the second is a template for an HMI page called HMI_Page VI.

The navigation engine uses VIs from the Touch Panel Navigation palette.

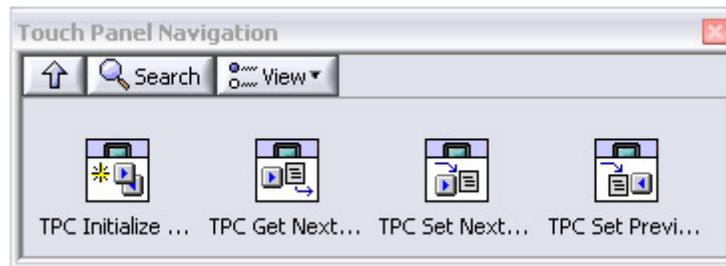


Figure 9.4. The Navigation Palette

- **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.

UI Pages

As stated above, the navigation loop contains all of the HMI pages for an application. Each HMI page is a LabVIEW VI created to monitor and configure a specific process or subprocess in the machine. The most common elements on a page front panel are navigation buttons, action buttons, numeric indicators, graphs, images, and Boolean controls and indicators. Figure 9.5 shows an example page containing a typical set of front panel elements.

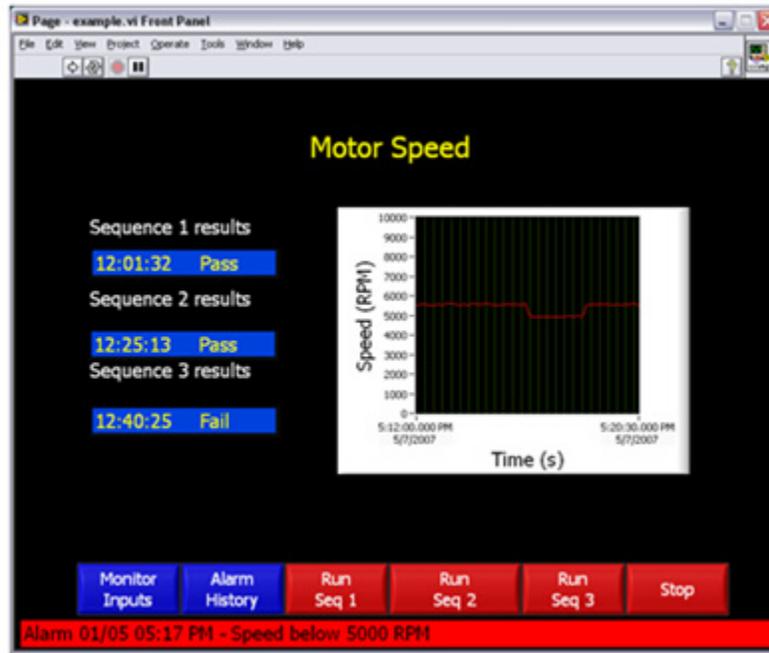


Figure 9.5. Example of a Typical UI Page in LabVIEW

The page block diagram uses the event-based producer consumer design pattern to implement a responsive, event-driven UI. The example in Figure 9.6 uses the Asynchronous Message Communication (AMC) reference library for interprocess communication. You can download the AMC reference library, which uses queues for interprocess communication, from the NI Developer Zone document [Asynchronous Message Communication \(AMC\) Reference Library](#). The AMC library also has an API based on UDP that you can use to send messages across the network. You can implement the Figure 9.6 example using queue functions for communicating between the Event Handler process and the Message process.

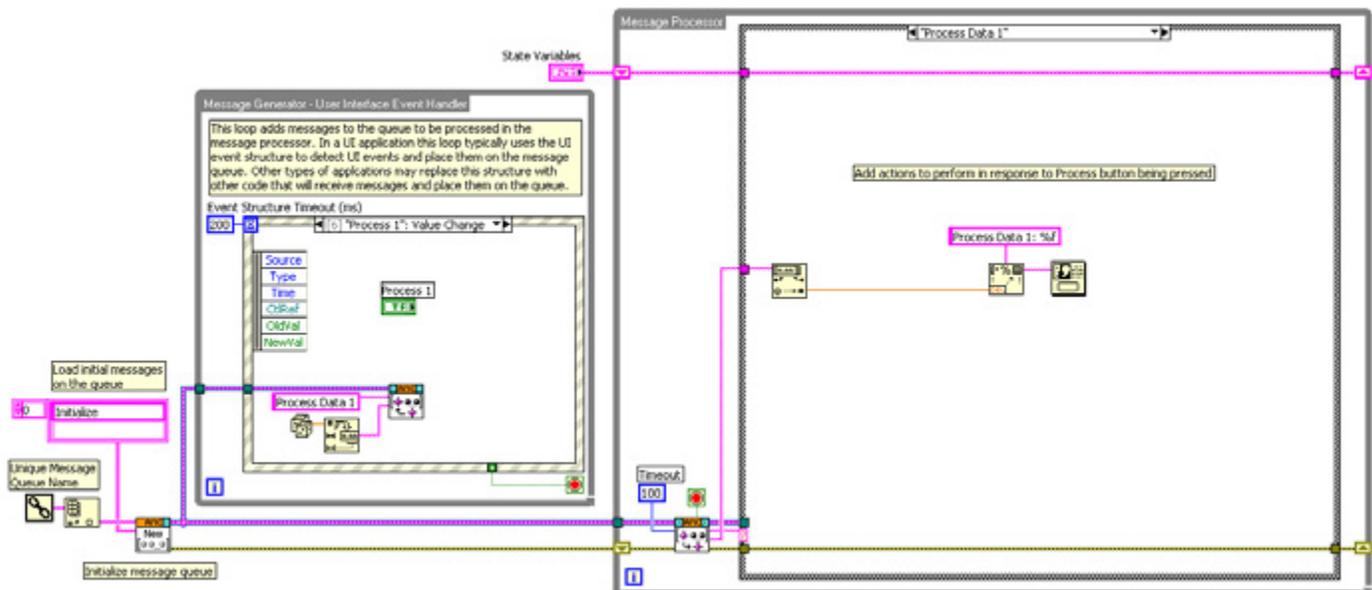


Figure 9.6. Example Page Block Diagram Using AMC Design Pattern

For more information on creating UI pages, see the NI Developer Zone document [Creating HMI Pages for the LabVIEW Touch Panel Module](#).

CHAPTER 10

Adding Vision and Motion

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.

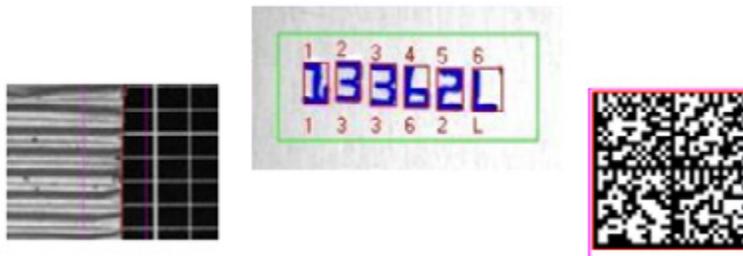


Figure 10.1. Edge detection, optical character recognition, and 2D code reading are common machine vision tasks.

Often 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 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, USB, or Camera Link. The real-time system processes the images and has I/O 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.

NI offers both types of embedded machine vision systems. The NI Compact Vision System (Figure 10.2) is a real-time embedded vision system that features direct connectivity to GigE Vision cameras as well as FPGA-based I/O channels for synchronization and triggering. This system features Ethernet connectivity to your industrial network, allowing communication back to CompactRIO hardware. Vision integration is also directly available on certain CompactRIO targets through GigE Vision cameras and USB3 Vision cameras connected through USB 2.0 ports.

NI Smart Cameras (Figure 10.2) are industrial image sensors combined with programmable processors to create rugged, all-in-one solutions for machine vision applications. They have multiple resolutions from VGA (640x480 pixels)

to 5MP with color and monochrome options. These cameras feature dual Gigabit Ethernet ports, digital inputs and outputs, and a built-in lighting controller.



Figure 10.2. NI Compact Vision System and NI Smart Camera

NI also offers plug-in image acquisition devices called frame grabbers that provide connectivity between industrial cameras and PCI, PCI Express, PXI, and PXI Express slots. These devices 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 it to the industrial platform that best suits your application requirements with minimal code changes.

Lighting and Optics

This guide does not cover lighting and optics in depth, but they are crucial to the success of your application. The following resources examine 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. You must ensure that the field of view (FOV) of your system includes the object you want to inspect. To calculate the horizontal and vertical FOV of your imaging system, use Equation 10.1 and the specifications for the image sensor of your camera.

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

Equation 10.1. 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

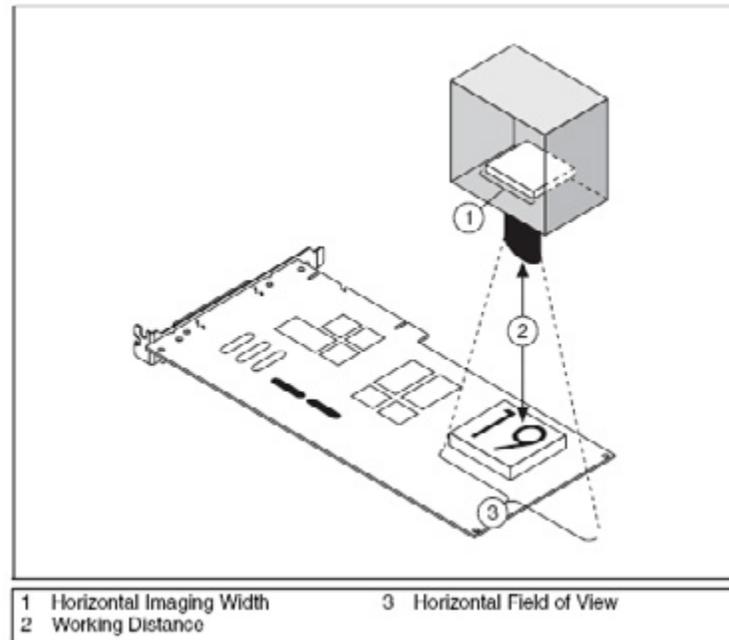


Figure 10.3. The lens selection determines the FOV.

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 FOV 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 FOV 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 may need to adjust the various parameters in the FOV equation until you achieve the right combination of components that match your inspection needs. This may 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. NI 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 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.

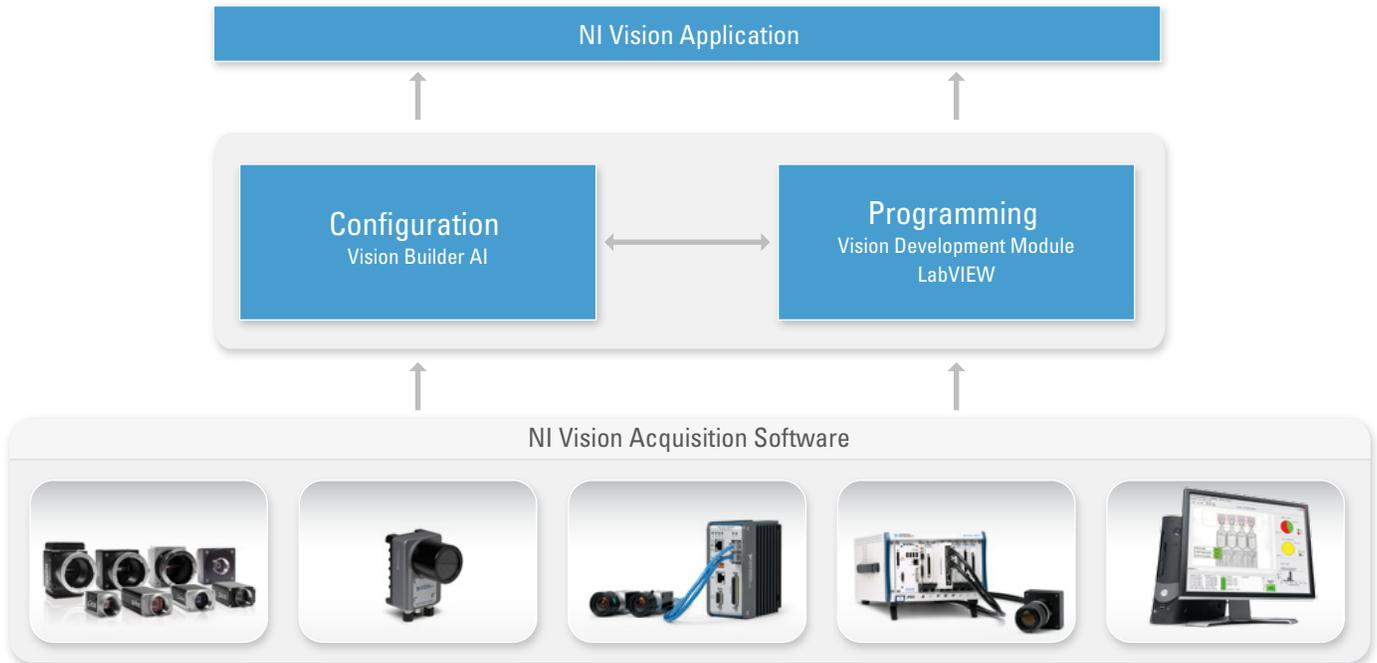


Figure 10.4. NI offers both configuration software and a full programming environment for machine vision application development.

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.

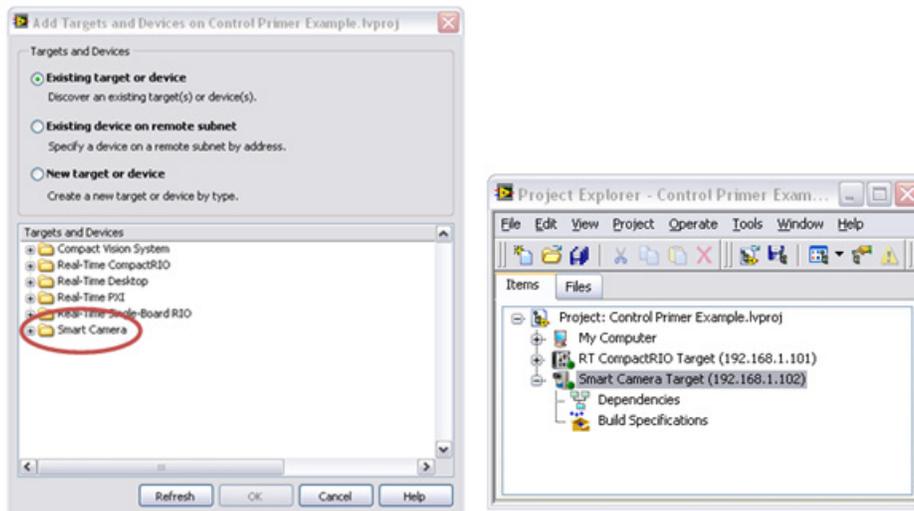


Figure 10.5. You can add NI Smart Camera systems to the same LabVIEW project as CompactRIO systems.

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 using the NI Vision Acquisition Software driver to acquire your images and algorithms in the Vision Development Module to process them.

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

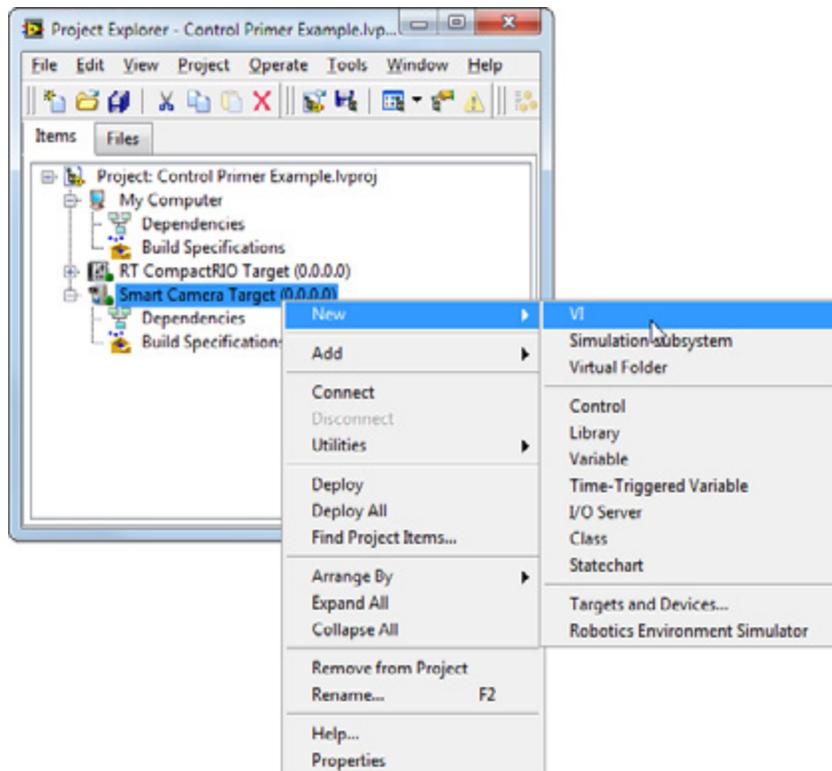


Figure 10.6. Adding a VI on Your NI Smart Camera to Your Project in LabVIEW Real-Time

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

To simplify the process of acquiring and processing images, NI includes Express VIs in the Vision palette. Use these Express VIs in this example to acquire images from the smart camera (or vision system) as well as process the images. To access these Express VIs, right-click on the block diagram and choose **Vision»Vision Express**.



Figure 10.7. The Vision Express Palette

The first step is to set up an acquisition from your camera. 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.

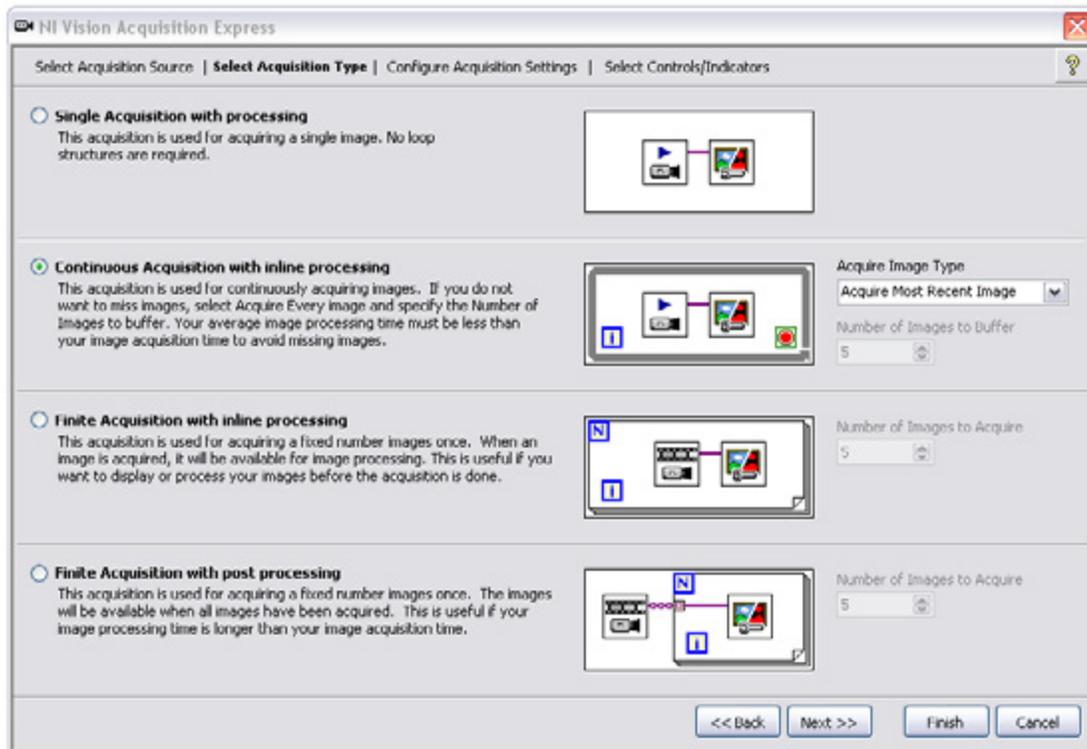


Figure 10.8. The Vision Acquisition Express VI guides you through creating a vision application in LabVIEW.

Next, choose which type of acquisition you are implementing. For this example, select Continuous Acquisition with inline processing so you can 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 Acquisition 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 real-time systems. For an overview of the tools 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 system. You can use network-published shared variables to pass data between the two systems. Network communication between LabVIEW systems is 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.

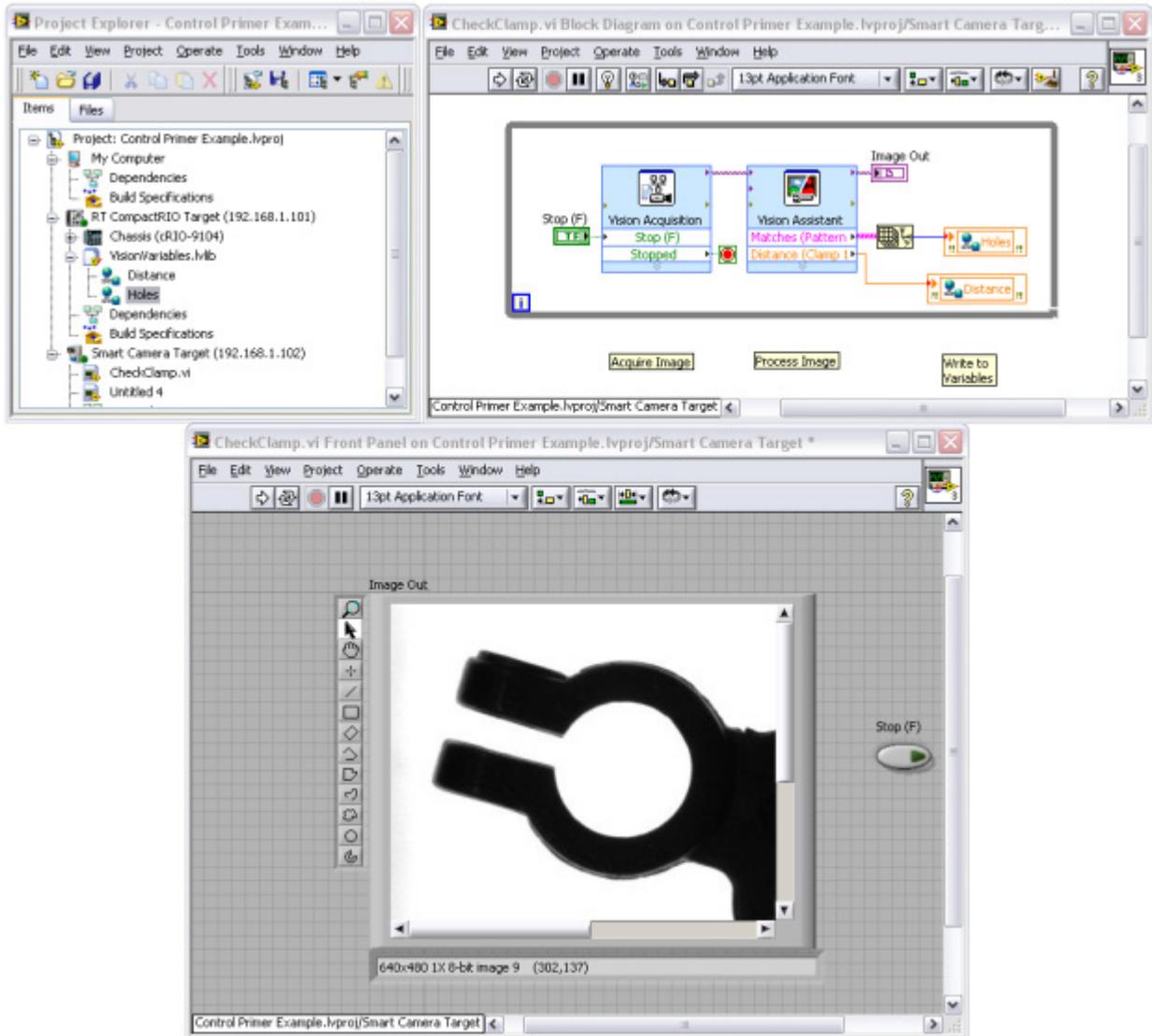


Figure 10.9. A Complete Inspection in LabVIEW

As you can see in Figure 10.9, the results of the inspection are passed as current values to the CompactRIO system via shared variables hosted on the CompactRIO system. You can also pass the data as commands to the CompactRIO system.

Machine Vision Using Vision Builder AI

As mentioned previously, 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.

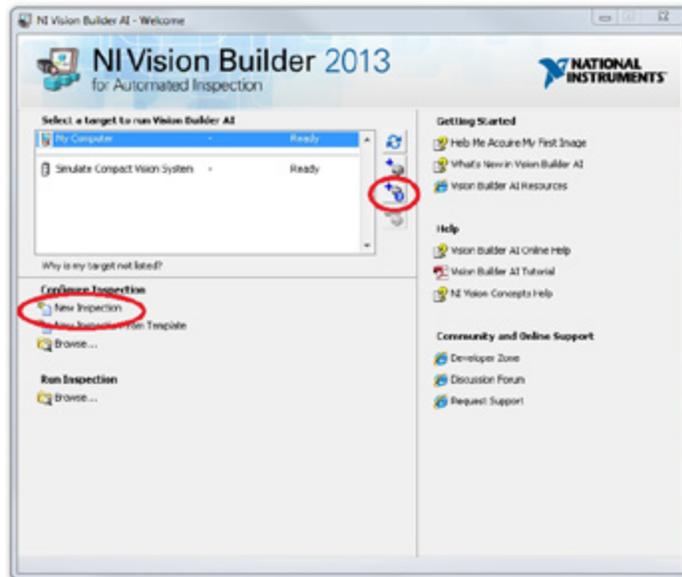


Figure 10.10. Choose your execution target from the Vision Builder AI splash screen.

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 New Inspection under 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.

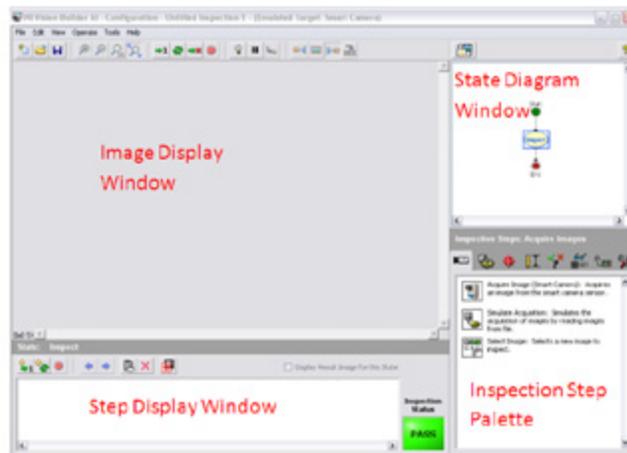


Figure 10.11. The Vision Builder AI Development Environment

This example implements the same inspection as the LabVIEW example that 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 2013\Demolmg\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.

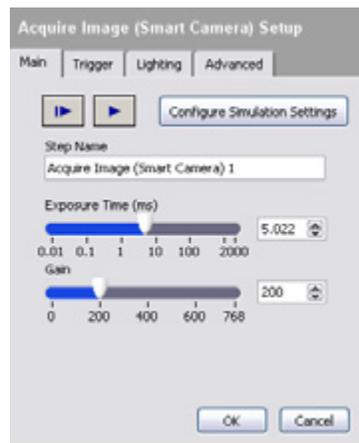


Figure 10.12. The Image Acquisition Setup Step

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.

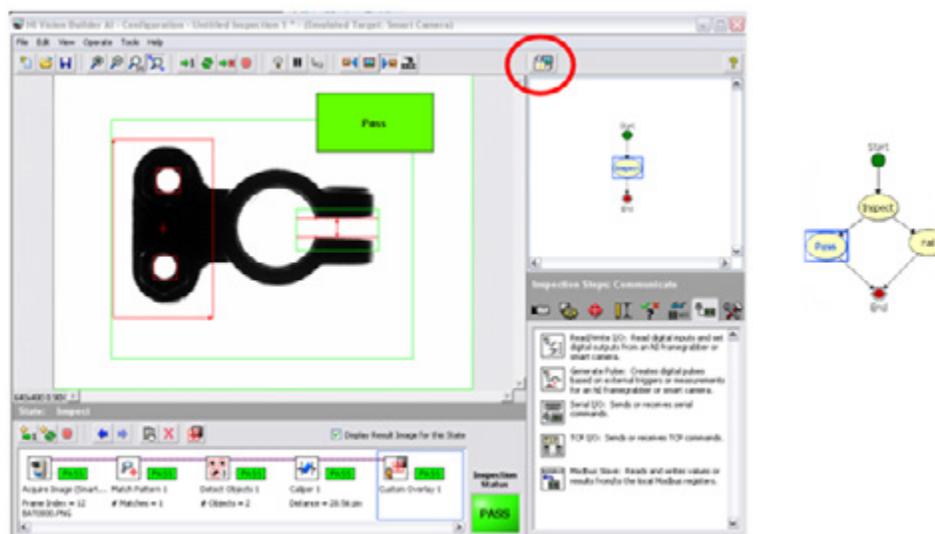


Figure 10.13. A Completed Inspection in Vision Builder AI

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 discussed previously, including network-published shared variables, 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 the CompactRIO system by navigating to the Network Variables tab. From there, select and add the variables and bind them to variables within the inspection.

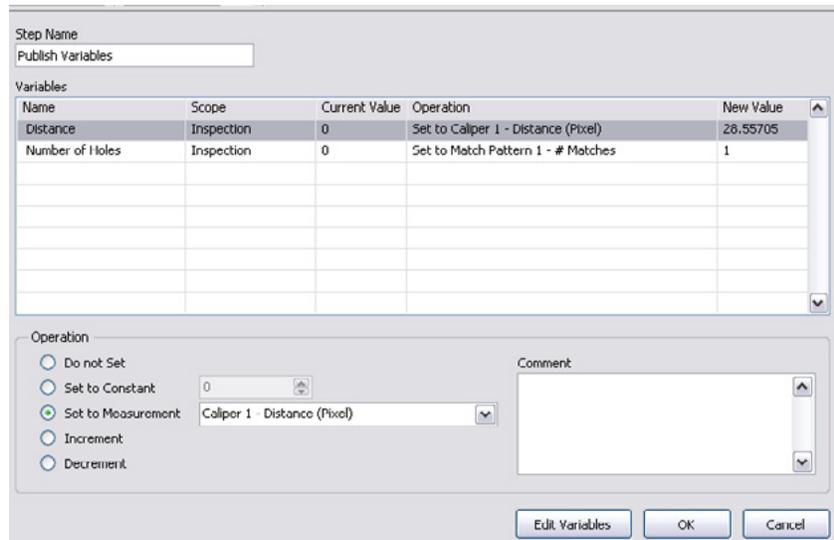


Figure 10.14. Setting Up Variable Communication in Vision Builder AI

Now these results are sent back to the CompactRIO system, 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.

Motion Control

The term “motion control” is applied to applications ranging from simple on-off or open-loop control of rotary equipment like fans and pumps to high-precision multiaxis position or velocity control synchronized to measurement and control I/O. You can implement simple motion control through industrial control I/O: a standard digital output to a motor starter or a PWM signal to a motor drive. Complex motion control applications, on the other hand, require specialized software and hardware: supervisory control, cascaded control loops, motor commutation schemes, advanced tuning algorithms, specialized sensors, actuators, feedback devices, and high-speed/high-precision I/O.

You can use CompactRIO for both simple and complex motion applications. For simple applications like standard pump or fan control, generic analog or digital I/O modules and user-defined logic are sufficient. For more complex applications that require precise position or velocity control, multiaxis coordination, or synchronization with I/O, NI offers the LabVIEW NI SoftMotion Module.

NI SoftMotion turns a CompactRIO controller into a configurable custom motion controller by providing modular motion control components that run on the CompactRIO real-time controller and user-configurable FPGA. Components include APIs for motion programming in LabVIEW or in C, a motion engine that handles supervisory control and trajectory generation, and motion control IP blocks for control loops and motion I/O. The modular components of NI SoftMotion are combined differently depending on the system architecture and motion I/O selection. Because NI SoftMotion is implemented in the LabVIEW Real-Time and LabVIEW FPGA modules with user-accessible components, it provides direct integration with all the other capabilities of CompactRIO at both the LabVIEW Real-Time and LabVIEW FPGA levels.

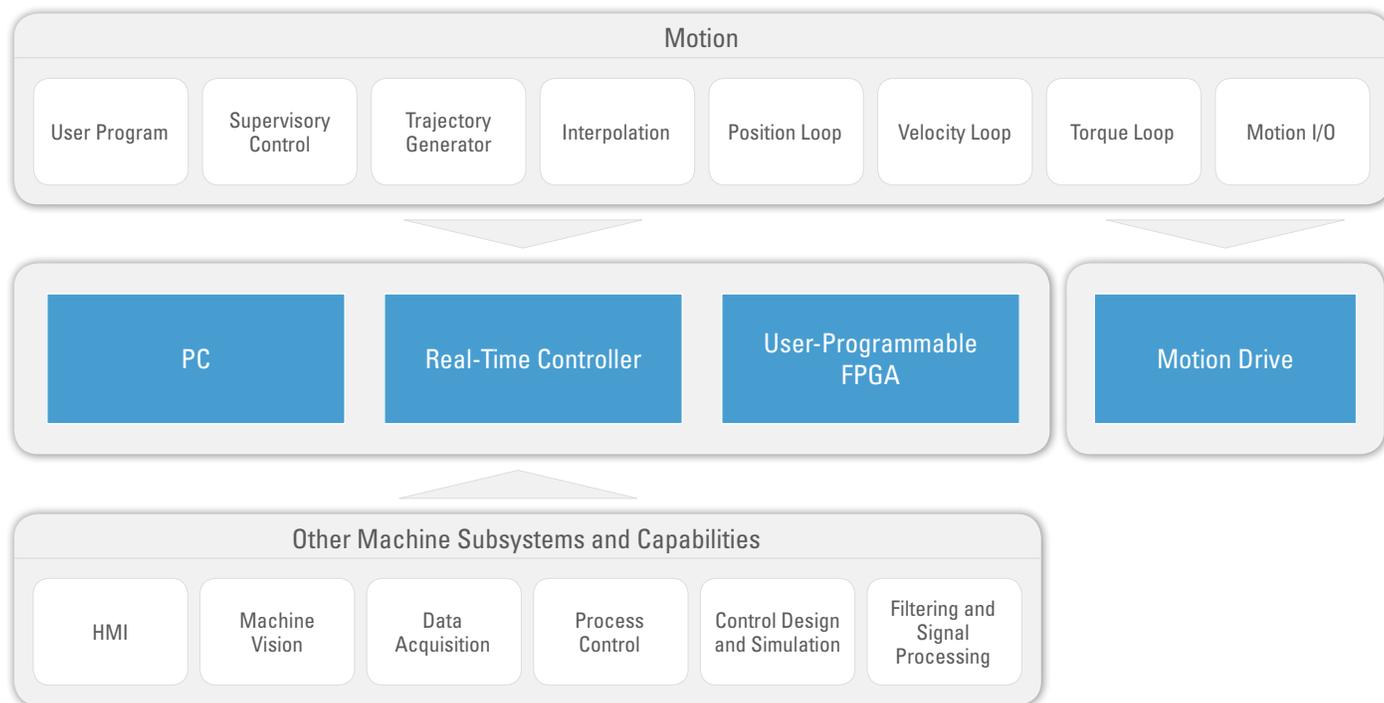


Figure 10.15. The LabVIEW NI SoftMotion Module turns a CompactRIO controller into a configurable custom motion controller.

You can choose from three versions of NI SoftMotion: Lite, Standard, and Premium.

- **LabVIEW NI SoftMotion Module Lite**—Provides a subset of functionality offered by the LabVIEW NI SoftMotion Module for free. With this version, you can perform simple single-axis point-to-point motion using NI C Series drive interfaces, create motion applications for NI CompactRIO hardware, and connect to AKD drives through EtherCAT.
- **LabVIEW NI SoftMotion Module Standard**—Offers full support for coordinated motion. You can use it to create coordinate spaces, easily achieve multi-axis synchronization, and perform electronic gearing and camming. The NI SoftMotion for SolidWorks interface is also included. With this tool, you can apply your custom motion application to a 3D CAD model and visualize, simulate, and validate your application before deploying to hardware.
- **LabVIEW NI SoftMotion Module Premium**—Includes everything that comes with LabVIEW NI SoftMotion Module Standard and adds support for creating custom motion applications that target hardware options other than the C Series drive interfaces and the AKD servo drive. Create custom axes using standard I/O devices such as C Series I/O modules for CompactRIO or third-party EtherCAT drives.

LabVIEW NI SoftMotion Features	Lite	Standard	Premium
Point-to-point motion	X	X	X
Motion I/O	X	X	X
Coordinated motion		X	X
Gearing/camming		X	X
Customization			X

Table 10.1. LabVIEW NI SoftMotion Module Features by Version

Visit the [LabVIEW NI SoftMotion Module page](#) on ni.com for more information and for the option to download an evaluation copy. This evaluation copy contains all the features of LabVIEW NI SoftMotion Premium for the evaluation period and then defaults to LabVIEW NI SoftMotion Lite until you provide an activation code for Standard or Premium.

Components of a Motion System

A typical motion control system used for position or velocity control consists of the following components:

- **Motion controller**—The processing element that runs software algorithms and closed control loops to generate the motion profile commands based on the move constraints from the user-defined application software and I/O feedback.
- **Communication interface**—Converts the command signals from a motion controller to actual digital or analog values that the drive can interpret.
- **Drive**—Receives the digital or analog data from a motion controller after conversion in the communication interface and converts it to real current/voltage that it applies to the motor to perform the commanded motion. In many cases, it also has a processing element that closes high-speed control loops. Some advanced drives can offload the position and velocity control loops from the motion controller and run them on board the drive.
- **Motor**—Converts the electrical energy from the drive/amplifier into mechanical energy. The motor torque constant, kt , and the motor efficiency define the ratio between motor current and mechanical torque output.
- **Mechanical transmission**—Consists of the components connected to the motor that direct the rotary motion of the motor to do work. This typically involves mechanical devices such as gearboxes or pulleys and lead screws that convert the rotary motion at the motor shaft to a linear movement at the payload with a certain transmission gear ratio. Common examples are belts, conveyors, and stages.
- **Feedback sensors**—Help close the control loops to provide instantaneous position and velocity information to the drive/amplifier and the motion controller.
- **Motion I/O**—Relays information such as limit switches, drive status, and other synchronization information back to the motion controller through the communication interface.

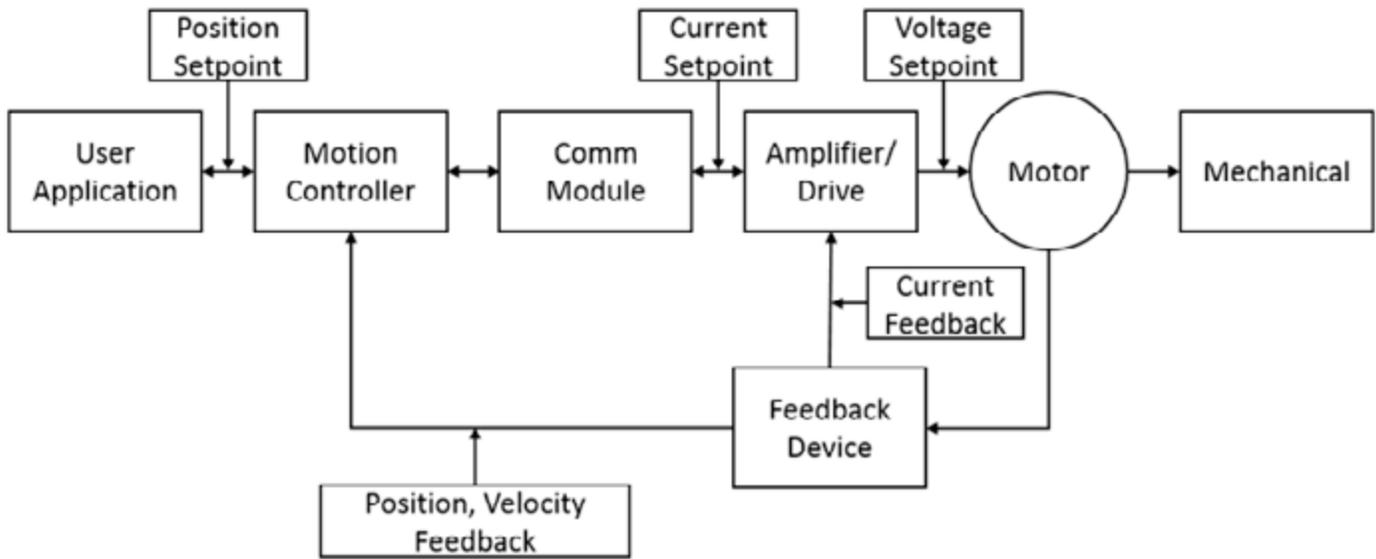


Figure 10.16. Simplified Motion System Diagram

The motion controller component of the system is divided into four components:

- **Supervisory control**—This top control loop executes command sequencing and passes commands to the trajectory generation loops. This loop performs the following:
 - System initialization, which includes homing to a zero position
 - Event handling, which includes triggering outputs based on position or sensor feedback and updating profiles based on user-defined events
 - Fault detection, which includes stopping moves on a limit switch encounter, safe system reaction to emergency stop or drive faults, and other watchdog actions
- **Trajectory generator**—This loop receives commands from the supervisory control loop and generates path planning based on the profile specified by the user. It provides new location setpoints to the control loop in a deterministic fashion. As a rule of thumb, this loop should execute with a 5 ms or faster loop rate.
- **Control loop(s)**—These are fast control loops that execute every 50 μ s. They use position and velocity sensor feedback and the setpoint from the trajectory generator to create the commands to the drive. Because these loops run faster than the trajectory generator, they also generate intermediate setpoints based on time with a routine called spline interpolation. For stepper systems, one of the control loops is replaced with a step generation component.
- **Feedback devices**—These are sensors such as encoders and limit switches that provide instantaneous position and velocity information to the drive/amplifier and the motion controller.

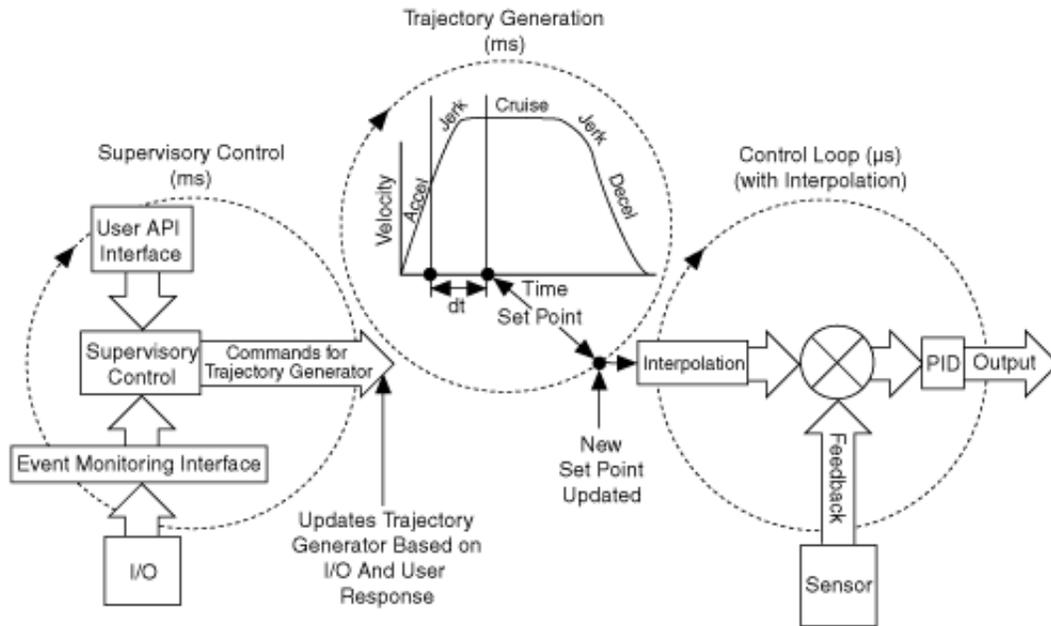


Figure 10.17. Functional Architecture of NI Motion Controllers

The LabVIEW NI SoftMotion Module provides these motion controller components implemented in the LabVIEW Real-Time and LabVIEW FPGA modules as well as an API in LabVIEW and C for creating motion control applications on top of these core components.

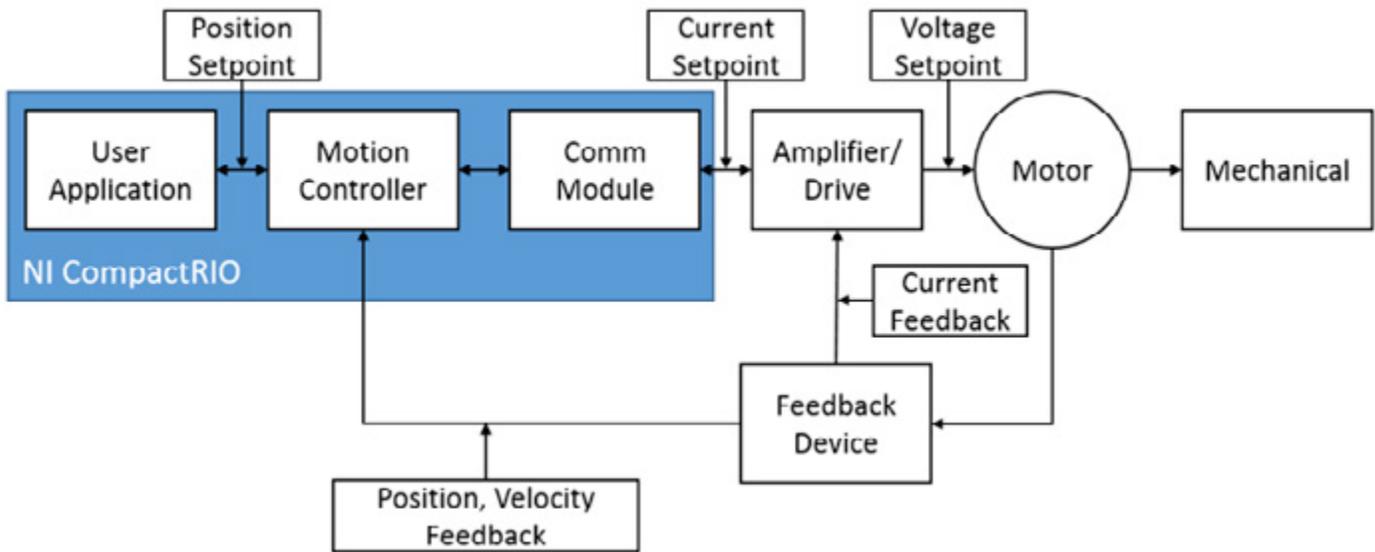


Figure 10.18. Simplified Motion System Diagram Showing Which Components Are Implemented in the LabVIEW RIO Architecture (in this case, with a CompactRIO controller)

NI SoftMotion Architecture

The LabVIEW NI SoftMotion Module contains components that run in LabVIEW for Windows, LabVIEW Real-Time, and LabVIEW FPGA. It also contains interfaces to a variety of motion specific devices and generic I/O.

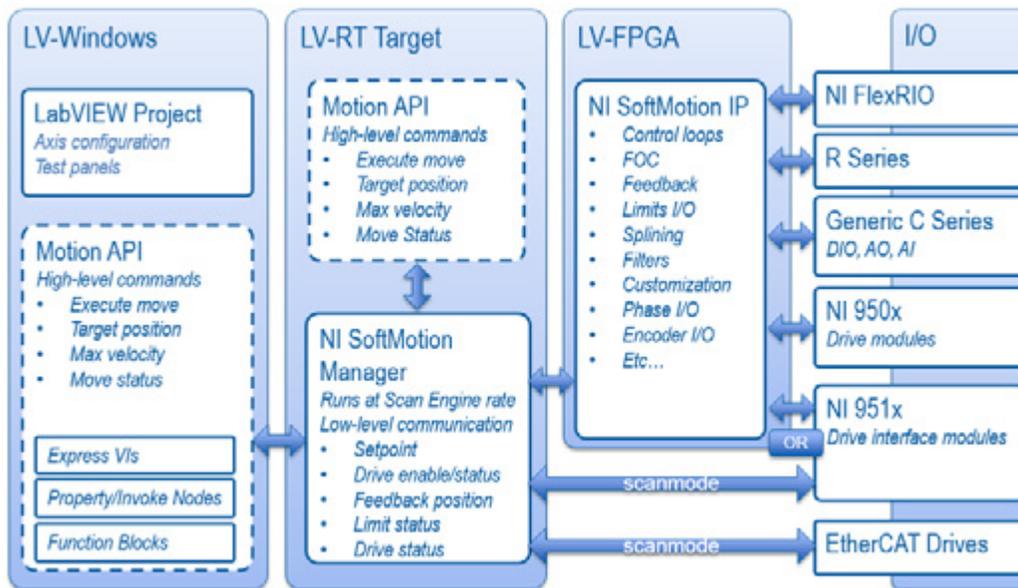


Figure 10.19. LabVIEW NI SoftMotion Components and Motion I/O Hardware Options

NI SoftMotion provides an architectural framework and common API that are hardware agnostic; it can be used to control a variety of different motion hardware configurations. NI SoftMotion is also implemented with open IP in the LabVIEW Real-Time and LabVIEW FPGA modules, so you can modify specific components as necessary but still use custom components with the other parts of the NI SoftMotion architecture. Because NI SoftMotion is designed to be hardware agnostic and open, it provides

- Easy coordination of different axis types
- High-level programming consistency
- Tight integration with other components running in LabVIEW Real-Time or LabVIEW FPGA
- Customization for unique application requirements

Examples of application-specific customization include FPGA triggering and timing based on other C Series I/O in the system, changing the trajectory generation method, or modifying the control-loop algorithms to implement a sensorless startup routine.

This section of the guide examines LabVIEW NI SoftMotion Module components and the motion-specific hardware with which they interface.

NI SoftMotion Project Items and APIs

PC-based NI SoftMotion components include the LabVIEW project-based configuration and high-level motion programming. All NI SoftMotion configuration and hardware setup is managed through the LabVIEW project. When NI SoftMotion is installed, you can access a list of NI SoftMotion items by right-clicking on a target in the system.

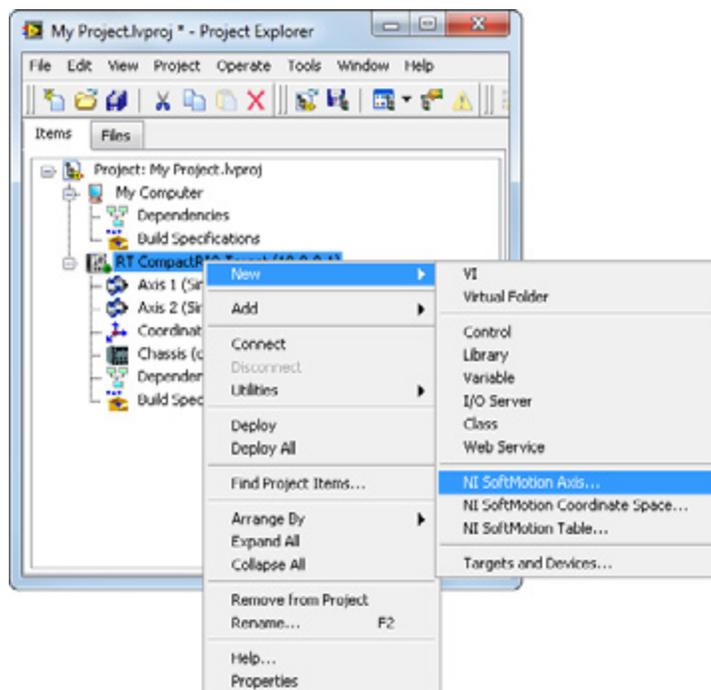


Figure 10.20. NI SoftMotion Resource Items Available From the LabVIEW Project

An NI SoftMotion axis is a resource that contains the settings and configuration for a specific motion drive, motor, or feedback device. You use axis resources on the block diagram to specify which resources interact with the NI SoftMotion API. An axis is bound to a specific hardware type to give it the necessary characteristics to configure and operate with that physical hardware. Axes are bound from the Axis Manager, which you access by right-clicking on an axis in the LabVIEW project.

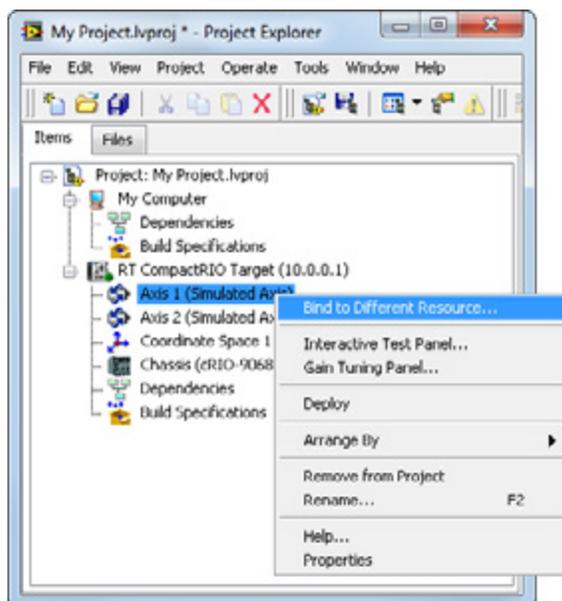


Figure 10.21. Access the Axis Manager by right-clicking on an axis in the LabVIEW project.

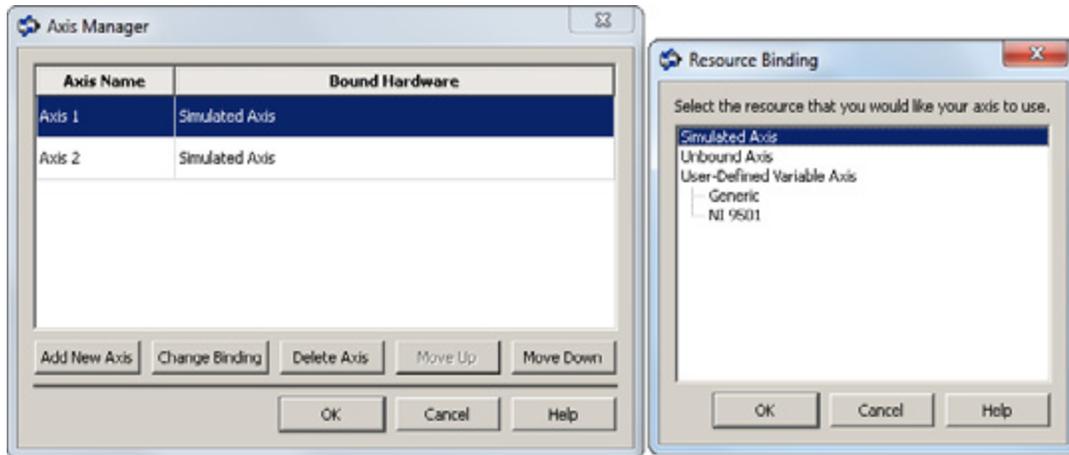


Figure 10.22. Use the NI SoftMotion Axis Manager and Resource Binding Dialog to associate an axis with a specific type of hardware.

NI SoftMotion contains the following axis types:

- Simulated axis**—Configure axes and create and execute motion applications even if you do not have hardware installed. This is useful for simple prototyping and setup, or for advanced applications as a trajectory master for electronic gearing and camming operations. However, because you are not connected to actual hardware, you cannot test control-loop settings or any I/O. The simulated axis, which is available only as a servo axis, features the following characteristics.

I/O Type	Value
Encoder 0 position	Trajectory generator setpoint
Encoder 0 velocity	Trajectory generator commanded velocity
Encoder 1 position and velocity	0
Analog input	Sine wave
Steps generated	0
Limits and Home input status	High or On
Position error, capture, watchdog, compare, fault status	FALSE
Digital input	Pulse

Table 10.2. A simulated axis returns simulated data for testing NI SoftMotion applications without requiring a physical drive and motor in the system.

- SolidWorks axis**—The SolidWorks axis type provides a connection to a motor defined in the SolidWorks Motion Analysis add-on. With this axis type, you can synchronize Dassault Systemes SolidWorks assemblies for motion system simulation. Using NI SoftMotion with SolidWorks to simulate your system with actual motion profiles, you can design motion profiles; detect collisions; simulate the mechanical dynamics of your machine including mass and friction effects; estimate machine cycle time performance; validate component selections for motors, drives, and mechanical transmissions; and evaluate engineering trade-offs between the mechanical, electrical, control, and embedded system aspects of the design. For more details, see the [Virtual Prototyping Training Series](#).

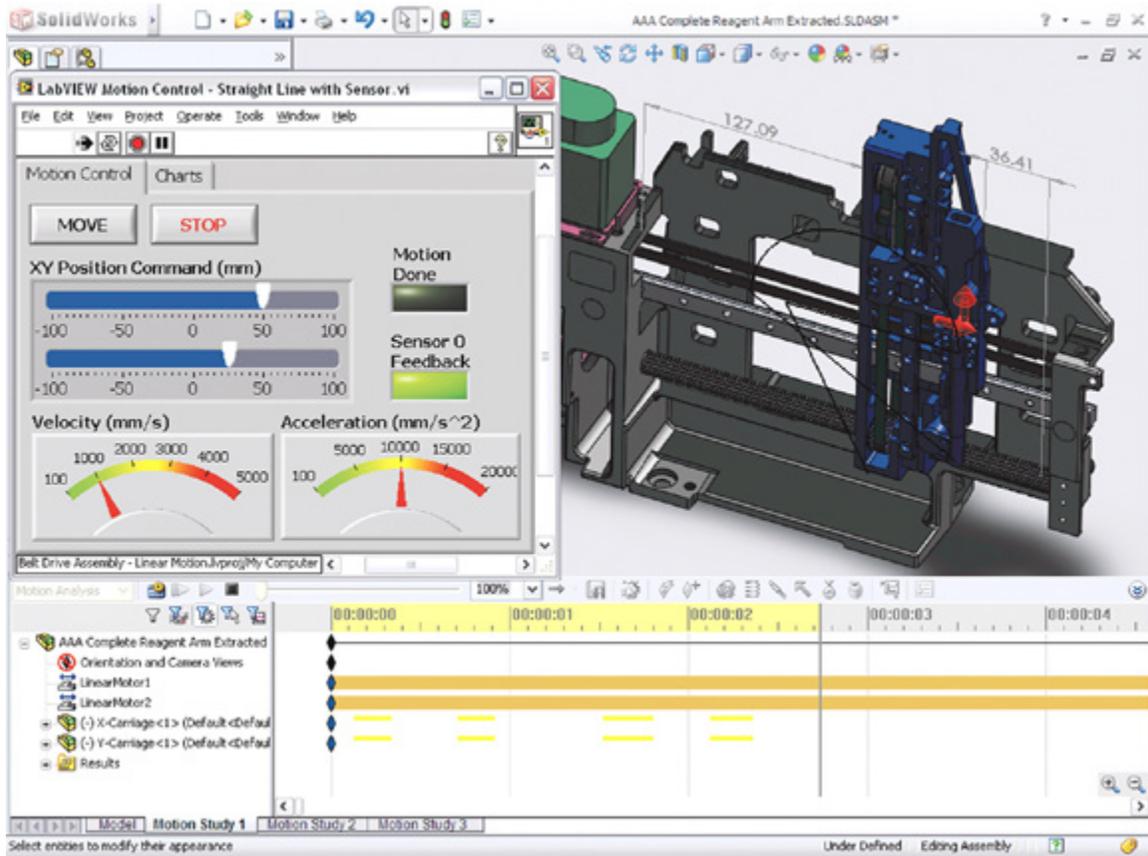


Figure 10.23. Interface NI SoftMotion with a SolidWorks assembly through the LabVIEW project to control the model with LabVIEW code.

- **NI 951x axis**—Use NI 951x C Series modules directly from LabVIEW Real-Time with the NI Scan Engine, or use these modules in FPGA mode with the FPGA I/O nodes for the modules.
- **EtherCAT drive**—Configure and use an EtherCAT AKD servo drive completely through NI SoftMotion. You do not need to use the I/O variables under the EtherCAT device when using the EtherCAT AKD drive with NI SoftMotion.
- **Unbound axes**—Create custom motion applications in situations where the system requires features or functionality not available in other motion controllers. This may be a non-supported communication interface, specialized I/O, or customized control algorithms for precise control. If you use unbound axes with the Axis Interface nodes, your main application VI can use the same NI SoftMotion APIs for all axis types, which means that you can have unbound axes and axes bound to other resources in the same application. Unbound axes are used with most NI 950x C Series drive modules and when implementing motion control with generic C Series I/O modules.
- **User-defined variable axes**—Use user-defined variable (UDV) axes to implement an interface for communication between the NI SoftMotion Engine and the LabVIEW FPGA Module. UDV axes allow your main application VI to use the same NI SoftMotion API for all axis types, which means that you can have UDV axes and axes bound to other resources in the same application. The UDV communication module handles communication between the NI SoftMotion Engine and the LabVIEW FPGA VI using user-defined variables that you add to the project. NI SoftMotion supports the following UDV axis types:

- **Generic user-defined variable axis**—Works with UDV axes not associated with a currently supported C Series module. When you use a generic UDV axis, the axis configuration dialog box contains all available configuration options.
- **NI 9501 user-defined variable axis**—Works with UDV axes associated with an NI 9501 C Series stepper drive module. When you use an NI 9501 UDV axis, the axis configuration dialog box contains configuration options customized for the NI 9501 properties.

As the above list attests, the axis is an abstraction that you can use to bind various types of hardware to it. The benefit to this is the ability to combine dissimilar axis types in the same VI using a consistent high-level programming API. You can prototype a program using a simulated axis and then deploy it to a real-time system, change the axis binding to the real hardware, and run the code on the real-time system without changing anything about the way the high-level code is written.

Right-click on each axis to bind that axis to a specific physical interface or drive in the system, and to configure the settings and properties for that axis.

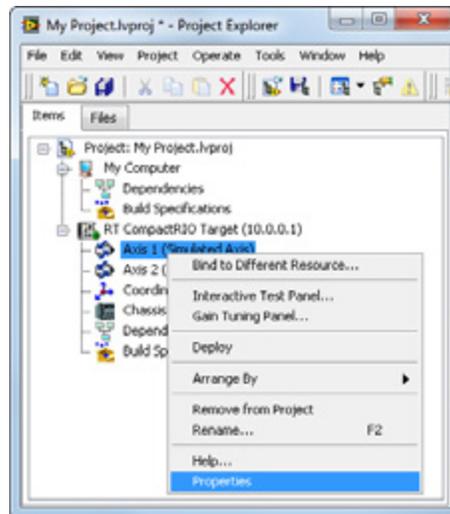


Figure 10.24. Right-click on an axis in the LabVIEW project to bind it to a physical resource, change its properties, or test the settings with the interactive test panel.

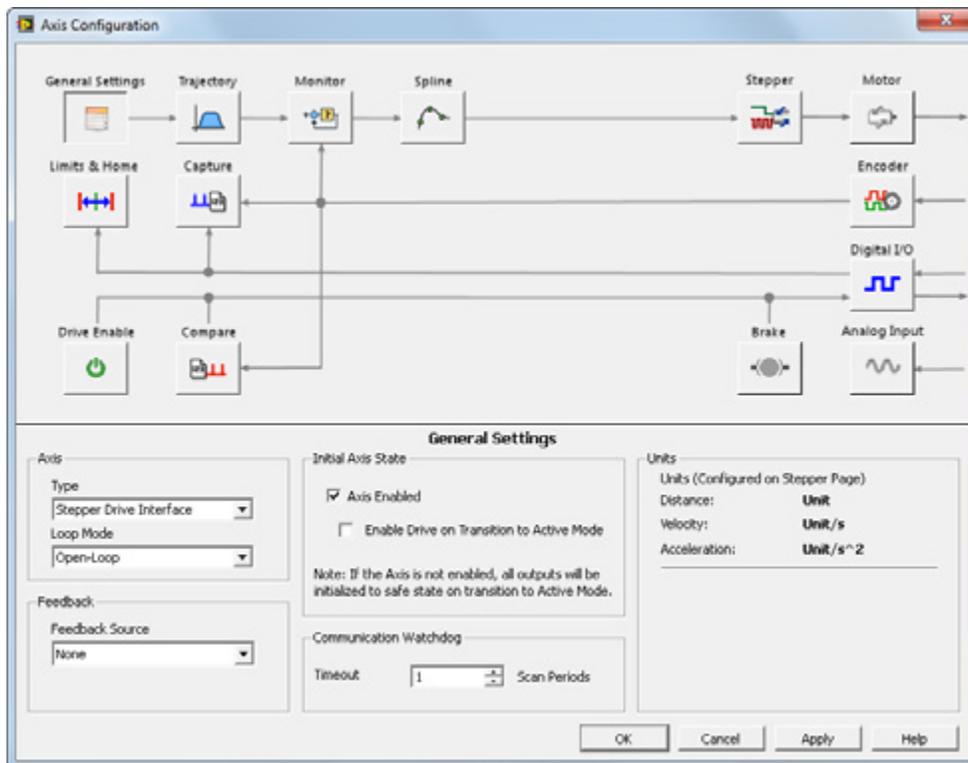


Figure 10.25. The items and settings on the Axis Configuration panel change depending on the axis type.

You can use the Interactive Test Panel to test and debug your motion system. With the Interactive Test Panel, you can perform a simple straight line move and monitor move and I/O status information, change move constraints, obtain information about errors & faults in the system, and view the position or velocity plots of the move. If you have a feedback device connected to your system, you can also obtain feedback position and position error information.

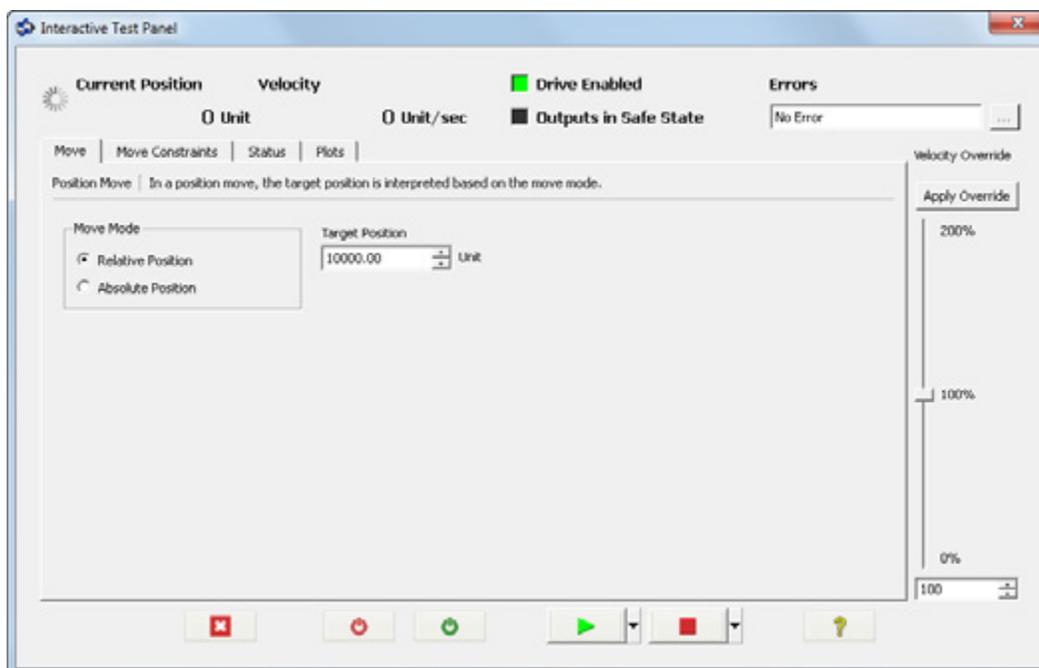


Figure 10.26. Perform test moves, check limit switch settings, enable/disable drives, and see any errors from the Interactive Test Panel.

To open the interactive window, right-click the axis in the LabVIEW Project Explorer window and select Interactive Test Panel. Set the desired position, move mode, and move constraints using the tabs. Click the Start button on the bottom of the dialog box to start the move with the configured options. Use the Status and Plots tabs to monitor the move while it is in progress.

Once you configure an axis, you use it on the LabVIEW block diagram as a reference to tell the NI SoftMotion API which axis to use when executing functions. You can drag and drop axes from the project to the block diagram or place them from the NI SoftMotion palette.

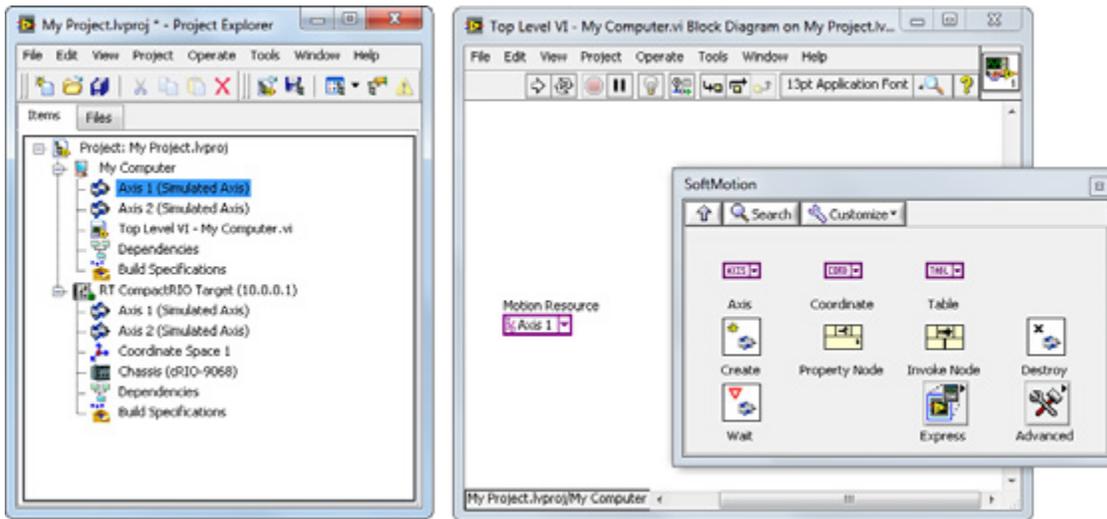


Figure 10.27. You can drag an Axis resource from the LabVIEW project to the block diagram or drop it from the palette and associate it with an item in the LabVIEW project.

When you install LabVIEW NI SoftMotion, you get a palette of motion VIs that includes two APIs: the Property/Invoke node API and the Express VI API.

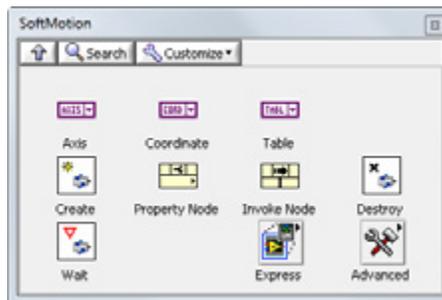


Figure 10.28. The NI SoftMotion Palette

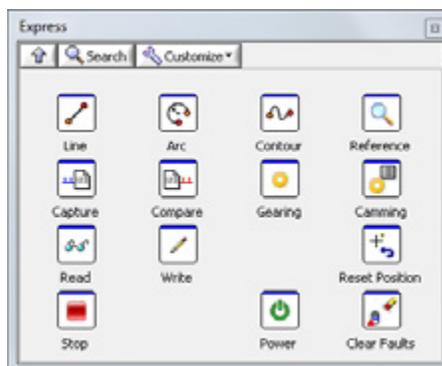


Figure 10.29. The NI SoftMotion Express VI Palette

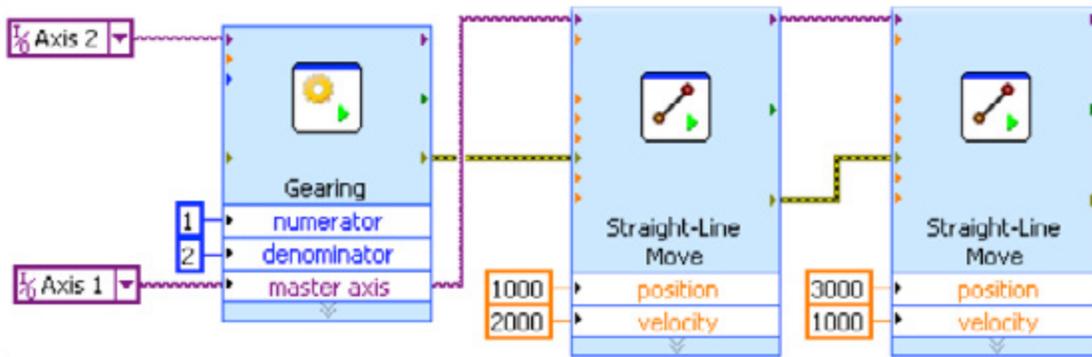


Figure 10.30. Axis 2 is geared to Axis 1 with a 1:2 gear ratio for the two straight line moves shown.

The Express VI API contains Express configuration dialogs with visualizations that help you quickly put together an application. Double-clicking on an Express VI opens up the properties dialog where you can configure the Express VI node, see the visualizations, and set parameter values.

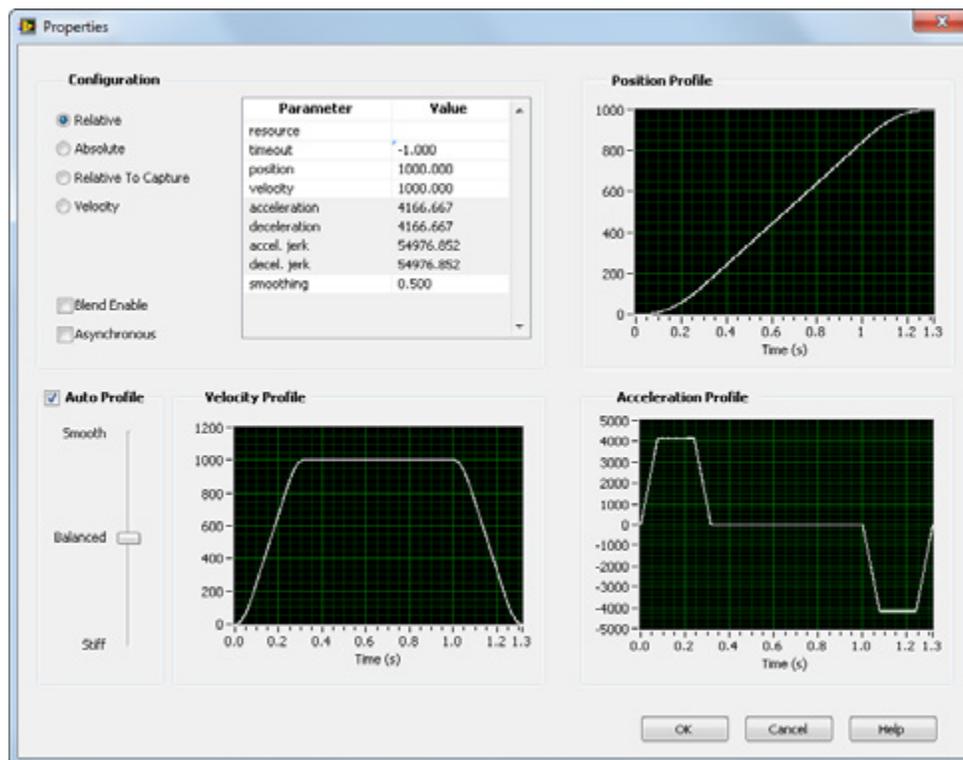


Figure 10.31. Use the Straight Line Move Express VI Properties Page to configure settings, change values, and see the resulting profiles.

Palette Object	Palette Symbol	Description
Line		Performs a straight-line move using an axis or coordinate resource. A straight-line move connects two points using one or more axes. The behavior of the move changes based on the Straight-Line Move Mode.
Arc		Performs a circular, spherical, or helical arc move. An arc move produces motion in a circular shape using a radius you specify. The type of arc to perform changes based on the Arc Move Mode.
Contour		Performs a contour move using an axis or coordinate resource. A contour move is a move expressed as a series of positions that the software uses to extrapolate a smooth curve. These positions are stored in a table. Each point in the move is interpreted as an absolute position using the starting point of the move as a temporary "zero" position. The type of contour move changes based on the Contour Mode.
Reference		Performs a reference move, such as locating a home or limit position, on an axis resource. Reference moves are used to initialize the motion system and establish a repeatable reference position. The behavior of the move changes based on the Reference Move Mode.
Capture		Records encoder position based on an external input, such as the state of a sensor. You can use the captured position to execute a move relative to a captured position, or simply record the encoder position when the capture event occurs.
Compare		Synchronizes the motor with external activities and specified encoder positions. When the specified position is reached, a user-configurable pulse is executed. The behavior of the position compare operation changes based on the Compare Mode.
Gearing		Configures the specified axis for gearing operations. Gearing synchronizes the movement of a slave axis to the movement of a master device, which can be an encoder or the trajectory of another axis. The movement of the slave axes may be at a higher or lower gear ratio than the master. For example, every turn of the master axis may cause a slave axis to turn twice. The type of gearing operation to perform changes based on the Gearing Mode.
Camming		Configures the specified axis for camming operations. These ratios are handled automatically by LabVIEW NI SoftMotion, allowing precise switching of the gear ratios. Camming is used in applications where the slave axis follows a nonlinear profile from a master device. The type of camming operation changes based on the Camming Mode.
Read		Reads status and data information from axes, coordinates, feedback, and other resources. Use the read methods to obtain information from different resources.
Write		Writes data information to axes, coordinates, or feedback resources. Use the write methods to write information to different resources.
Reset Position		Resets the position on the specified axis or coordinate.
Stop		Stops the current motion on an axis or coordinate. The behavior of the move changes based on the Stop Mode.
Power		Enables and disables axes and/or drives on the specified axes or coordinate resources.
Clear Faults		Clears LabVIEW NI SoftMotion faults.

Table 10.3. Motion Express VI Overview

You can configure NI SoftMotion Express VIs for either synchronous or asynchronous operation by right-clicking and selecting **Timing Model** from the menu.

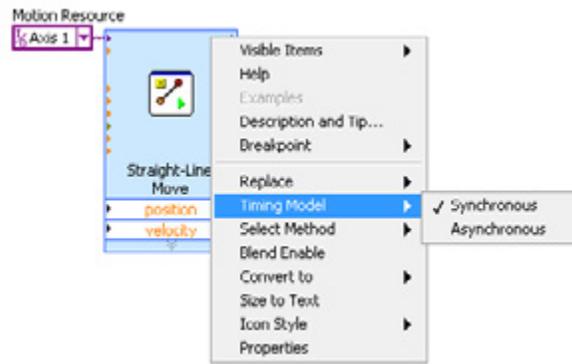


Figure 10.32. Select the timing model for each Express VI by right-clicking and selecting Timing Model.

When configured for synchronous operation, Express VIs wait until the node finishes executing before returning, and you can use standard LabVIEW programming methods (dataflow).

When configured for asynchronous operation, the Express VIs return immediately, and you must control execution by using the status parameters provided: execute, error out, done, aborted, busy, and active.

These two programming methods are very different, as illustrated from the two versions of a simple straight line move example shown in figures 10.33 and 10.34.

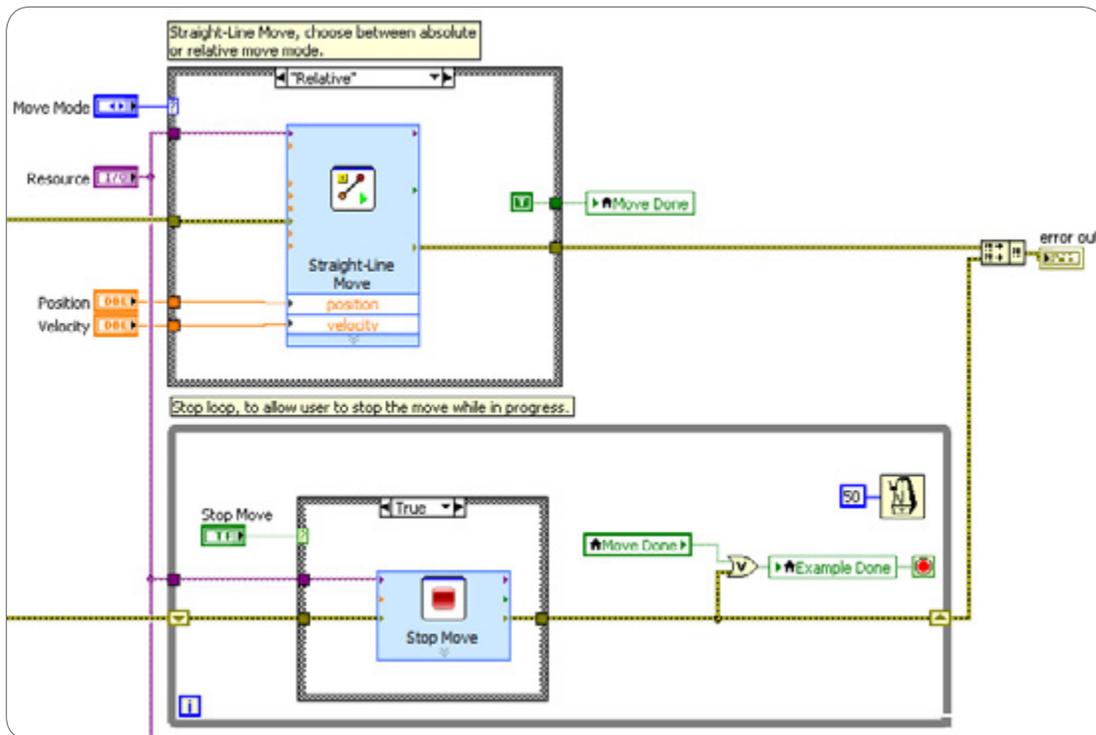


Figure 10.33. Straight Line Move Example Using Synchronous Express VIs

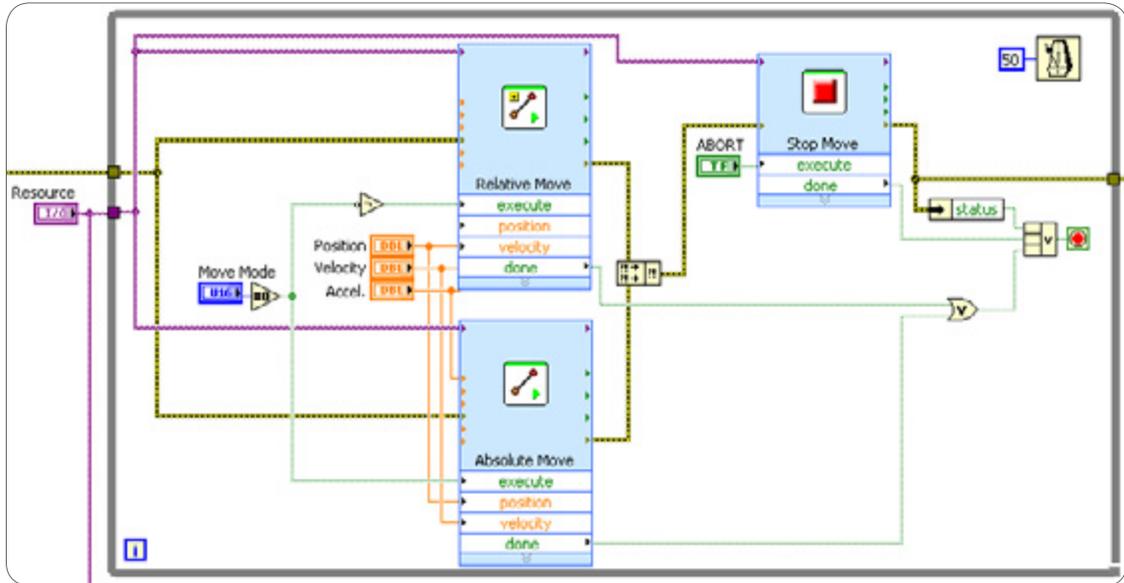


Figure 10.34. Straight Line Move Example Using Asynchronous Express VIs

You can convert Express VIs to subVIs that uses the Property/Invoke node API by right-clicking on the Express VI and selecting **Convert To»SubVI**.

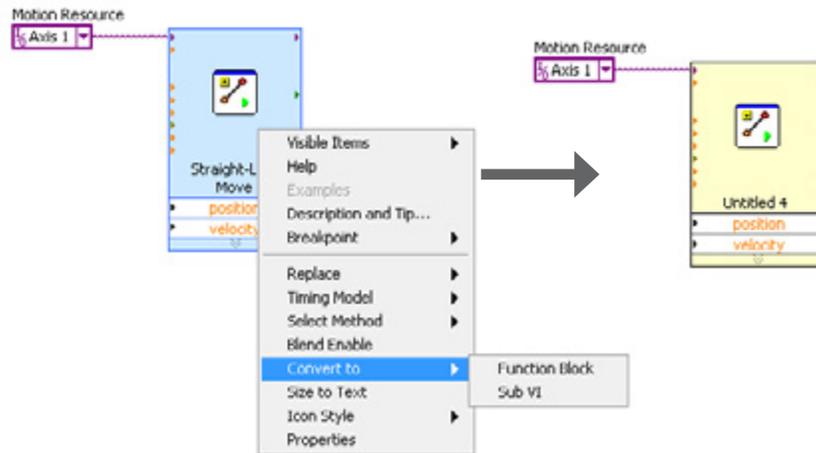


Figure 10.35. Right-click on an NI SoftMotion Express VI to convert it to a subVI.

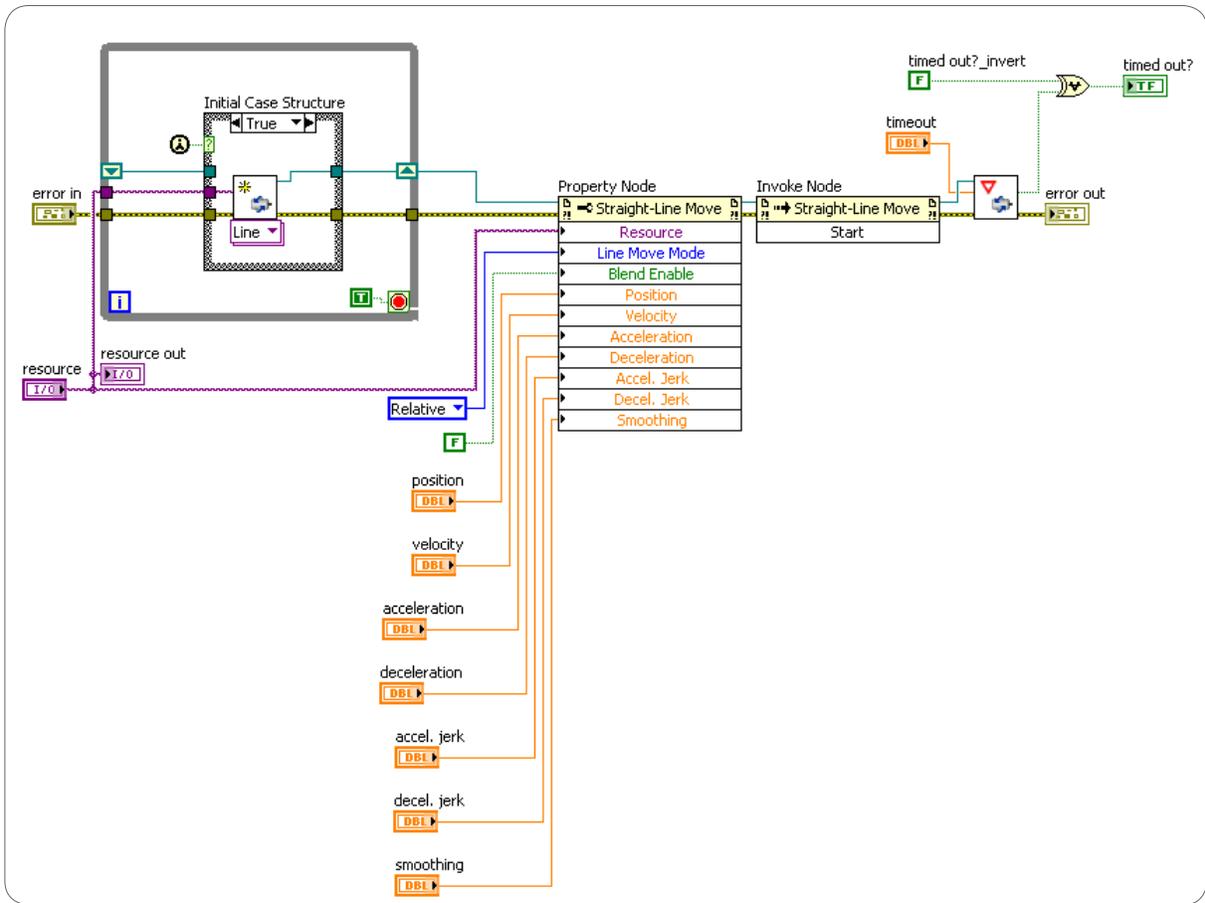


Figure 10.36. Resulting Block Diagram After Converting the Straight Line Move Express VI to a SubVI

See examples of using both the Express VI and Property/Invoke node APIs in the NI Example Finder.

The Property/Invoke node API provides functionality similar to the Express VI API but with a more granular level of control over how NI SoftMotion processes and executes commands. In addition to the axis resource, the Property/Invoke node API uses the concept of a motion resource, which creates a reference to specific NI SoftMotion interface objects.

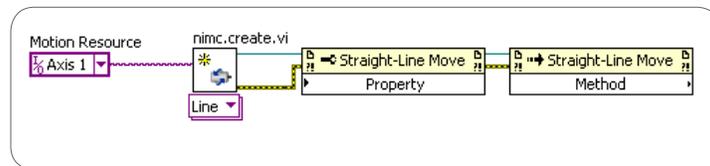


Figure 10.37. The NI SoftMotion Property/Invoke node API uses motion resources as well as axis resources.

With this property node, you can configure all of the same properties and settings available from the Express VI configuration dialog. But where the Express VI execution is determined by dataflow in the synchronous case and by the done terminal in the asynchronous case, the Property/Invoke node API execution is controlled through the Invoke node.

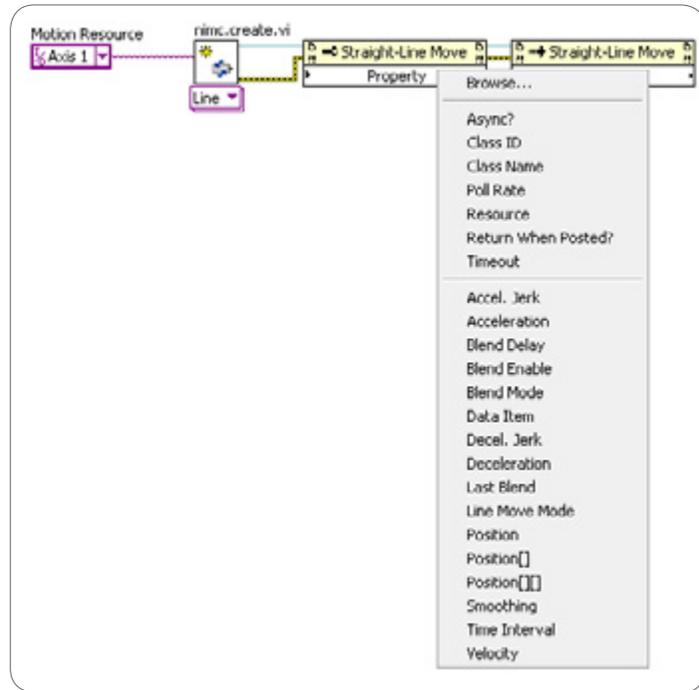


Figure 10.38. Available Properties of a Straight Line Move

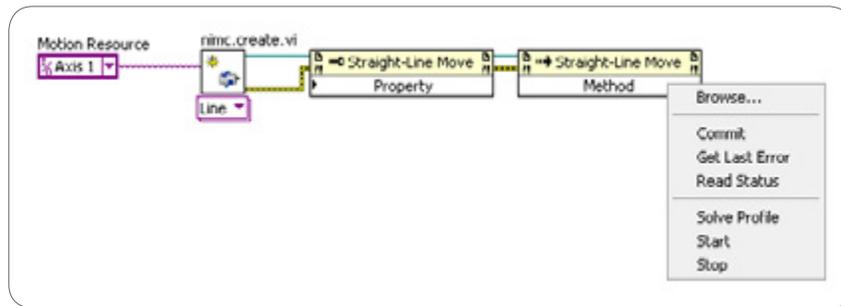


Figure 10.39. Available Methods of a Straight Line Move

You can also use the Property/Invoke node API to programmatically configure settings for all axes in the LabVIEW project.



Figure 10.40. All the same settings that you can configure through the axis in the LabVIEW project can also be configured programmatically through the Property/Invoke node API.

Which API to use is largely determined by your application architecture. Use the Property/Invoke node API when you need tight synchronization with the rest of your application or when you use a state machine architecture. In all other cases, use the NI SoftMotion Express VI API, which is lean and efficient.

You can program VIs written using these motion APIs and deploy them to the Windows machine, or you can run them in LabVIEW Real-Time on a CompactRIO controller. If you deploy and run the VIs on a Windows system while using resources deployed to a CompactRIO controller, then use Web Service APIs to act on those resources remotely. Typically, it is best to deploy the code to LabVIEW Real-Time and then implement a communications protocol back to a high-level API to execute certain move sequences and interact with the application.

Real-Time-Based NI SoftMotion Components: The NI SoftMotion Engine

As mentioned, you can run the high-level motion code on the host PC in LabVIEW for Windows or in LabVIEW Real-Time on the CompactRIO controller. All the deterministic components of the motion system also run on the real-time system, including trajectory generation and supervisory control. The NI SoftMotion Manager runs at the scan rate of the CompactRIO controller and communicates between the high-level API code and the various motion hardware and I/O set up through the axis in the system.

FPGA-Based NI SoftMotion Components

The goal for NI SoftMotion low-level implementation on the FPGA is to keep the high-level API programming but add FPGA flexibility and performance. The following section provides an overview of the motor control IP included in NI SoftMotion.

When you open a VI targeted to the CompactRIO FPGA, you can see an NI SoftMotion palette that includes the following motor control IP:

- PID/PI
- PWM
- FOC
- I²T
- Motion I/O
- Feedback
- Filters
- Stepper

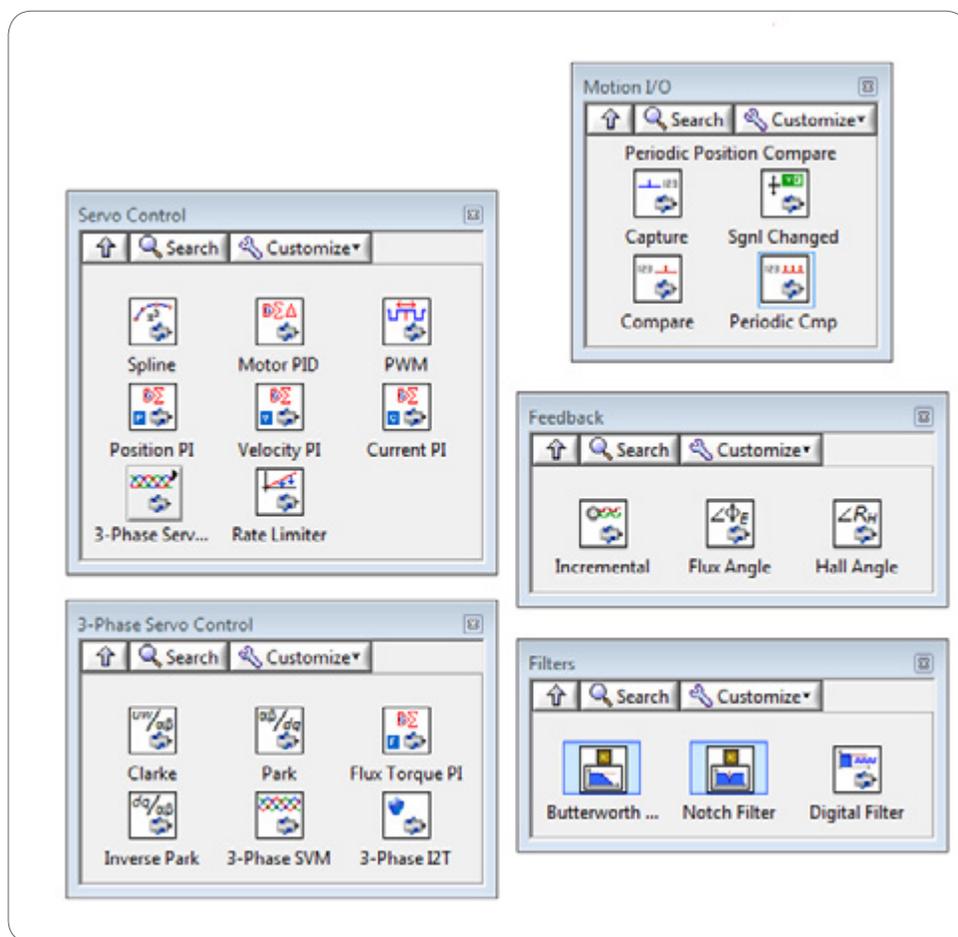


Figure 10.41. NI SoftMotion Motor Control IP Palettes in LabVIEW FPGA

Use this IP to build all the necessary components to control a motion axis. There are two basic elements to controlling a torque amplifier: control and feedback. The control loop provides a current setpoint and the feedback loop provides an encoder position.

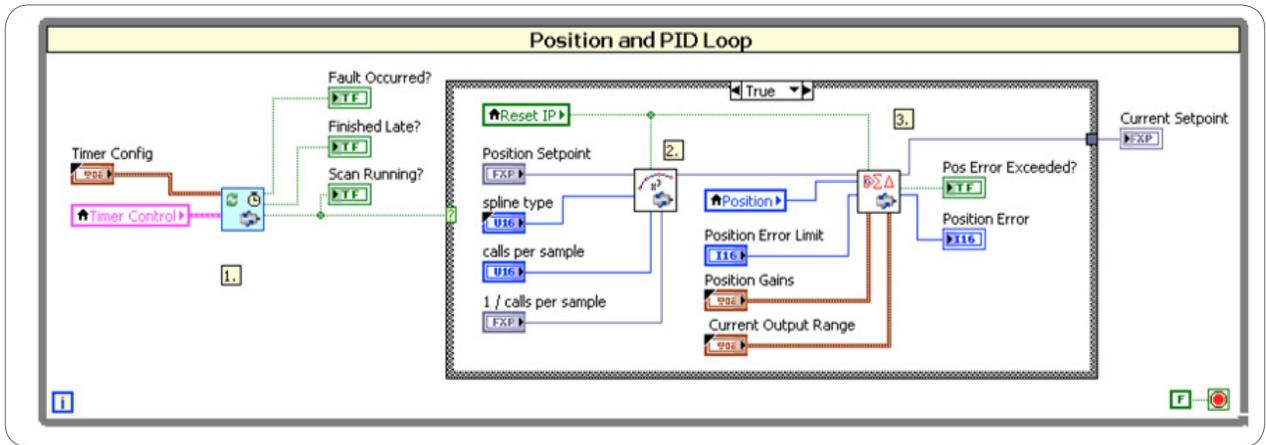


Figure 10.42. Spline Interpolation and PID Loop Executed in LabVIEW FPGA With NI SoftMotion IP

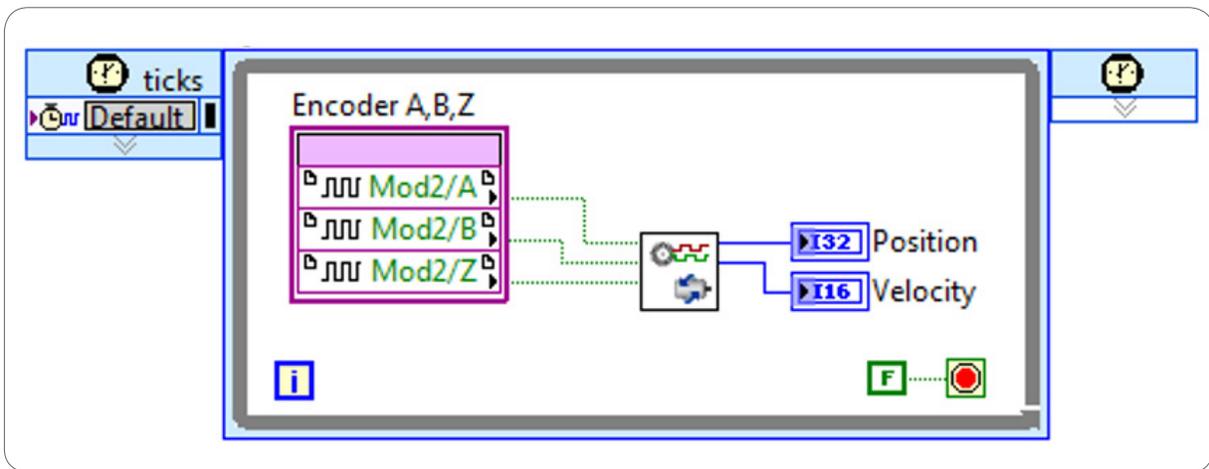


Figure 10.43. Incremental (Quadrature) Encoder Decoder

To control the NI 9505 brushed DC motor drive module, you also need current control and PWM control of H-bridges.

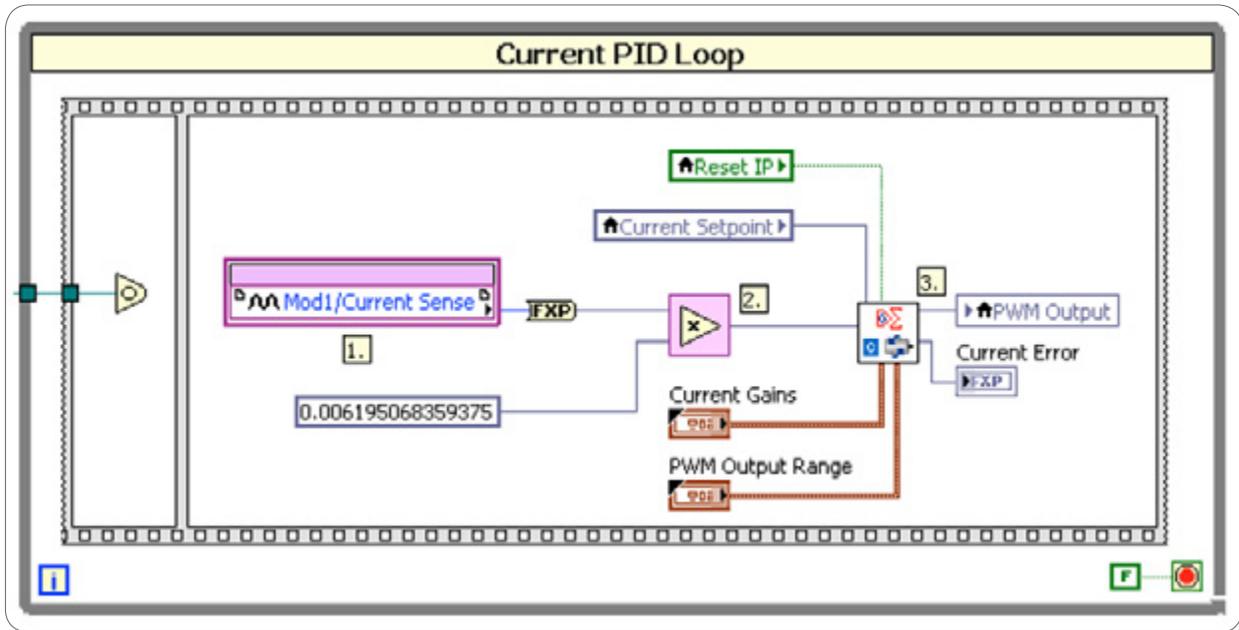


Figure 10.44. Current Control IP

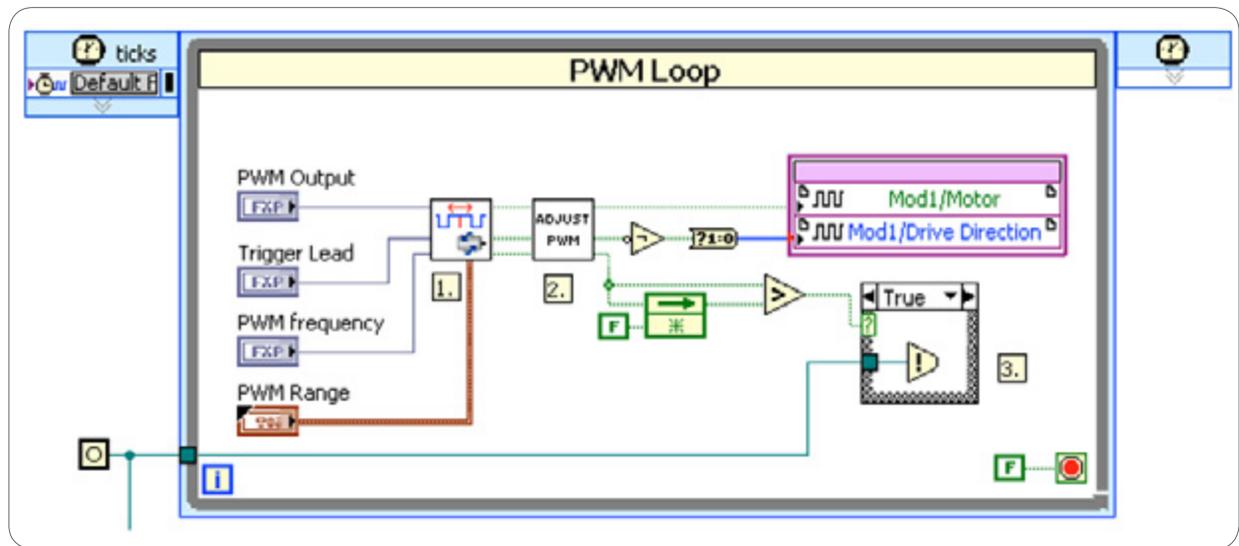


Figure 10.45. PWM Control of H-Bridges

To control a brushless servo drive, you need to use two new elements: field oriented control and Hall effect sensors. This additional complexity is necessary because a brushless motor is electrically commutated by the drive instead of mechanically commutated due to the inherent construction of the motor.

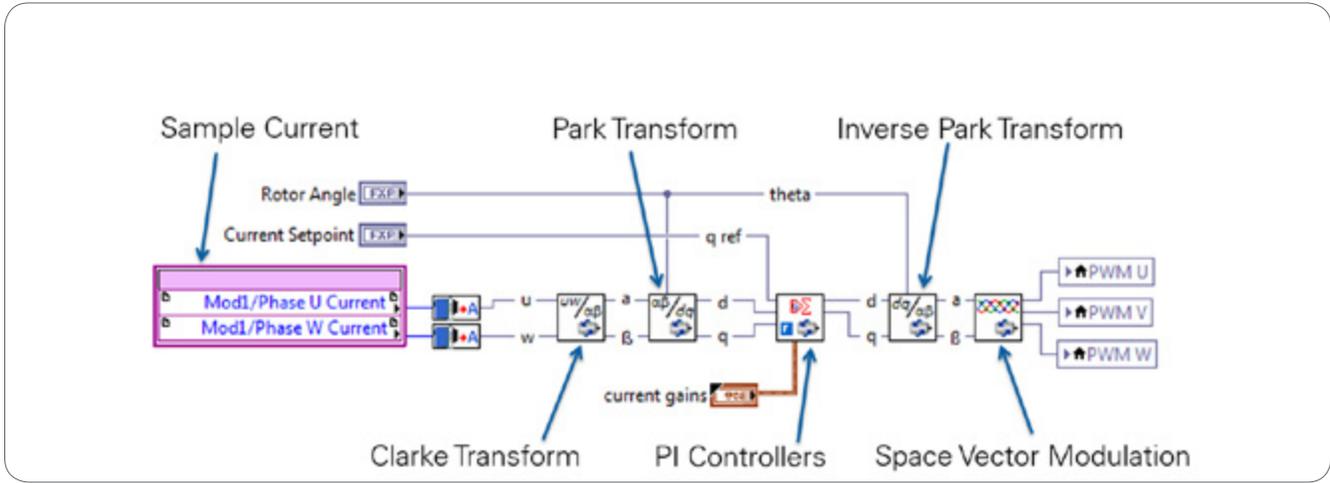


Figure 10.46. Field Oriented Control (FOC) Algorithm Implemented in LabVIEW FPGA

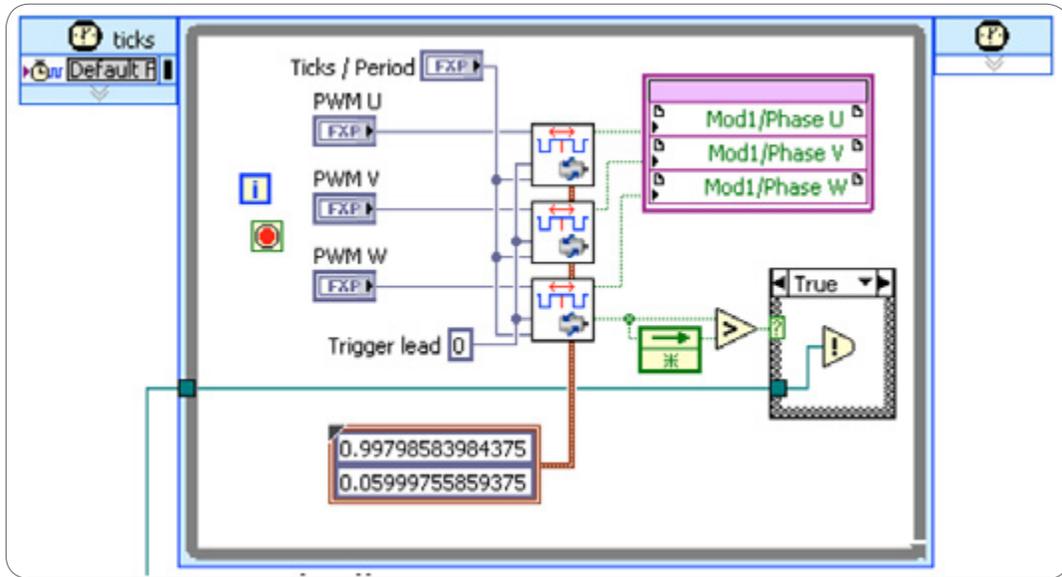


Figure 10.47. Three PWM Generators, One for Each Motor Phase

Use Hall effect sensor inputs to determine the angle of the motor shaft, and use the angle upon startup to eliminate a jump in rotor position.

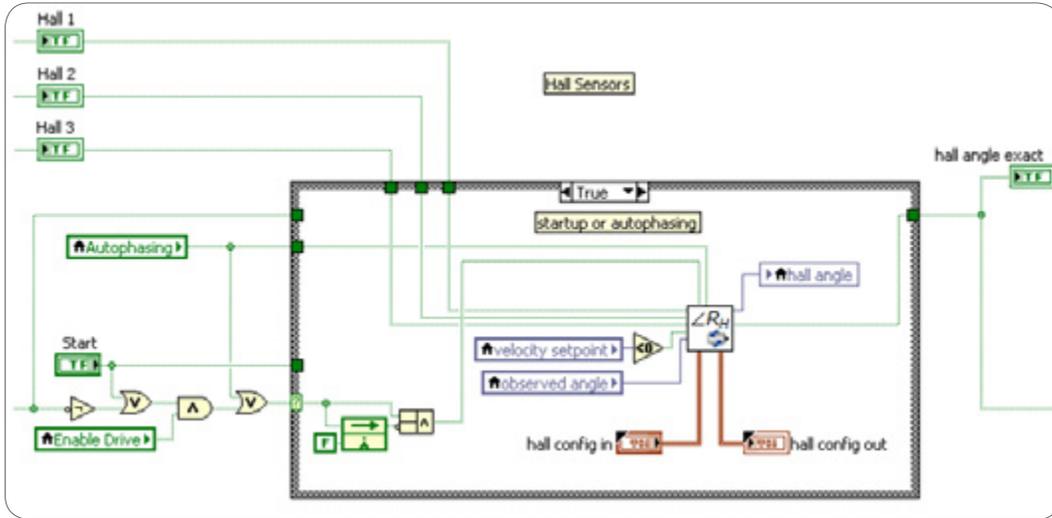


Figure 10.48. Hall Sensors for Initialization

Motion I/O is also implemented through the LabVIEW FPGA Module.

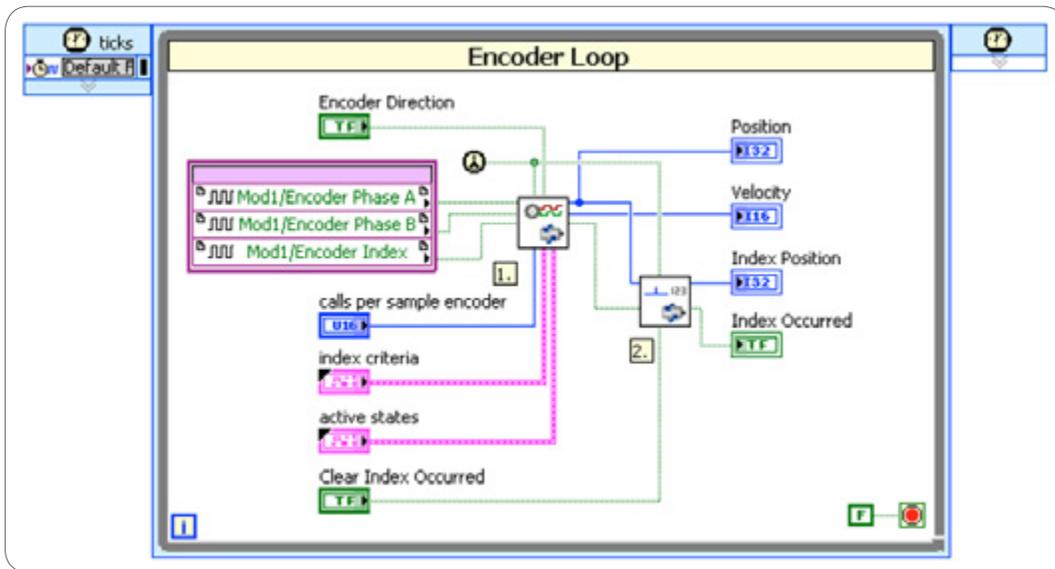


Figure 10.49. Encoder Loop With Index Read Functionality

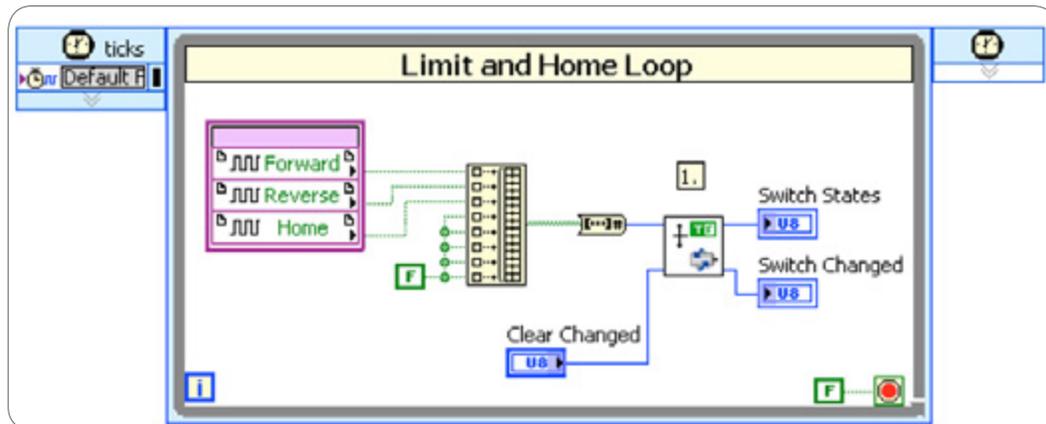


Figure 10.50. Limit and Home Loop

Consider an example application for which you need to trigger gripper action based on multiple conditions:

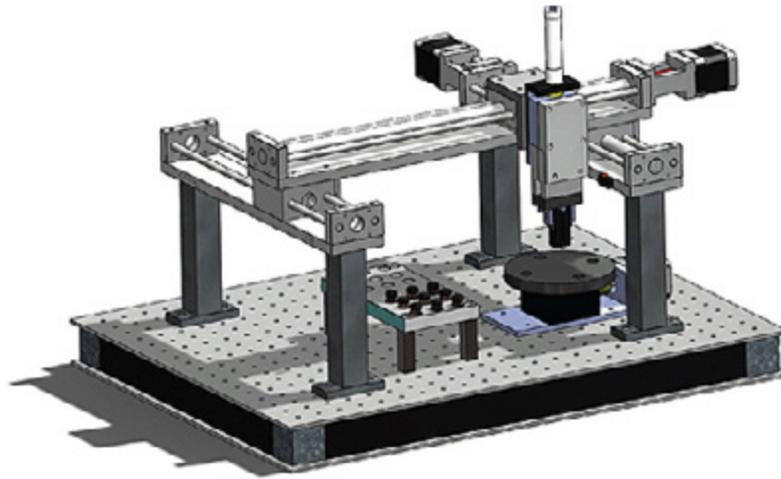


Figure 10.51. Example Gantry System

1. Position of the gantry (X, Y, and Z)
2. Rotation angle of the stage
3. Force exerted by the gripper

It's possible to get the position of the gantry (X, Y, Z, and Θ) at 25 ns intervals from the single-cycle Timed Loop housing the quadrature decoder code.

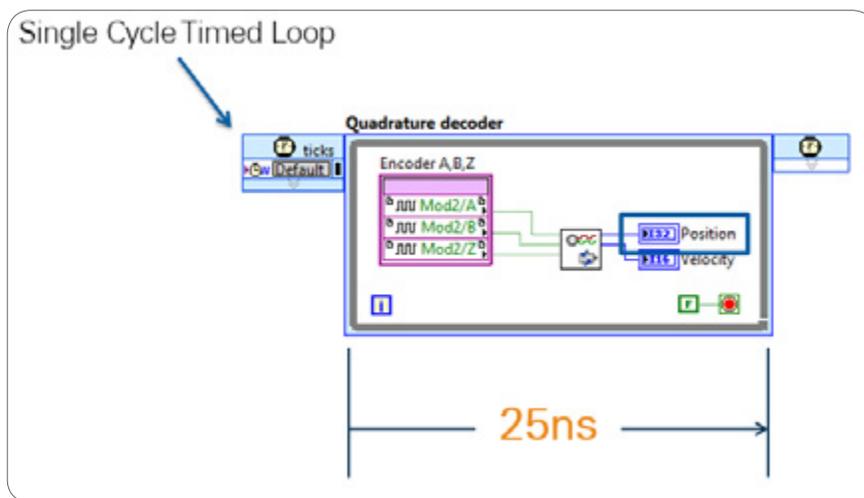


Figure 10.52. You can read position from the quadrature decoder loop.

You can determine the force of the gripper from the sampled quad current in the FOC algorithm every 1.4 μ s.

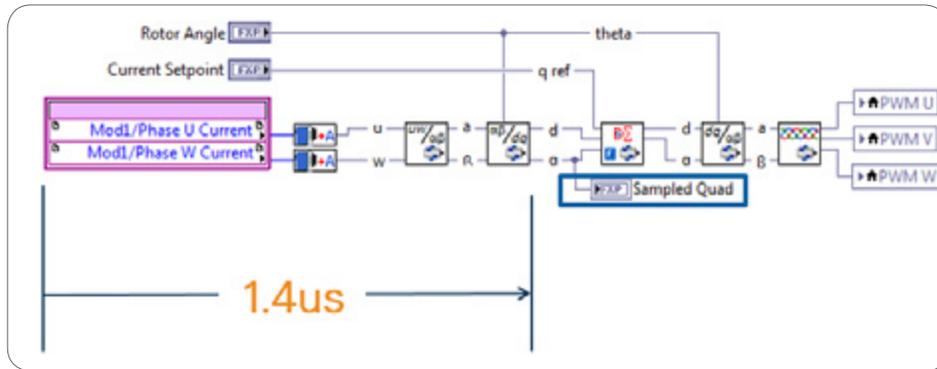


Figure 10.53. FOC Algorithm

You can then define this information in a user-defined trigger running in a single-cycle Timed Loop on the FPGA and executing as fast as every 25 ns.

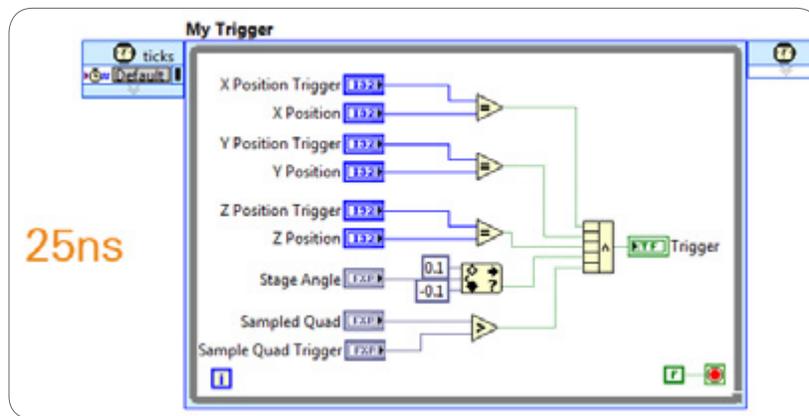


Figure 10.54. User-Defined Trigger

NI SoftMotion has IP for triggering, including position compare, position capture, and periodic position compare functionality. These functions are full featured with enable/disable, pulse width, periodic trigger, digital filter, and trigger window capabilities. They are single-cycle Timed Loop compatible and open so that you can modify them if needed.

Hardware Interfaces to NI SoftMotion

You can choose from a variety of hardware interfaces to NI SoftMotion. The LabVIEW NI SoftMotion Module is written so that you can write custom hardware interfaces for specific hardware such as third-party drives or C Series modules. The most common hardware interfaces are

- NI SoftMotion and EtherCAT AKD drives
- NI SoftMotion and C Series drive interface modules (NI 951x)
- NI SoftMotion and C Series drive modules (NI 950x)

NI SoftMotion and EtherCAT AKD Drives

NI SoftMotion contains an interface to the EtherCAT AKD servo drive. Use this interface when drives are connected to an NI real-time controller over EtherCAT.



Figure 10.55. Any CompactRIO controller with two Ethernet ports can act as an EtherCAT master to control multiple daisy-chained AKD drives from the second Ethernet port.

All NI real-time targets with two Ethernet ports and supported by the NI-Industrial Communications for EtherCAT driver are configurable as NI EtherCAT masters. This includes PXI controllers, NI industrial controllers, the NI 9792 programmable wireless sensor network gateway, and CompactRIO controllers.

You can add coordinated high-performance multi-axis servo control to any system based on these NI real-time controllers. Which controller to select depends on the number of axes in the system (amount of data being processed each scan cycle on the EtherCAT bus) and on your desired control-loop rate. The higher the axis count, the more data that must be processed within a scan cycle. The faster the control-loop rate, the less time the controller has to complete that processing. These amount of data on the bus and control-loop rate determine the practical limit on the number of slave devices allowed in a particular system.

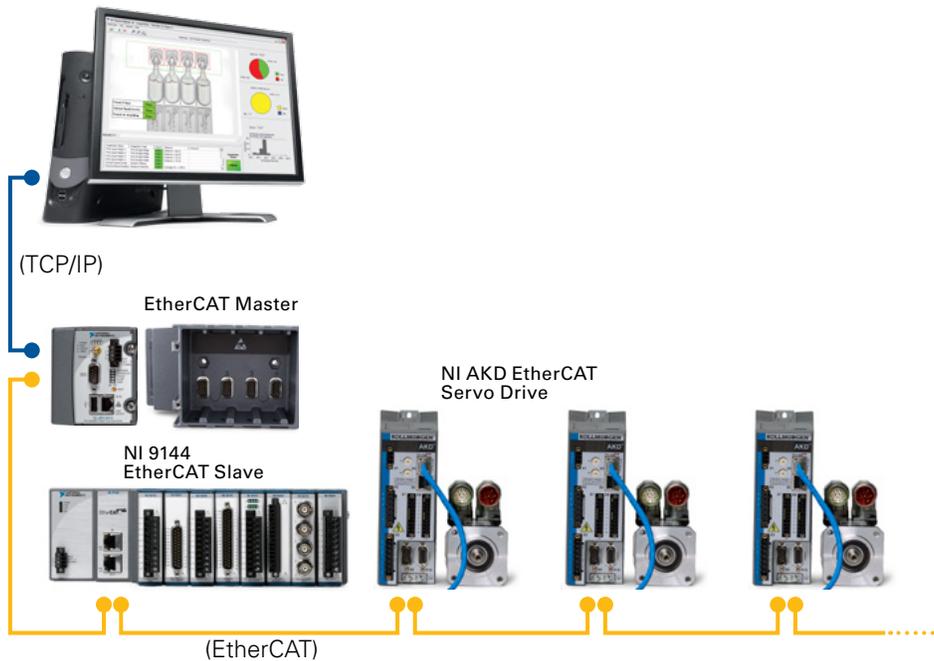


Figure 10.56. CompactRIO acts as the EtherCAT master. You can add multiple NI 9144 and EtherCAT AKD slaves to extend this system up to the practical limit imposed by the processing power of the EtherCAT master.

When added to the system, EtherCAT drives show up under the EtherCAT master in the system. Then you can bind them to an axis.

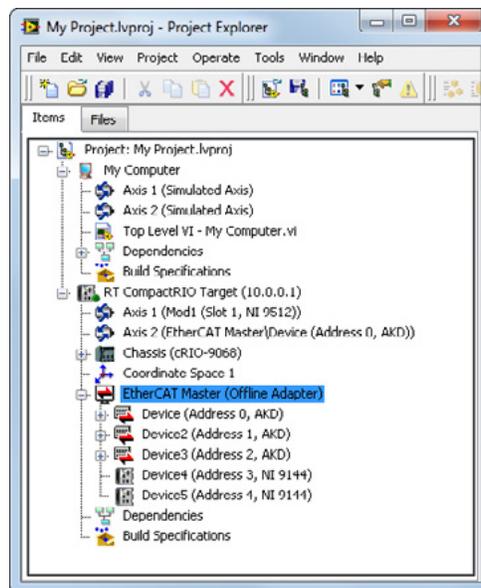


Figure 10.57. EtherCAT devices show up under the EtherCAT master adapter (second Ethernet port of the CompactRIO system). You can bind EtherCAT AKD drives to an NI SoftMotion axis as shown.

Once bound to an axis, NI SoftMotion takes care of all drive communication over EtherCAT, and you can use that axis with the high-level motion APIs. This is because NI SoftMotion includes a specific drive interface and LabVIEW project integration for the AKD drive. This drive interface abstracts all the EtherCAT programming and variables; just use the axis bound to the AKD with the high-level NI SoftMotion API.

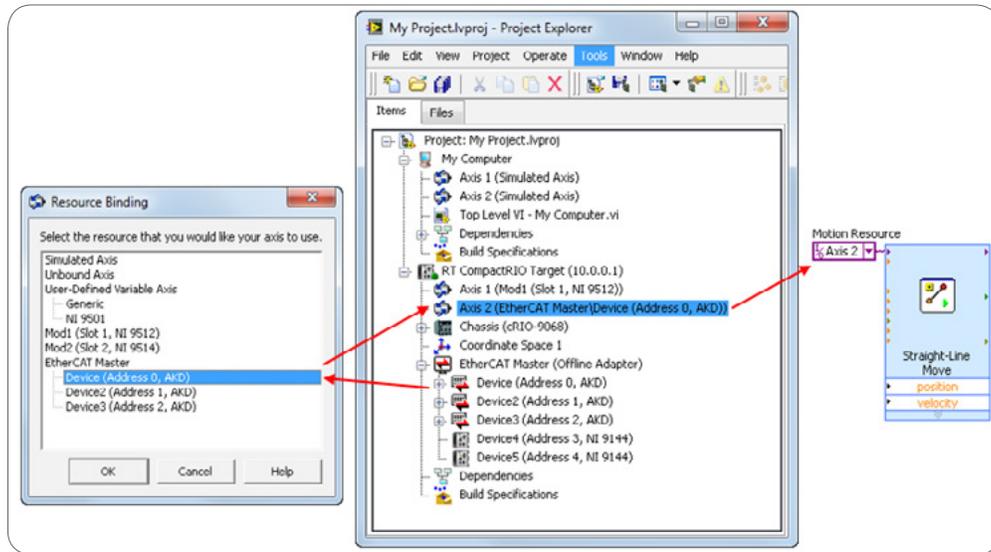


Figure 10.58. AKD EtherCAT Drives can be bound to an axis, and then directly used with the NI SoftMotion API

NI SoftMotion runs on the NI EtherCAT master (CompactRIO), generates trajectory information (setpoints), and sends them to the EtherCAT AKD slave drive, which runs the position, velocity, and torque control loops. You can configure the AKD drive for position mode, velocity mode, or torque mode. In the standard use case, the AKD is configured for position mode when operating with NI SoftMotion over EtherCAT.

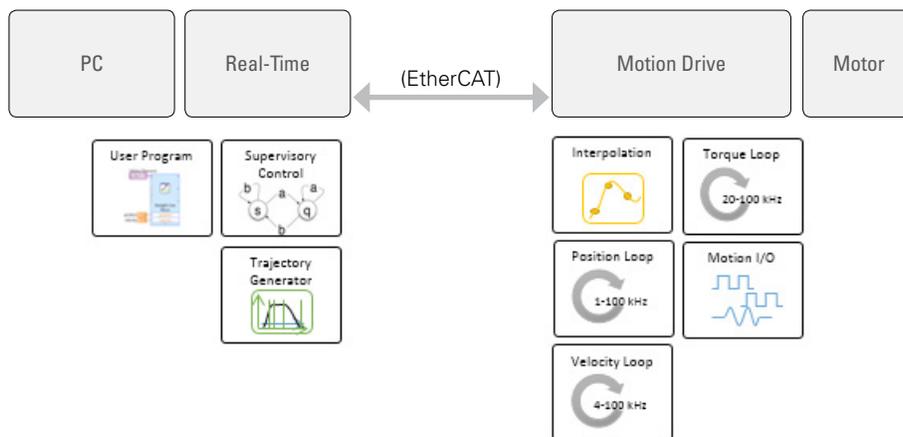


Figure 10.59. In the most common configuration, NI SoftMotion sends trajectory information (position commands) to the AKD, which then closes the control loops.

Benefits of building a motion system with CompactRIO and EtherCAT AKD drives:

- Simplest cabling and setup
- AKD autotuning feature
- No additional C Series modules required
- Fault code can be read from the AKD drive across the network

Disadvantages:

- No position compare functionality

- Encoder feedback to EtherCAT master is slower—same as the scan rate
- No control loop customization
- Position/velocity loop rates limited to 62.5 μ s

See the video [NI SoftMotion and AKD EtherCAT Brushless Servo Drives](#) for a demonstration of the AKD setup experience.

Read the case study [Master Machinery Builds Semiconductor Pick and Place Machine With CompactRIO and LabVIEW](#) for an industry example of this configuration.

NI SoftMotion and Drive Interface Modules (NI 951x C Series)

These motion modules provide servo or stepper drive interface signals for a single axis to communicate from NI SoftMotion running on a CompactRIO chassis to an external drive. In addition, they offer a full set of motion I/O including inputs for a home switch and limit switches, incremental encoder inputs for position feedback, and digital input and digital output lines. NI 951x drive interfaces include a processor to run the spline interpolation engine and the patented NI step generation algorithm or servo control loop.

You can choose from three C Series drive interfaces:



Figure 10.60. NI C Series Drive Interfaces

- NI 9512 single-axis stepper or position command (P-command) drive interface with incremental encoder feedback
- NI 9514 single-axis servo drive interface with incremental encoder feedback
- NI 9516 single-axis servo drive interface with dual incremental encoder feedback

NI 951x modules were designed to simplify wiring by providing the flexibility to hook up all the I/O for motion control to a single module. To further ease connectivity, the digital I/O is software configurable to connect to either sinking or sourcing devices. The modules offer the following:

- Analog or digital command signals to the drive
- Drive enable signal that is software configurable as sinking or sourcing (24 V)
- Digital input signals for home, limits, and general digital I/O; all software configurable as sinking or sourcing (24 V)

- Encoder inputs and 5 V supply; configurable for single ended or differential
- Digital input and digital output for high-speed position capture/compare functions (5 V)
- LEDs to provide quick debugging for encoder states, limit status, and axis faults

To further simplify wiring, NI offers several options for connecting NI 951x drive interface modules to external stepper drives or servo amplifiers including the following:

- NI 951x Cable and Terminal Block Bundle—Connects an NI 951x module with 37-pin spring or screw terminal blocks
- D-SUB and MDR solder cup connectors—Simplify custom cable creation
- D-SUB to pigtails cable and MDR to pigtails cable—Simplify custom cable creation
- Direct connect cables to the analog AKD servo drive



Figure 10.61. Directly connect cables between an NI 951x module and the analog AKD servo drive.

NI C Series drive interface modules are supported in NI Scan Mode and in FPGA mode. In NI Scan Mode, you do not need LabVIEW FPGA to configure and run the module. This configuration is the easiest to set up and use, and it provides all the necessary functionality for standard use cases. If you need to add custom triggering and timing with other C Series I/O or modify the control, then use the modules in FPGA mode.

By default in LabVIEW 2013 and earlier, you can use only NI 951x modules in the first four slots in a CompactRIO chassis because these are the only slots compiled with the high-speed interface (HSI). See the KnowledgeBase article [How Do I Use the NI 951x Motion Control Modules in Slots 5-8 \(Non High Speed Interface Slots\) of My CompactRIO Chassis?](#) for instructions on compiling the entire chassis with HSI support. Some older CompactRIO chassis such as the NI cRIO-9072, NI cRIO-9073, and NI cRIO-9074 have FPGAs that are not big enough to support HSI for all slots.

The NI 9512 drive interface module can control stepper drives that accept step and direction signals as well as control brushless servo drives that accept P-commands. Thus, you can use this module to interface to brushless servo axes as well as take advantage of the tuning features of those servo drives.

When a servo drive is connected to an NI 9514 or an NI 9516, it is configured in torque mode, which means position and velocity loop moved from the drive and executed in the NI 951x module. Torque loop always runs in the drive.

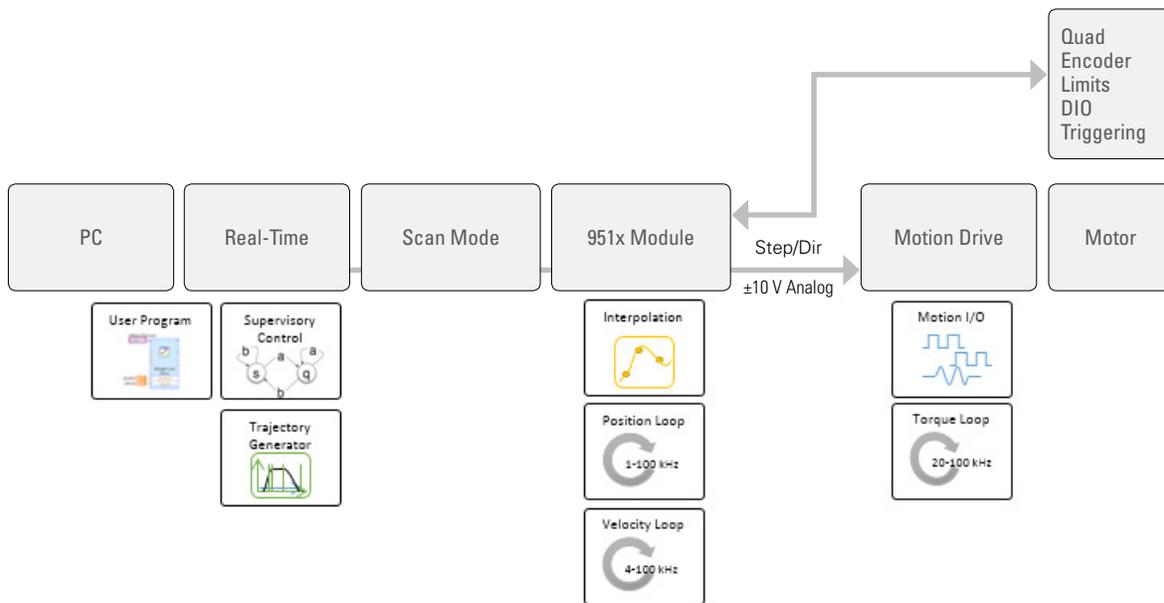


Figure 10.62. Using NI 951x modules in NI Scan Mode, spline interpolation, position, and velocity loops are executed on the module. Using NI 951x modules in FPGA mode, these loops are executed in LabVIEW FPGA in the backplane FPGA.

NI SoftMotion and Drive Modules (NI 950x)

NI 950x drive modules are motion drives implemented in the C Series form factor. They connect directly to motors to provide an embedded form factor.



Figure 10.63. All that you need to run this system is an external power supply. An NI 9502 and NI 9411 are connected to the AKM brushless servo motor (left). An NI 9503 is connected to the stepper motor (center), and an NI 9505 is connected to the brushed DC motor (right). You can control all three motor types and organize them in different combinations from the same CompactRIO chassis.

All drive firmware for these modules is implemented on the backplane FPGA using NI SoftMotion motor control IP blocks. These modules open up the ability to customize pretty much anything in an application because you implement all the control, including the torque loop, in LabVIEW FPGA. This means these drives require substantial FPGA resources and LabVIEW Real-Time/LabVIEW FPGA knowledge. Because all the control is open, the NI 950x drive modules provide a configurable off-the-shelf alternative to custom drive development. They save on cabling costs, so you can implement a configurable motion drive with a user-defined axis mix and I/O in a compact form factor.



Figure 10.64. NI 9503 (Stepper), NI 9502 (Brushless Servo), and NI 9505 (Brushed DC) Drive Modules

Because these drives are implemented in the CompactRIO C Series form factor, power dissipation is limited to 1.5 W. This spec is important because it says you have no limits on the other modules you can use in the CompactRIO chassis next to these drive modules. For instance, there are no thermal issues putting a high-accuracy 24-bit analog input module next to an NI 950x motion drive. However, this limits the power output of these drives, so they can control only small to mid-size motors. When selecting a platform, make sure that the drive modules can provide enough power with your selected motor to give you the torque you need in your application.



Figure 10.65. Connect motors directly to a CompactRIO system with NI 950x Drive Modules

- NI 9503 stepper, 3 A rms, 4.24 A peak current per phase
- NI 9502 brushless servo, 4 A continuous/8 A peak current
- NI 9505 brushed DC, 5 A continuous/12 A peak current

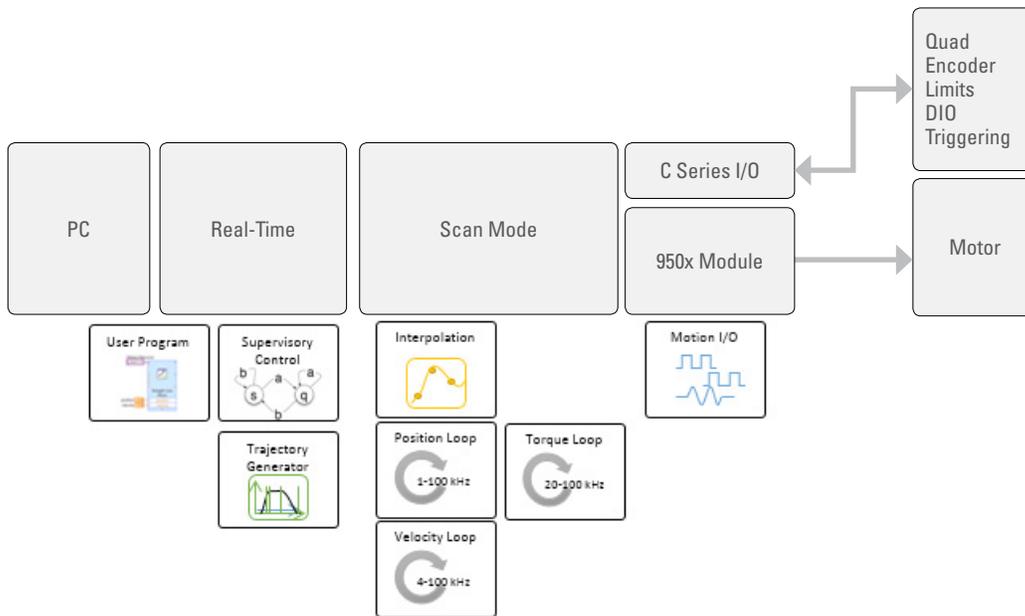


Figure 10.66. In the NI 950x drive module architecture, all control is implemented in NI SoftMotion on LabVIEW FPGA.

When using NI 950x drive modules, components that are typically executed on external drive firmware are pulled in from the drive and implemented in LabVIEW FPGA on the CompactRIO backplane instead. The NI SoftMotion shipping examples provide working starting points for NI 950x modules, and can be used with little modification to the motion IP and control loops for many motion applications.

Example Configurations

The following are some common/interesting ways to configure a CompactRIO motion system. They illustrate the flexibility and possibilities in assembling a configurable off-the-shelf system to meet a variety of application needs.

Ethernet RIO With C Series Motion Modules

Ethernet RIO expansion chassis are the only NI reconfigurable I/O (RIO) targets that do not require any additional software other than the LabVIEW development system and necessary driver. NI Ethernet RIO technology offers a simplified software experience directly out of the box. The LabVIEW Real-Time Module is not required to access the NI C Series I/O from a Windows-based system. The LabVIEW FPGA Module is not required to access the C Series I/O using NI Scan Mode.

Because NI 951x drive interface modules are supported in NI Scan Mode, you can add them to an Ethernet RIO chassis to create a distributable motion controller with stepper or servo motion combined with other C Series modules in the remaining slots. The system requires no LabVIEW Real-Time or LabVIEW FPGA programming; all you need is LabVIEW and the LabVIEW NI SoftMotion Module installed on your development PC.



Figure 10.67. When using Ethernet RIO chassis and NI 951x drive interface modules, no software is needed beyond LabVIEW with the LabVIEW NI SoftMotion Module installed on the development machine.

When you run a high-level VI containing an application built with the NI SoftMotion API on the development PC, NI SoftMotion uses web services to deploy the necessary code to the Ethernet RIO chassis, where it runs all the necessary supervisory and trajectory tasks locally to maintain the determinism and performance needed for motion control applications.



Figure 10.68. This 4-axis stepper motion controller was built using the NI 9076 Ethernet RIO chassis and four NI 9512 drive interface modules.

Axes within the chassis are coordinated. Axes across multiple Ethernet RIO chassis on the same Ethernet network are not coordinated.

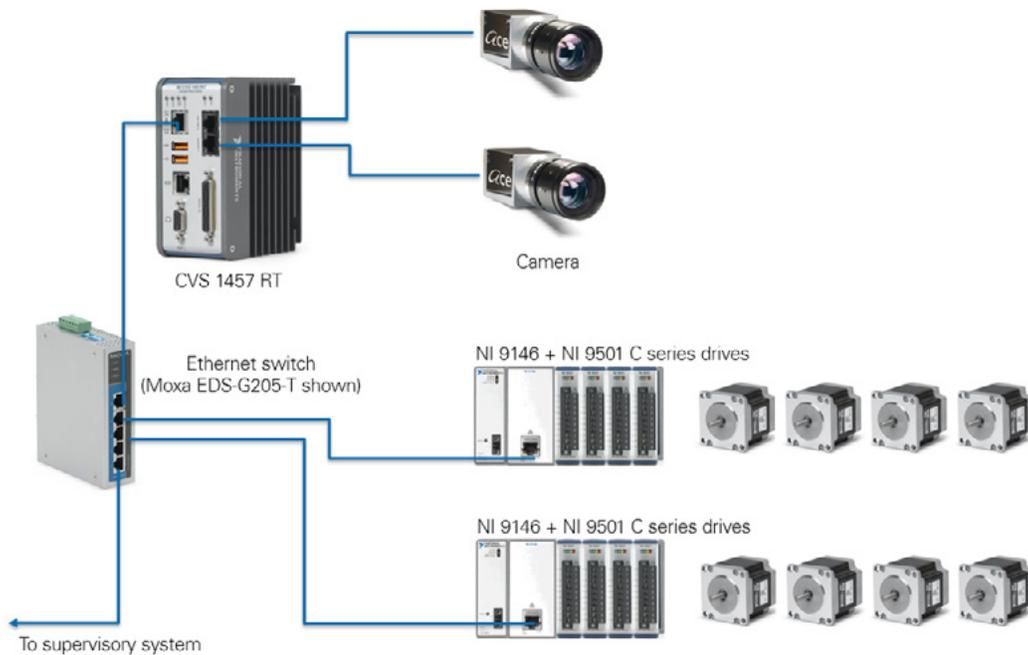


Figure 10.69. You can use the Ethernet RIO chassis to add a modular motion control unit to any application over an Ethernet network. This example shows an integrated vision and motion system for an inspection and positioning application.

In Figure 10.69, the NI 950x drive modules are used in the Ethernet chassis to create an embedded motion drive. This configuration requires the correct LabVIEW FPGA code to be running on the backplane to implement the drive firmware for these modules. However, once you develop the bitfile, you do not need LabVIEW FPGA to run or interact with the Ethernet RIO motion controller except to change the functionality of the drive firmware implemented in LabVIEW FPGA.

EtherCAT-Based System

The EtherCAT-based system can include EtherCAT AKD drives and any motion C Series module in the NI 9144 EtherCAT slave chassis. Any CompactRIO controller with two Ethernet ports can act as the EtherCAT master. Because of the determinism of EtherCAT, you can synchronize motion axes across multiple chassis at the scan rate of the EtherCAT master. EtherCAT provides for high-axis-count and mixed-axis systems. The axis count is limited only by the processing power of the EtherCAT master and the desired scan rate. It's fairly common to have more than a dozen axes in an EtherCAT system with an appropriate CompactRIO controller (NI cRIO-9024/9025/9068/9081/9082) as the EtherCAT master.

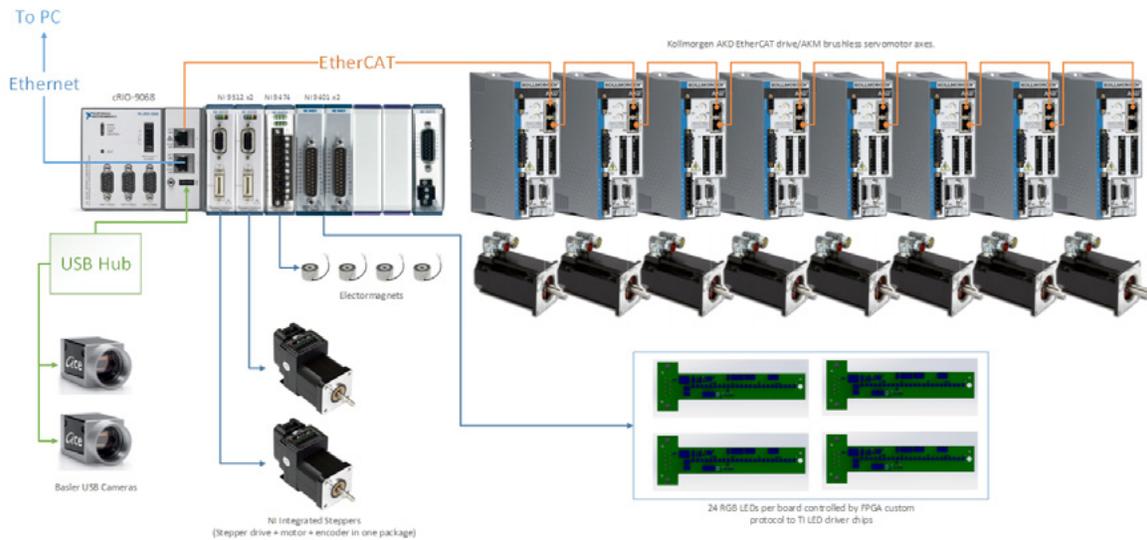


Figure 10.70. System Setup Used on the NI 3D Gears Demonstration System (The NI cRIO-9068 is controlling eight AKD servo drives over EtherCAT, two USB cameras, and a variety of I/O from the C Series slots including two stepper drives, four electromagnets, and a custom digital protocol controlling 96 RGB LEDs with a custom FPGA-based digital protocol).

In the Figure 10.70 setup, if there were insufficient slots in the CompactRIO chassis and more motion axes or I/O channels were required, you can add an NI 9144 chassis with the right mix of NI 951x drive interfaces, NI 950x drives, and C Series I/O modules.

Mixed-Axis Systems

Figure 10.70 features mixed-axis systems. With NI SoftMotion, you can synchronize axes of different types as long as they all use the same motion controller. For example, using an NI 9024 EtherCAT master, you could perform synchronized moves using NI 9512 stepper axes, NI 9514 servo axes, EtherCAT AKD axes, NI 9502 servo axes, or any other axis type. Axes in NI SoftMotion, though often bound to a particular hardware type, are not associated with hardware when using the high-level NI SoftMotion API. A straight line move is the same on an AKD axis as it is on an NI 9512 axis. It is only when the straight line move command gets to the communication interface that it is transformed into the signals that are expected by the hardware.

NI Drives and Motors

NI partners with Kollmorgen to provide a full suite of brushless servo drives and motors, gearheads, and linear actuators. Interfacing the AKD drive to CompactRIO through NI SoftMotion is discussed in an earlier section. All the motors shown in Figure 10.71 are compatible with the AKD drive and can be used in a CompactRIO system.



Figure 10.71. Kollmorgen Servo Product Offering Available Through NI

NI offers a full suite of stepper products including integrated motors, DC and AC stepper drives, and NEMA 8–34 frame-sized stepper motors with integrated encoder options.



Figure 10.72. NI Stepper Offering

These industry-leading drives and motors are fully compatible with NI controllers and drive interfaces.

Determining System Requirements and Components

- Controllers (number of axes, processing power, I/O count and synchronization, communication strategy)
- Motor and drive (comparison of motor types, power ranges, typical application uses)
- Feedback devices (I/O synchronization)

Start by selecting the right mechanical components and motors for your system. One of the most common applications involves moving an object from one position to another. A typical way to change the rotary motion of a rotary motor to a useful linear motion is by connecting the motor to a stage and using a leadscrew mechanism to move the payload. Stages are mechanical devices that provide linear or rotary motion useful in positioning and moving objects. They come in a variety of types and sizes, so you can use them in many different applications. To find the correct stage for your application, you need to be familiar with some of the common terminology used when describing stages. Some of the key items to consider when selecting a stage include the following:

- **Transmission gear ratio**—Determines the linear travel distance of the stage per rotary revolution of the motor.
- **Accuracy**—How closely the length of a commanded move compares to a standard length.
- **Resolution**—The smallest length of travel that a system is capable of implementing can be as small as a few nanometers.

- **Travel distance**—The maximum length the system is capable of moving in one direction.
- **Repeatability**—The repeated motion to a commanded position under the same conditions. Often this is specified in unidirectional repeatability, which is the ability to return to the same point from one direction, and bidirectional repeatability, which specifies the ability to return from either direction.
- **Maximum load**—The maximum weight the stage is physically designed to carry, given accuracy and repeatability.

Stage Selection

You can choose from a variety of stages to meet your application needs. You can narrow down these stages to two main types—linear and rotary. Linear stages move in a straight line and are often stacked on each other to provide travel in multiple directions. A three-axis system with an x, y, and z component is a common setup used to position an object anywhere in a 3D space. A rotary stage is a stage that rotates about an axis (usually at the center). Linear stages are used to position an object in space, but rotary stages are used to orient objects in space and adjust the roll, pitch, and yaw of an object. Many applications, such as high-precision alignment, require both position and orientation to perform accurate alignment. The resolution for a rotary stage is often measured in degrees or arc minutes (1 degree equals 60 arc minutes). Special types of stages include the goniometer, which looks like a linear stage that moves in an arc rather than a straight line, or a hexapod, which is a parallel mechanism that gives you movement in six axes to control—x, y, z, roll, pitch, and yaw. With a hexapod, you can define a virtual point in space about which the stage can rotate. While that is a benefit, the disadvantage is that hexapods are parallel and the kinematics involved are much more complex than those for simple stacked stages.

Backlash

Another stage attribute to consider when choosing a stage for your precision motion system is backlash. Backlash is the lag created when one gear in a system changes direction and moves a small distance before making contact with a companion gear. It can lead to significant inaccuracies especially in systems using many gears in the drive train. When moving on a nanometer scale, even a small amount of backlash can introduce major inaccuracy in the system. Effective mechanical design minimizes backlash but may not be able to eliminate it completely. You can compensate for this problem by implementing dual-loop feedback in software. The NI 9516 C Series servo drive interface module supports dual-encoder feedback and accepts feedback from two different sources for a single axis. To understand the circumstances for which you need to use this feature, consider a stage. If you monitor the stage position directly (as opposed to the position of the motor driving the stage), you can tell if a move you have commanded has reached the target location. However, because the motion controller is providing input signals to the motor and not to the stage, which is the primary source of feedback, the difference in the expected output relative to the input can cause the system to become unstable. To make the fine adjustments necessary to help the system stay stable, you can monitor the feedback directly from the encoder on the motor as a secondary feedback source. Using this method, you can monitor the real position of your stage and account for the inaccuracies in the drive train.

Motor Selection

A motor provides the stage movement. Some high-precision stages and mechanical components feature a built-in motor to minimize backlash and increase repeatability, but most stages and components use a mechanical coupling to connect to a standard rotary motor. To make connectivity easier, the National Electrical Manufacturers Association (NEMA) has standardized motor dimensions. For fractional horsepower motors, the frame sizes have a two-digit designation such as NEMA 17 or NEMA 23. For these motors, the frame size designates a particular shaft height, shaft diameter, and mounting hole pattern. Frame designations are not based on torque and speed, so you have a range of torque and speed combinations in one frame size.

The proper motor must be paired with the mechanical system to provide the performance required. You can choose from the following four main motor technologies:

- **Stepper motor**—A stepper motor is less expensive than a servo motor of a similar size and is typically easier to use. These devices are called stepper motors because they move in discrete steps. Controlling a stepper motor requires a stepper drive, which receives step and direction signals from the controller. You can run stepper motors, which are effective for low-cost applications, in an open-loop configuration (no encoder feedback). In general, a stepper motor has high torque at low speeds and good holding torque but low torque at high speeds and a lower maximum speed. Movement at low speeds can also be choppy, but most stepper drives feature a microstepping capability to minimize this problem.
- **Brushed servo motor**—This is a simple motor on which electrical contacts pass power to the armature through a mechanical rotary switch called a commutator. These motors provide a 2-wire connection and are controlled by varying the current to the motor, often through PWM control. The motor drive converts a control signal, normally a ± 10 V analog command signal, to a current output to the motor and may require tuning. These motors are fairly easy to control and provide good torque across their ranges. However, they do require periodic brush maintenance, and, compared to brushless servo motors, they have a limited speed range and offer less efficiency due to the mechanical limitations of the brushes.
- **Brushless servo motor**—These motors use a permanent magnet rotor, three phases of driving coils, and Hall effect sensors to determine the position of the rotor. A specialized drive converts the ± 10 V analog signal from the controller into three-phase power for the motor. The drive contains intelligence to perform electronic commutation and requires tuning. These efficient motors deliver high torque and speed and require less maintenance. However, they are more complex to set up and tune, and the motor and drive are more expensive.

Stage Drive Technology	Speed	Maximum Load	Travel Distance	Repeatability	Relative Complexity	Relative Cost
Stepper	Medium/Low	Medium/Low	High	Medium/Low	Low	Low
Brushed Servo	High	High	High	Medium	Medium	Low
Brushless Servo	Very High	High	High	High	High/Medium	High

Table 10.4. Comparison of Motor Technology Options

Because the motor and drive are so closely coupled, you should use a motor and drive combination from one vendor. Though this is not a requirement, it makes tuning and selection easier.

CHAPTER 11

Deploying and Replicating Systems

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 on a target.

Deploy to Volatile Memory

If you deploy the application to the volatile memory on a target, the application does not remain on the target after you cycle power. This is useful when iteratively developing your application and testing your code.

Deploy to Nonvolatile Memory

If you deploy the application to the nonvolatile memory on a target, the application remains after you cycle the power on the target. You can set applications stored on nonvolatile memory to start up automatically when the real-time target boots. This is useful when you have finished development and want to create a stand-alone embedded system.

Deploying Applications to CompactRIO

Deploy a LabVIEW VI to Volatile Memory

When you deploy an application to the volatile memory of a CompactRIO controller, LabVIEW collects all of the necessary files and downloads them over Ethernet to the CompactRIO controller. To deploy an application to volatile memory you need to

- Target the CompactRIO controller in LabVIEW
- Open a VI under the controller
- Click 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. Use the ability to deploy to volatile memory to quickly iterate through application development.

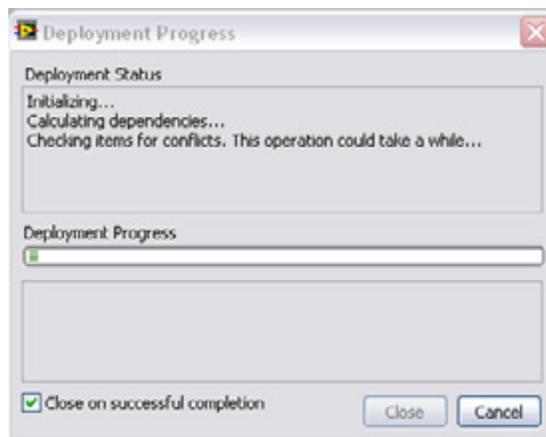


Figure 11.1. LabVIEW Deploying an Application to the Nonvolatile Memory of the Controller

Deploy a LabVIEW VI to Nonvolatile Memory

Once you have finished developing and debugging your application, you likely want to deploy your code to the nonvolatile memory on the controller so that it persists through power cycles. To deploy an application to the nonvolatile memory, you first need to build the VI into a real-time executable (RTEXE). RTEXEs can be configured to run when the real-time device boots, allowing for truly headless embedded operation.

Building an Executable From a VI

With the LabVIEW project, you can build an executable real-time application from a VI by creating a build specification under the real-time target in the LabVIEW Project Explorer. When you right-click 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.

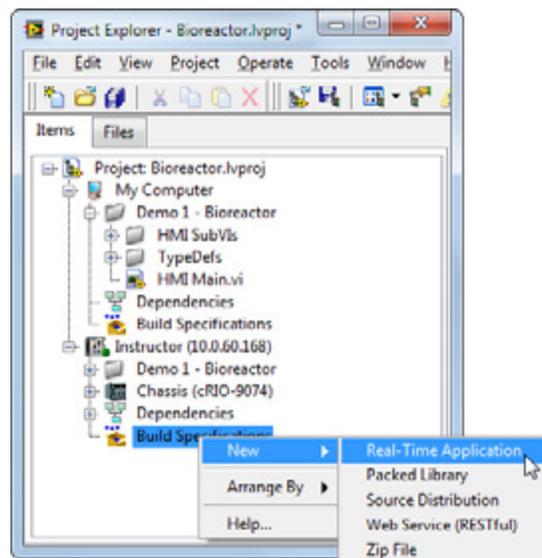


Figure 11.2. Create a new real-time application build specification.

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 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.

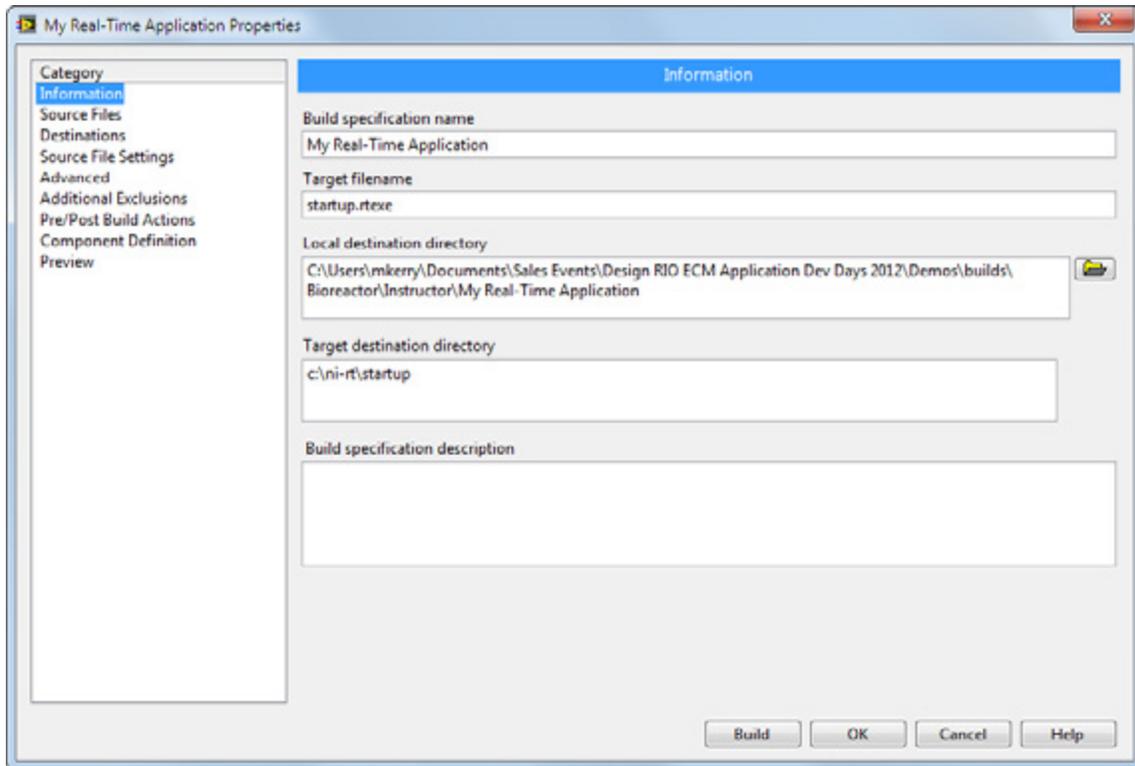


Figure 11.3. The Information Category in the Real-Time Application Properties

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 as Startup VIs or Always Included unless they are called dynamically in your application.

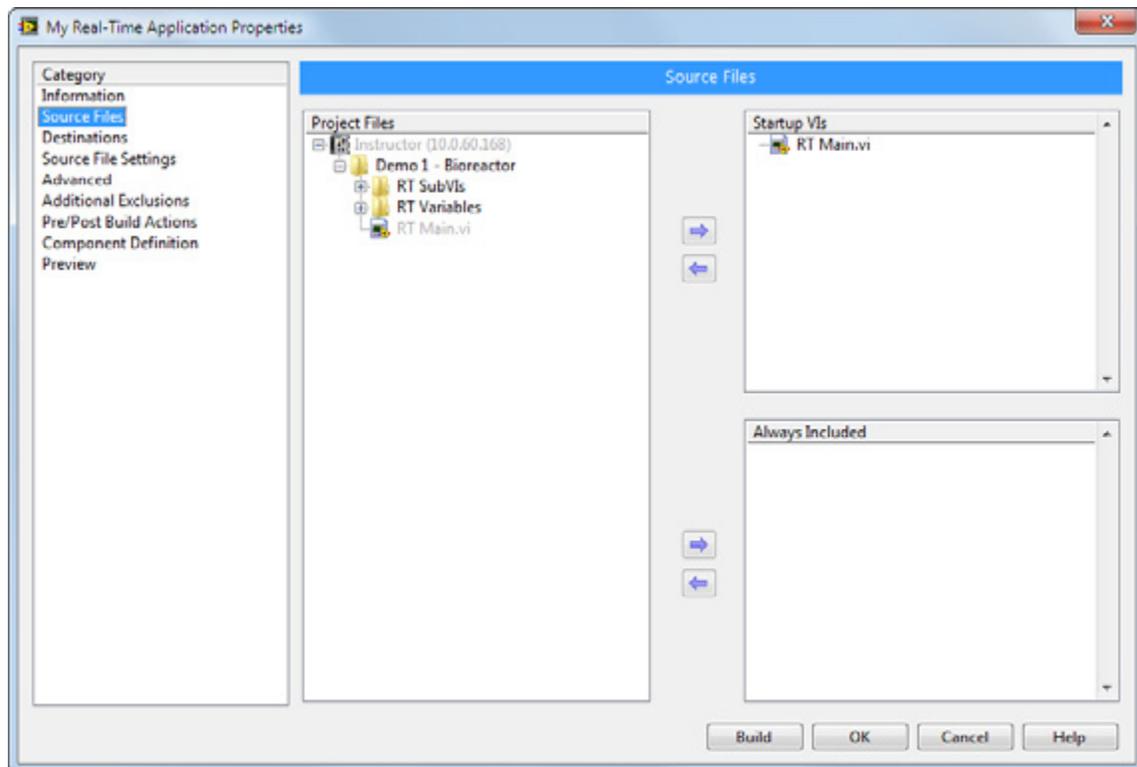


Figure 11.4. Source Files Category in the Real-Time Application Properties (In this example, the *cRIOEmbeddedDataLogger (Host).vi* was selected to be a Startup VI.)

After all of the 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.

Setting 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.

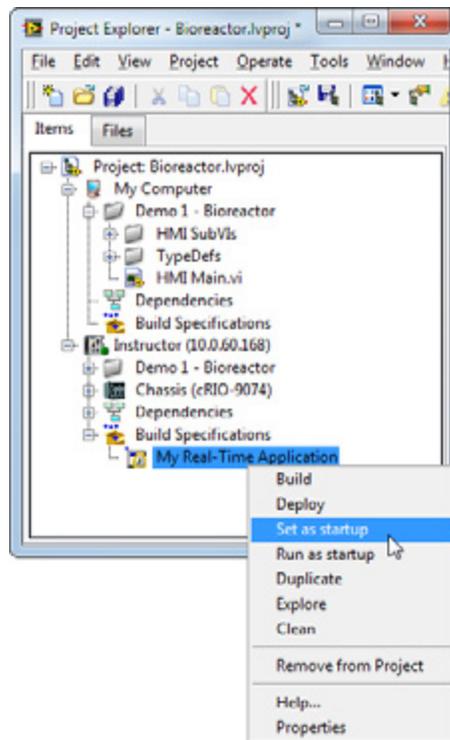


Figure 11.5. Configuring a Build Specification to Run When an Application Boots

Deploy Executable Real-Time Applications 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 controller 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. LabVIEW then copies the executable onto the controller's nonvolatile memory 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.

If you used the default settings, the real-time executable (RTEXE) is located in the `NI-RT\ni-rt\startup` folder on the target with the name supplied in the target filename box from the Information category and the extension `.rtexe`.

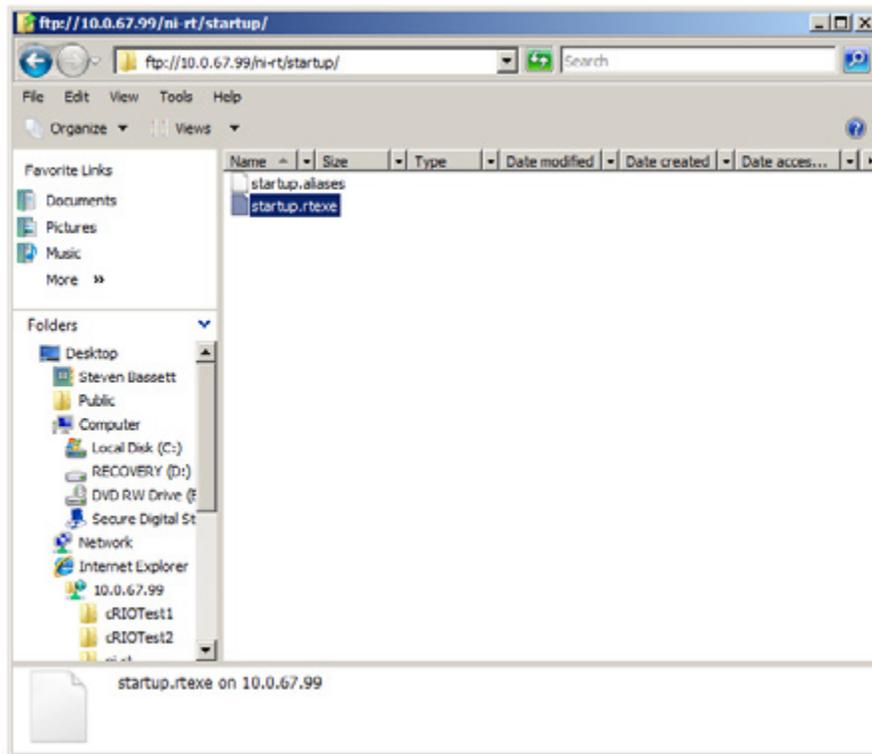


Figure 11.6. The Default Location of the startup.rtexe on a CompactRIO Controller

Deploying LabVIEW FPGA Applications

Once the development phase of the FPGA application is complete, you need to deploy the generated bitfile, also referred to as a personality, to the system. You have two main ways to embed the FPGA application onto a target with a programmable onboard FPGA. In the vast majority of applications, an Open FPGA VI Reference is used in the host application (real-time or Windows) to communicate with the FPGA application. The Open FPGA VI Reference stores the FPGA application, and is capable of deploying to the FPGA at run time. In addition to the Open FPGA VI Reference, the FPGA bitfile (representing the FPGA application) can be downloaded to nonvolatile Flash memory alongside the FPGA, and can be configured to deploy and run on the FPGA when the target is powered. The two FPGA deployment methodologies are illustrated in Figure 11.7.

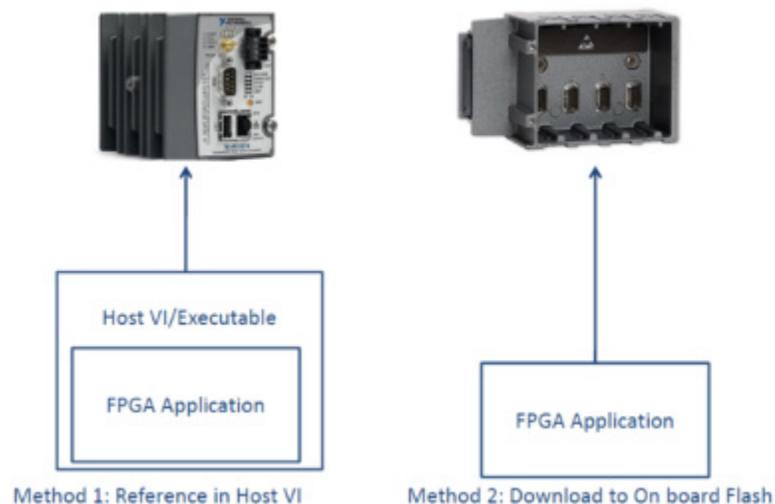


Figure 11.7. Two Stand-Along Deployment Options for LabVIEW FPGA

Method 1: Reference in Host VI

One method for deploying an FPGA personality is to embed it in the host application as shown in Figure 11.8. This can be achieved by using the Open FPGA VI Reference function in the host application. When the host application is then compiled into an executable, the FPGA application is embedded in the Open FPGA VI Reference function call. When the host application is embedded on the target and runs, the Open FPGA VI Reference function is called and then it downloads and runs the FPGA bitfile and outputs a reference to the bitfile for subsequent use in the host application. If the referenced FPGA bitfile is already running on the FPGA when the Open FPGA VI Reference is called, the function does not download the FPGA bitfile and instead only generates a reference to the bitfile for downstream use in the host application.

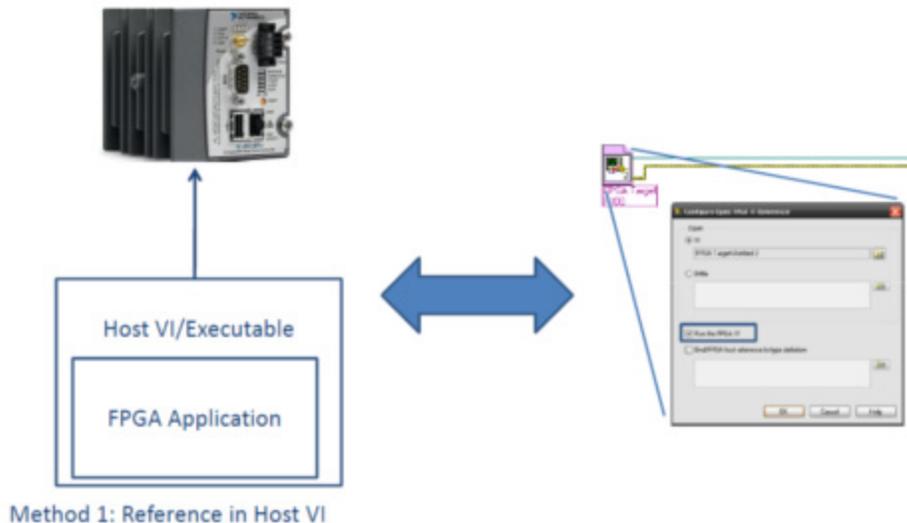


Figure 11.8. If you use an Open FPGA VI Reference in a host VI, then you embed the FPGA's bitfile in the host executable.

One caveat to relying solely on the Open FPGA VI Reference function to download and run the FPGA application is that the host application must first initialize before the FPGA application can be loaded. This causes a delay from device power up to the configuration and operation of the FPGA. On device power, the input and output lines on the FPGA are in an indeterminate state until the host application loads and executes the Open FPGA VI Reference function. The delay in loading and executing the host application can easily be on the order of a minute, posing a potential safety and reliability risk as the FPGA I/O states remain uncontrolled. Therefore, it's recommended that the FPGA bitfile implements safe states and be stored on the onboard FPGA flash memory.

Method 2: Storing the Application in Nonvolatile Flash Memory on the FPGA Target

In addition to relying solely on the Open FPGA VI Reference, you can download the FPGA bitfile to flash memory on the target device using the RIO Device Setup that is included with the NI-RIO driver as shown in Figure 11.9. You can then configure the bitfile downloaded to flash memory to load and run the personality onto the FPGA as well as load independently of the Open FPGA VI Reference function. To learn more about this download process, reference the [KnowledgeBase article How Do I Download a Bitfile to My Target Without LabVIEW FPGA?](#) You should download an FPGA bitfile implementing safe states to the flash memory and set it to run on device power up, or reboot, so that all of the input and output lines are immediately driven to a known state.

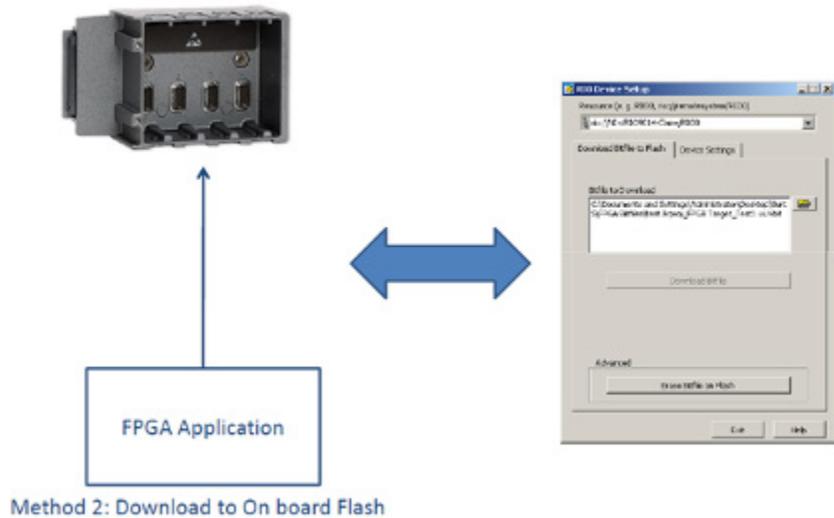


Figure 11.9. You can download the FPGA bitfile to the FPGA's onboard flash memory using the RIO Device Setup.

If you have downloaded an FPGA application to flash memory and set it to run on device power or device reboot, you can still use the Open FPGA VI Reference to communicate between the FPGA and host applications. The Open FPGA VI Reference simply generates a reference if the FPGA bitfile running on the device matches the FPGA bitfile stored within the Open FPGA VI Reference node. If, however, the bitfile stored in the Open FPGA VI Reference function differs from the bitfile running on the FPGA, note that the Run the FPGA VI replaces the bitfile running on the FPGA with the bitfile contained in the Open FPGA VI Reference function. For more information, refer to the [Open FPGA VI Reference function documentation](#) in the LabVIEW Help.

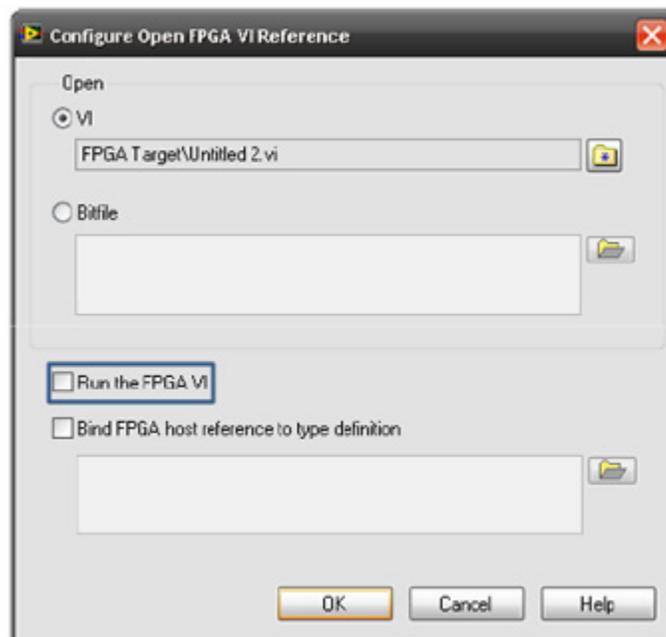


Figure 11.10. The Run the FPGA VI option in the Open FPGA VI Reference configuration window toggles whether the Open FPGA VI Reference function overwrites the bitfile loaded to the FPGA in cases where the bitfile running on the FPGA doesn't match the bitfile embedded in the Open FPGA VI Reference function.

It is generally recommended that you embed the same FPGA personality referenced in the host application into flash memory. This is to enable greater safety and reliability by having the I/O state immediately controlled on device

power or reboot. For more information on managing FPGA deployments, see the NI Developer Zone document [Managing FPGA Deployments](#).

Beginning in LabVIEW 2013, the System Configuration API, Measurement & Automation Explorer (MAX), and the Web-Based Configuration and Monitoring interface all provide support for managing FPGA bitfiles for targets that run NI Linux Real-Time, such as the NI cRIO-9068.

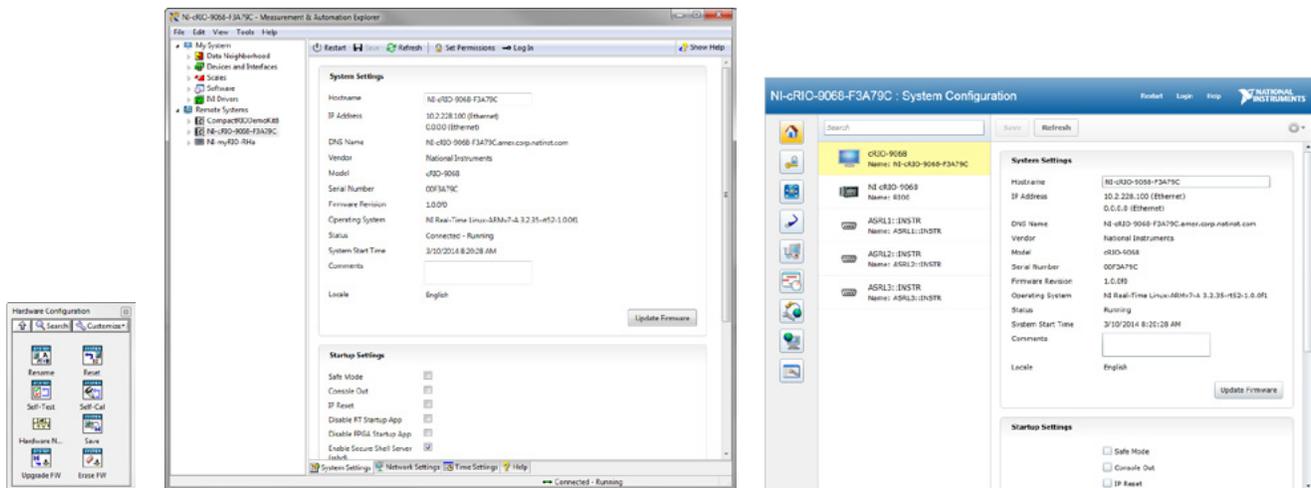


Figure 11.11. In LabVIEW 2013, FPGA flash deployment on NI Linux Real-Time targets is supported through the System Configuration API, MAX, and the Web-Based Configuration and Monitoring interface.

Deploying Applications That Use Network-Published Shared Variables

The term *network shared variable* refers to a software item on the network that can communicate between programs, applications, remote computers, and hardware. Find more information on network shared variables in [Chapter 4: Best Practices for Network Communication](#).

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 Shared Variable Engine to create new data items on the network.

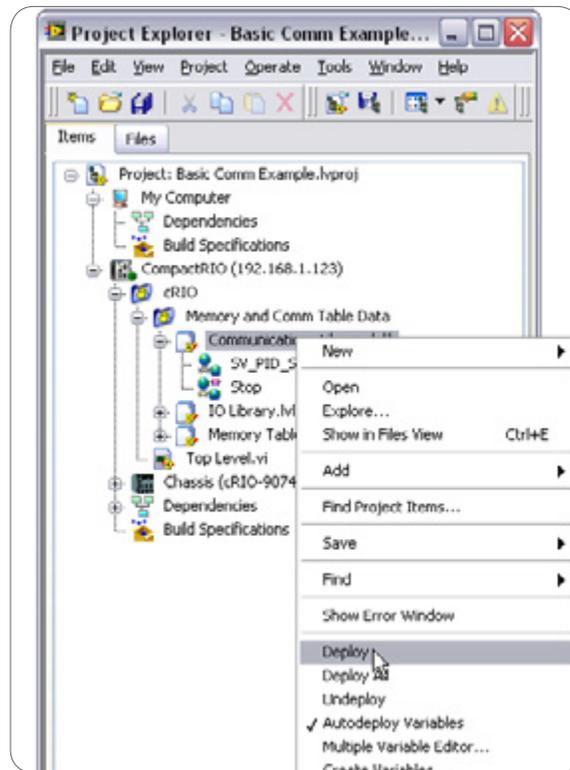


Figure 11.12. Deploy libraries to real-time targets by selecting Deploy from the right-click menu.

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 open the programming palette, go to **Programming»Application Control**, and place the **Application Invoke Node** on the block diagram.
- Using the hand tool, click on **Method** and select **Library»Deploy Library**.



Figure 11.13. 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 you deploy a library to a Shared Variable Engine, those settings persist until you manually undeploy them. To undeploy a library

3. Launch the NI Distributed System Manager (from **LabVIEW»Tools** or from the Start Menu).
4. Add the real-time system to My Systems (**Actions»Add System to My Systems**).
5. 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 11.14. 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 need to edit only 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.

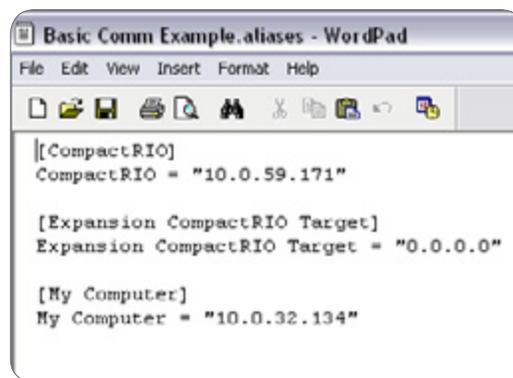


Figure 11.15. The alias file is a human-readable file that lists the target name and IP address.

If you are deploying systems with dynamic IP addresses using DHCP, you can use the domain name system (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.

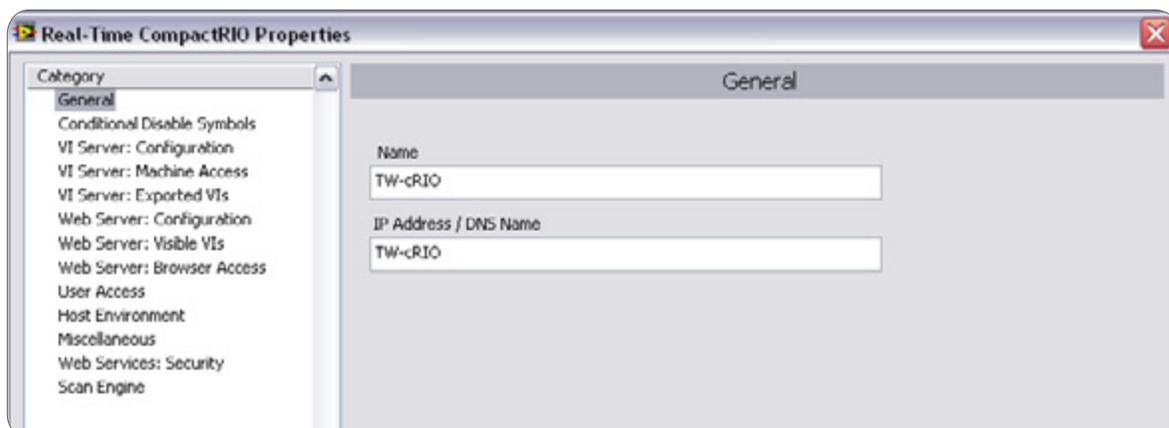


Figure 11.16. For systems using DHCP, you can enter the DNS name instead of the IP address.

One good approach if you need scalability is to develop using a generic target machine (you can develop for remote machines that do not exist) with a name indicating its purpose in the application. Then as part of the installer, you can run an executable that 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

NI also provides several sets of commonly used drivers called recommended software sets. You can install recommended software sets on CompactRIO controllers from MAX.

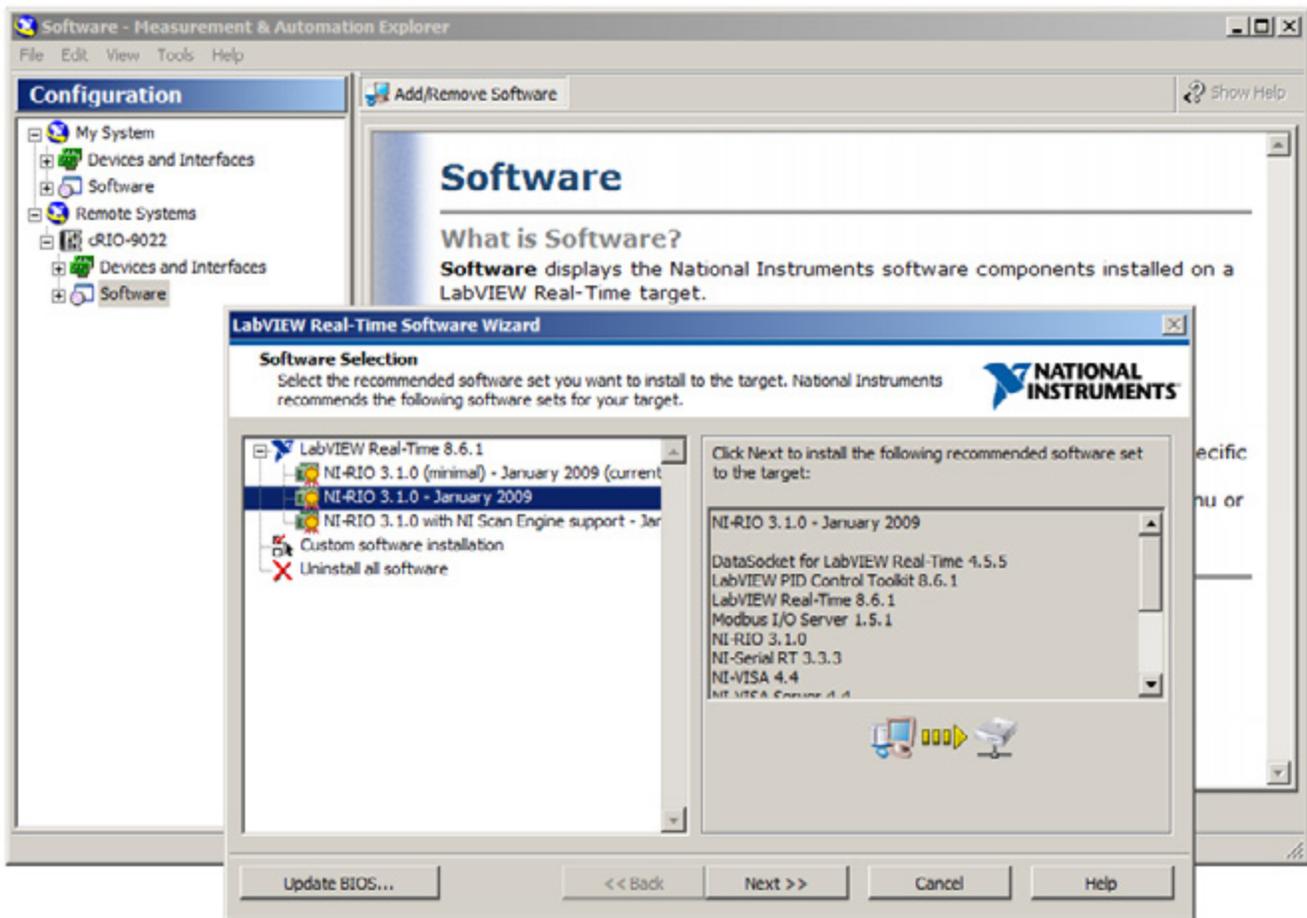


Figure 11.17. Recommended Software Sets Being Installed on a CompactRIO Controller

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.

Deployment Beyond the LabVIEW Project

Beyond the LabVIEW project, you can choose from two main techniques for deploying to embedded targets. These techniques, imaging and application components respectively, are far more scalable than LabVIEW Project based deployment. Imaging relies on copying the hard drive of one embedded system and propagating the copy to other systems, making it a useful technique when initially replicating embedded systems. Application components alternatively encapsulate just the real-time executable (RTEXE), and provide an effective means to perform small system upgrades to deployed systems. In addition to application components and images, deployed embedded systems often rely on local system critical files (example: application configurations). An efficient configuration management solution for an embedded system will provide for tracking and updating system critical files.

Imaging

An image in the context of NI embedded hardware and software is a single file which can contain, at a maximum, a copy of all of the files on an embedded target's disk. Given that the RTEXE, system drivers, network settings, shared variables, scan engine settings, web services, and any application configurations or plug-ins are all stored as files on disk for an embedded system, Imaging is an effective way to create a backup of a system and to replicate a given embedded system.

In order to leverage imaging, a fully configured embedded system is required. Often, this means relying on LabVIEW Project based deployment to properly deploy to the first embedded system, after which imaging can be used to setup additional systems. Given that Imaging relies on copying all the files on a properly configured embedded target, there are a few key caveats to be aware of.

The most significant caveat is that imaging can only be used to deploy/replicate to other embedded targets that are of the same model. For example, if a cRIO 9074 is used to get an image, when applying the image to setup subsequent systems, only cRIO 9074s can be used. The image from a cRIO 9074 cannot be applied to a different model cRIO, such as an NI 9024. Beyond being restricted to the same model, imaging doesn't copy the FPGA flash settings between controllers as it only copies the disk that belongs to the real-time operating system on the embedded device. When applying an image, users need to take precautions to configure the imaging settings properly to preserve existing configuration or log files on the embedded target, as there is a risk overwriting existing files on the target.

Imaging is recommended when deploying to multiple targets, and is especially recommended when initially setting up large numbers of embedded targets for field deployment. Imaging is also recommended for creating restore points or backups for embedded applications. To begin using imaging to manage large numbers of embedded deployments, please refer to the following Systems Engineering utility: Replication and Deployment (RAD) Utility. The RAD utility makes use of the System Configuration API which can be used to create and deploy images. It's recommended to use the source code provided for the RAD as a starting point when attempting to create a custom imaging utility as opposed to starting from a blank VI.

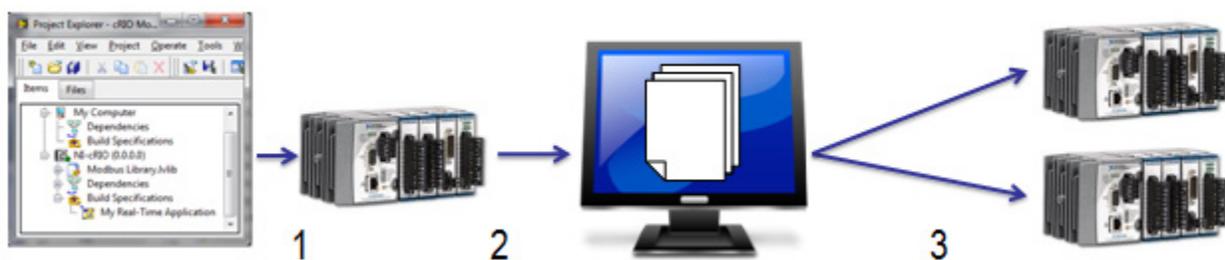


Figure 11.18. By relying on the System Configuration API and the imaging tools it provides, you can deploy images to multiple real-time targets.

Starting in LabVIEW 2013, system deployment with system images has been improved greatly on NI Linux Real-Time targets such as the cRIO-9068. These targets have the ability to run the 'Set Image' function from the System Configuration API directly, allowing for intelligent self-updating. Targets can be programmed to pull system images down from a network or locally attached storage and allow you to more securely manage larger scale deployments.

Application Components

Application components, unlike images, only encapsulate the real-time application (RTEXE) and associated driver dependencies. To better understand application components, it's useful to first cover driver components. When going through the Add/Remove Software dialog for a real-time target such as a CompactRIO, the list of options that appear both under the Recommended Software Set and the Custom Software Installation are known as components. Example components for the CompactRIO are System State Publisher, Network Variable Engine, and NI-RIO I/O Scan. Application components are similar to the driver components, but encapsulate the RTEXE, and can have driver components listed as dependencies.

Since application components only encapsulate the RTEXE and do not contain other configuration information about the embedded target, they are not a reliable way to define a backup or restore point for an embedded target. However, this same characteristic allows for them to install across any real-time target, much like driver components. Key benefits of application components are that they can be installed to other targets, and deploy much more quickly than images. For example, a real-time application developed on a cRIO 9074 can be built into an application component and deployed to an NI 9024. Application components can also be built directly from the LabVIEW Project, and unlike images, do not require an initial deployment and retrieval. To learn more about configuring and using application components, please refer to: [Using Application Components to Deploy LabVIEW Real-Time Applications](#).

The Replication and Deployment Utility (RAD)

The RAD utility is an imaging-based, turnkey application for deploying and replicating LabVIEW Real-Time applications. It is one of the easiest and most effective ways to distribute and manage updates to embedded systems. OEMs may use this type of utility as part of their factory installation processes when assembling their products, or NI can provide a modified version of the RAD executable to end clients as a tool for performing local system updates.

You can download this utility from the NI Developer Zone document at: [Replication and Deployment \(RAD\) Utility](#). After installation, you can start the RAD directly from the Tools menu in the LabVIEW environment.

When replicating a given system, an image is retrieved from the given real-time target and applied to other systems. The application image captures the contents of the hard drive of the real-time target that is fully configured with the real-time application. In addition, the RAD utility also stores information about the FPGA bitfile (if any), which is set to deploy to the FPGA flash memory. Note again that since the RAD utility relies on imaging, it can only replicate images to identical controllers and backplanes. **If you have an image from a specific controller, you can deploy that image only to controllers with the same model number.**

The main UI of the RAD utility shows two columns, listing Real-Time Targets and local Application Images.

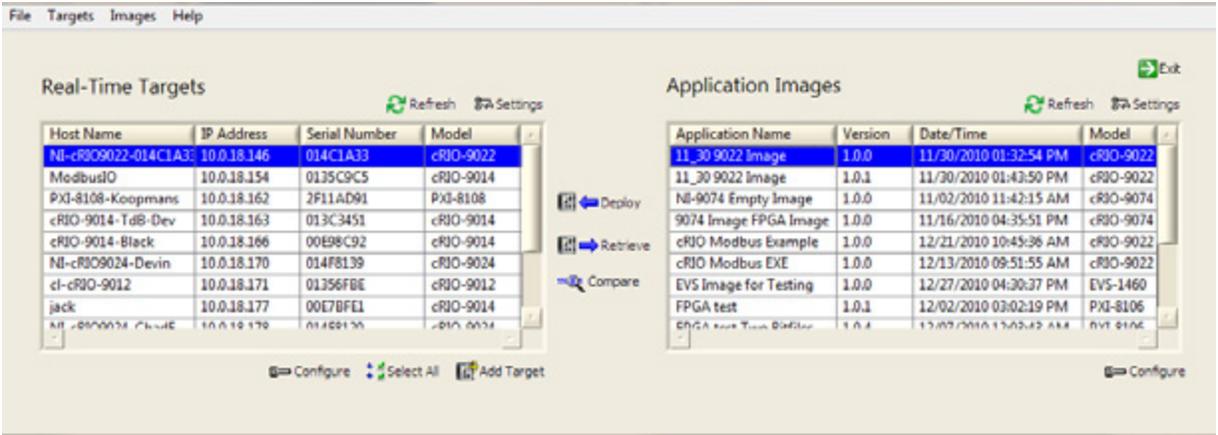


Figure 11.19. With the NI Replication and Deployment Utility, you can quickly and easily deploy images to multiple real-time targets.

The Real-Time Targets table shows all of the targets on the local subnet as well as any network targets manually added to the list. You can use these targets for both image retrieval and image deployment. The Application Images table shows all of the images that are stored on the local hard drive that can be deployed to target systems.

Retrieving Application Images

To copy the application image from a real-time target to the local hard drive, select the appropriate target in the table and click the Retrieve button in the center of the UI. You can retrieve an application image from a target in three ways. The application could either be a brand new application or a new version of a previous image either currently on the controller or previously saved to disk. If the current application is a new version of an older application, it is best to inherit the old application properties. If you are creating an image of the application for the first time, select New Application Image.

After making this selection, you are presented with the dialog box in Figure 11.20 to specify the local file for the application image and specify some additional information that is stored with the application image. If this is not a new image, properties from the old version of the image are automatically populated.



Figure 11.20. Configuring Your Application Image Properties

In addition to retrieving and deploying an image on the real-time hard drive, the utility can deploy bitfiles to FPGA flash memory. Saving a bitfile in flash memory has some advantages over deploying the bitfile from the RTEXE. For example, the host application is required to initialize before the FPGA application is loaded and, as a result, there is a delay from device power up to the configuration of the FPGA. In addition, on power up, the state of the input and output lines of your target is unknown since the FPGA has not yet been configured. Therefore, if the FPGA is completely independent of the host application, its personality should be stored in the onboard FPGA flash memory.

Using the RTAD utility, you can now save bitfiles with an image during retrieval and then later deploy them to flash memory when you deploy the image. Click Configure Bitfile(s) for FPGA Flash Deployment to edit your FPGA flash deployment settings.

Deploying Application Images

To deploy an application image to one or more targets, select the image in the right table and select the desired real-time targets in the left table. You can select multiple real-time targets using <Ctrl-Click> or <Shift-Click> or by clicking the Select All button. After completing your selection, click on Deploy at the center of the UI. A dialog confirms your real-time target selection and allows for additional target configuration options.

After verifying the selected targets and setting any desired network configuration options for your real-time targets, click on Deploy Application Image to Listed Targets to begin the deployment process. The application image is deployed to the selected targets in sequential order, one at a time. During this process, the Figure 11.21 progress dialog is shown. This process takes several minutes per real-time target.

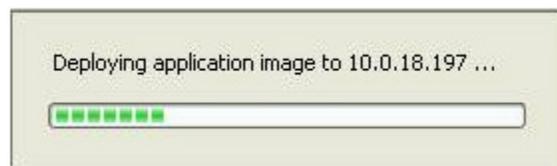


Figure 11.21. Application Image Deployment Process

Comparing Application Images

In addition to retrieving and deploying application images, the utility offers an image comparison feature. The comparison tool compares each file in two images to notify you if any files have been modified, added, or removed. You can choose to compare a local image with an image on a controller, or two local images.

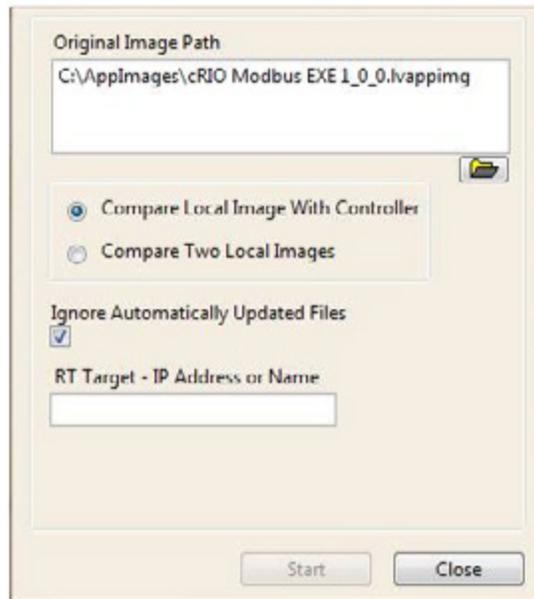


Figure 11.22. Compare a local image to network controller dialog.

Click the Start button to begin the comparison process. Once the comparison begins, each file on both images is compared for changes. Files that are found in one image and not the other are also identified. This process may take several minutes to complete, and a progress dialog is displayed until completion. If the tool completes with no differences found, the Figure 11.23 dialog appears.



Figure 11.23. Identical Images Notification

However, if any differences are identified, they are listed in Table 11.1. You can then log the results to a TSV file.

Image differences are listed below...		
List of Modified Files	Files Found Only In Original Image (local)	Files Found Only In Comparison Image (local or remote)
c:\ni-rt\config\criocfg.bin	c:\lvappimage.info	c:\ni-rt\config\nimcdata.xml
c:\ni-rt\config\masterRegistry.xml	c:\ni-rt\system\mxsCheckpoints\20101119_225018.cpt\config3.mxs	c:\ni-rt\system\dmECAT3rdPartyComm.out
c:\ni-rt\config\scanConfig.xml		c:\ni-rt\system\dmECATComm.out
c:\ni-rt\config\variables.xml		c:\ni-rt\system\dmRIOComm.out
c:\ni-rt\system\config.cdf		c:\ni-rt\system\dmsimcom.out
c:\ni-rt\system\mxsjar.ini		c:\ni-rt\system\nimcca.out
c:\ni-rt\system\mxsjar.mxs		c:\ni-rt\system\nimcdm.out
c:\ni-rt\system\mxsSchema.log		c:\ni-rt\system\nimcdmtg.out
c:\ni-rt\custom\nimcdm.ini		c:\ni-rt\custom\nimcdm\shMEMArbiter.out

Table 11.1. Image Comparison Results

Deploying CompactRIO Application Updates Using a USB Memory Stick

If your CompactRIO systems are not available on the network, you may want to deploy an image using a USB stick. The process of updating or deploying a new application image from a USB memory device to a CompactRIO controller is based on a simple file copy operation that replaces the files on the hard drive of the CompactRIO controller with files stored on the USB memory device. This file copy operation is added as a VI to the main LabVIEW Real-Time application deployed to the controller.

A LabVIEW Real-Time application once loaded and running on the controller is stored completely in the active memory (RAM) of the controller. Because of this, you can delete and replace the application files stored on the controller hard drive while the application is running in memory. The new application becomes active by performing a reboot operation on the controller and loading the new application during the boot process.

To update the deployed code from the USB memory device in the future, you must add code to the main application that handles the deployment process. The Deploy Image.vi shown in Figure 11.24 handles the update process.

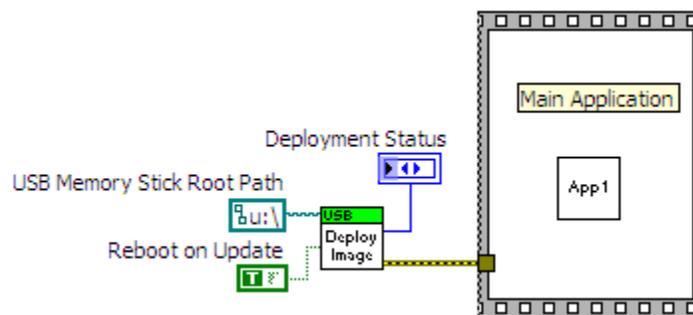


Figure 11.24. Include the USB Deploy Image VI in your deployed real-time application to enable updates from a USB memory stick.

Once you have added this code, you can build the main application into an executable and deploy it to a CompactRIO controller. The deployed application executable, together with all of the other files on the CompactRIO controller hard drive, becomes the application image. For more information on this utility including downloadable files, see the NI Developer Zone document [Reference Design for Deploying CompactRIO Application Updates Using a USB Memory Device](#).

APIs for Developing Custom Imaging Utilities

You can choose from several imaging APIs—all with nearly identical functionality—depending on the version of LabVIEW you are using.

- **System Configuration API**—Recommended for LabVIEW 2011 or later
- **RT Utilities API**—Recommended only for LabVIEW 2009 and 2010
- **NI System Replication Tools**—Recommended for LabVIEW 8.6 or earlier

Another useful API discussed in this section is the CompactRIO Information Library (CRI), which you can use with the three APIs listed above to return information about the current hardware configuration of a CompactRIO system.

System Configuration API

The System Configuration API exposes MAX features programmatically. For example, you can use this API to programmatically install software sets, run self-tests, and apply network settings in addition to image retrieval

and deployment. This API is located in the Functions palette under **Measurement I/O»System Configuration»Real-Time Software**.

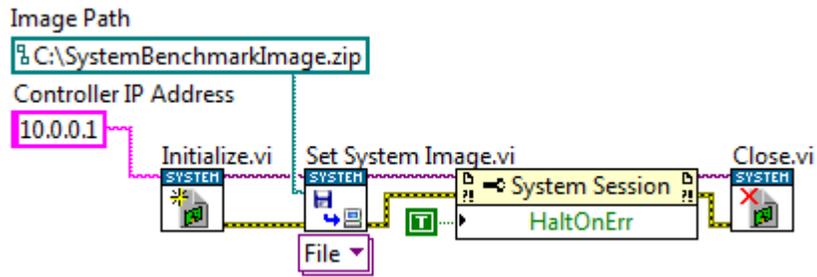


Figure 11.25. Programmatically Setting a System Image Using the System Configuration API

RT Utilities API

The RT Utilities API also includes VIs for image retrieval and deployment. It was deprecated in LabVIEW 2011 and is not recommended for new designs. The functionality it offers has been fully absorbed into the System Configuration API

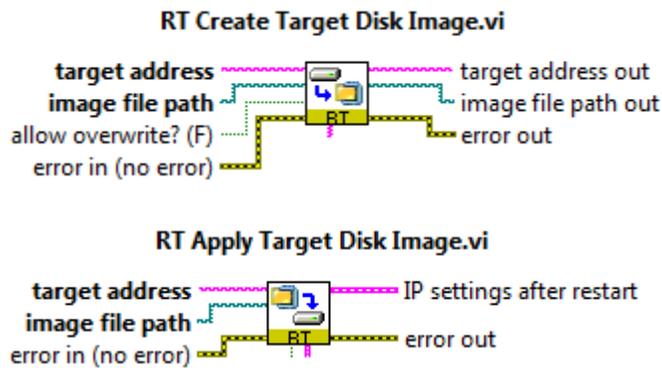


Figure 11.26. You also can use the RT Utilities VIs in LabVIEW 2009 or later to programmatically deploy and retrieve a system image.

LabVIEW Real-Time System Replication Tools

For LabVIEW 8.6 and earlier, neither the RT Utilities API nor the System Configuration API is available. For these applications, NI recommends using LabVIEW Real-Time system replication tools. For more information on these tools, including the downloadable installation files, see the NI Developer Zone document [Real-Time Target System Replication](#).

CompactRIO Information Library (CRI)

You can use the CRI with the three APIs discussed above to detect the current configuration of a CompactRIO system. The CompactRIO Information component provides VIs 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.

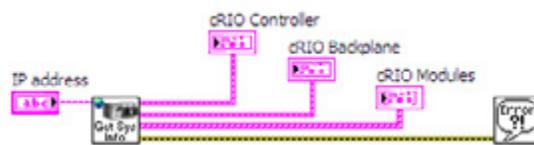


Figure 11.27. CRI Get Remote cRIO System Info.vi

You can download this library from the NI Developer Zone document [Reference Library for Reading CompactRIO System Configuration Information](#).

IP Protection

Intellectual property (IP) in this context refers to any unique software or application algorithm(s) that you have 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 gives companies a way to differentiate from the competition; therefore, protecting this software IP is important. LabVIEW development tools and CompactRIO provide the ability to protect and lock your IP. In general, you can implement two levels of IP protection:

Lock Algorithms or Code to Prevent IP From Being Copied or Modified

If you have created algorithms for a specific function, such as performing advanced control functions, implementing custom filtering, and so on, you may want to distribute the algorithm as a subVI but prevent someone from viewing or modifying that actual algorithm. This may be to achieve IP protection or 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

Use this method if you want to ensure that a competitor cannot replicate your system by running your code on another CompactRIO system or if you 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 startup application on a CompactRIO controller is by default locked and cannot be opened. Unlike other off-the-shelf controllers or some PLCs for which the raw source code is stored on the controller and protected only 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 bitfile and cannot be decompiled back to LabVIEW code. To aid in future debugging and maintenance, you can store the LabVIEW project on the controller or 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 offers a few ways to provide usable subVIs while 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 VI prohibits others from editing the VI or viewing its block diagram without the password. However, if the password is lost, you cannot 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.

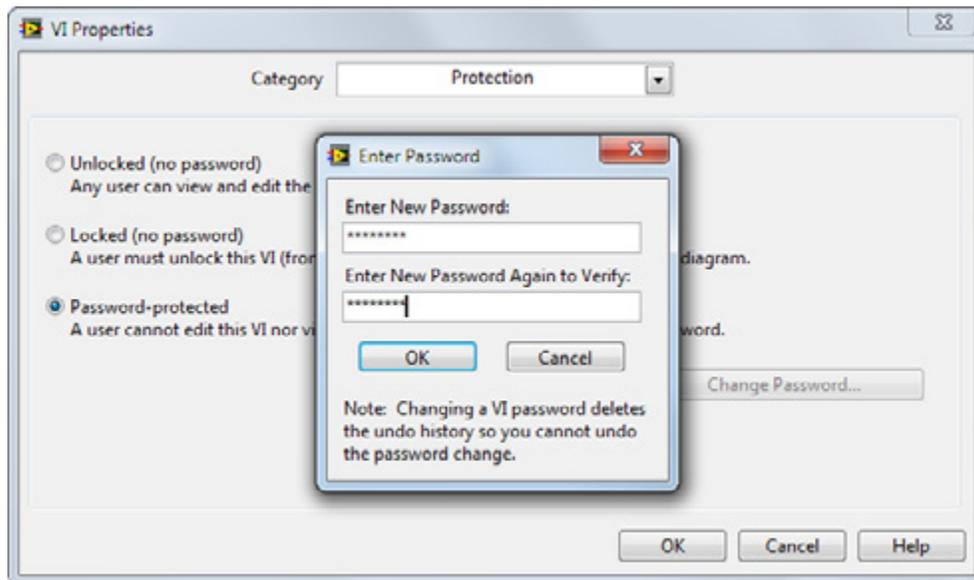


Figure 11.28. Password Protecting LabVIEW Code

The LabVIEW password mechanism is quite difficult to defeat, but no password algorithm is 100 percent secure from attack. If you need total assurance that someone cannot gain access to your source code, you should consider removing the block diagram.

Method 2: Removing the Block Diagram

To go beyond the protection offered by password protecting a VI and 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. Do not forget to make a backup of your files if you use this technique because the block diagram 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.

- 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.
- 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.
- Build the source distribution to create a copy of the VI without its block diagram.

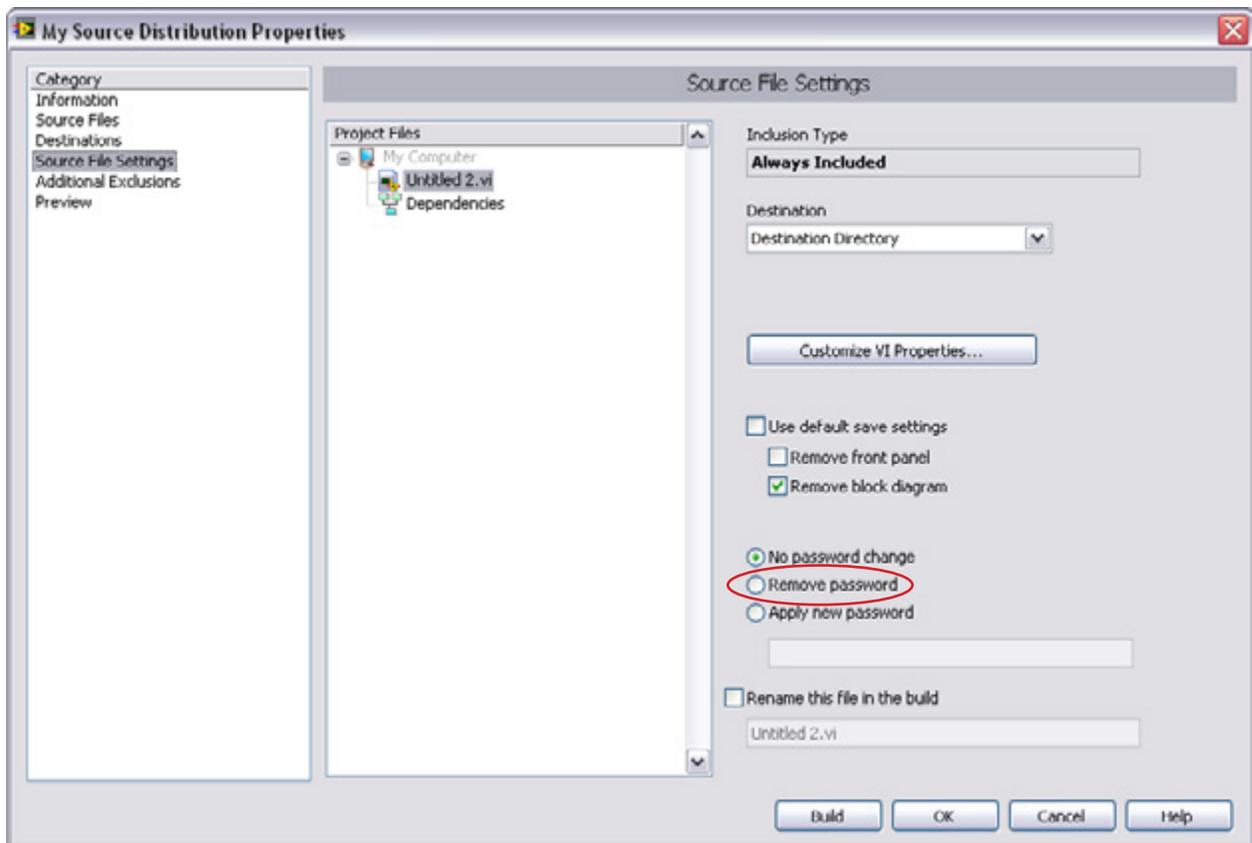


Figure 11.29. Removing the Block Diagram From LabVIEW VIs

Note: 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 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 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 following:

- 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

You can use the following steps 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 following procedures for more guidance 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 CompactRIO hardware.

License Key

Adding licensing to a LabVIEW Real-Time application can protect a deployed application from being copied and run on another similar or identical set of hardware without obtaining a license from the vendor or distributor of the application. Most modern-day applications running on desktop computers are protected by a license key that is necessary to either install the application or run it in its normal operational mode. Many vendors use license keys to determine if an application runs in demo mode or is fully functional. License keys may also be used to differentiate between versions of an application or to enable/disable specific features.

You can add this behavior to a LabVIEW Real-Time application using the reference design and example code developed by NI Systems Engineering. You can download this code from the NI Developer Zone document [Reference Design for Adding Licensing to LabVIEW Real-Time Applications](#). The main modification is how you create a unique system ID for a specific hardware target. In the case of CompactRIO, you use the controller, backplane, and module serial numbers. For other targets, you may use the serial number or Ethernet MAC address of a given target to create a unique system ID.

Choosing a License Model

The license model defines how the license key is generated based on particular characteristics of the target hardware. One simple example of a license model is to base the license key on the serial number of the controller. In this case, the application runs if the controller has the serial number matching the license key provided. If the controller serial number does not match the license key, the application does not run.

If the hardware target includes additional components such as a CompactRIO backplane and CompactRIO modules, you may choose to lock the application not only to the controller but also to these additional system components. In this case, you can link the license key to the serial numbers of all of these hardware target components. All components with the correct serial numbers must be in place to run the application.

The unique characteristic of the hardware target (for example, a serial number) is called the system ID for the purpose of this guide.

One example of a more complex license model is to base the license key on the serial numbers of multiple system components but require only some of these components to be present when running the licensed application. This allows the application user to replace some of the system components, in case a repair is required, without needing to acquire a new application license key. The number of components linked to the license key that must be present to run the application is defined by the developer as part of the license model.

Application Licensing Process

Adding licensing (creating and using a license key) to a LabVIEW Real-Time application consists of the following steps:

1. Create a unique system ID for each deployed hardware target
2. Create a license key based on the system ID
3. Store the license key on the hardware target
4. Verify the license key during application startup

Create a Unique System ID for Each Deployed Hardware Target

To create a license key for a specific hardware target, you must first create a unique system ID. The system ID is a number or string that uniquely identifies a specific hardware target based on the physical hardware itself. The licensed application is locked to the hardware target using the license key, which is linked to the system ID. The system ID can be a number such as the serial number of the target or a value that is a combination of each of the system component's serial numbers. Another source for a system ID can be the Media Access Control (MAC) address stored in every Ethernet interface chipset. The example uses the [Reference Library for Reading CompactRIO System Configuration Information](#) to retrieve these different pieces of information from a CompactRIO system.

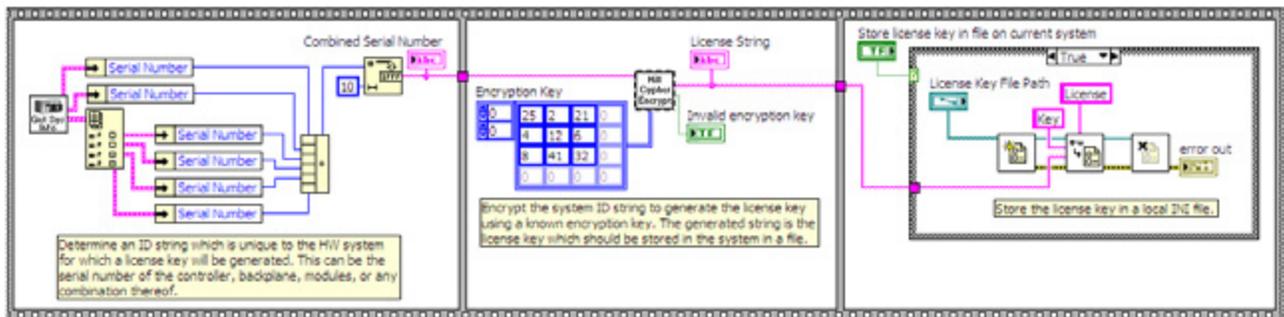


Figure 11.30. Block Diagram Showing the Process of Creating the License Key

Figure 11.31 shows one possible example of generating a system ID for a 4-slot CompactRIO system using these VIs. The serial numbers for all six system components are added together. While this is not a truly unique value for a given CompactRIO system, it is unlikely that after replacing one or more system components, the sum of all the serial numbers will be the same as when the license key was generated.

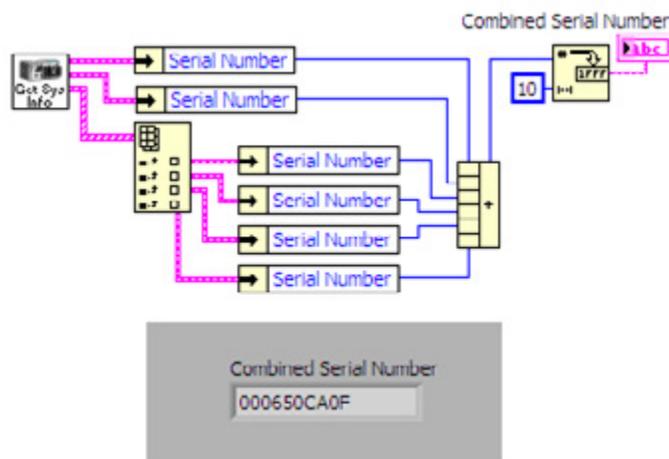


Figure 11.31. Generating a System ID Using Serial Numbers

Create a License Key Based on the System ID

Once you create a system ID, you derive a license key from the system ID. The license key is a unique string that is linked to the system ID using an encryption algorithm. The goal is to create a string that can be verified against the system ID but cannot be easily generated by the user for a different hardware target to disallow using the application on a different set of hardware.

Encryption

The encryption used in the reference example is a version of the Hill Cipher, which uses a simple matrix multiplication and a license key matrix. The matrix multiplication converts the input string into a different output string that cannot be easily decrypted without knowing the key matrix. The Hill Cipher is not considered to be a strong encryption algorithm and is only one possible option you can use to generate a license key. You can easily replace the encryption subVI in the reference example with your own encryption tool.

The encryption VI provided with the reference example converts the system ID using the Hill Cipher and a 3x3 key matrix (Encryption Key). The key matrix is the key to unlocking the license key; therefore, you should change the key matrix value before using the reference example in a real-world application. Not all 3x3 matrices are valid for use as a key matrix. The Hill Cipher Encryption VI tells you if your chosen matrix is invalid. If it is invalid, try other values for your key matrix.

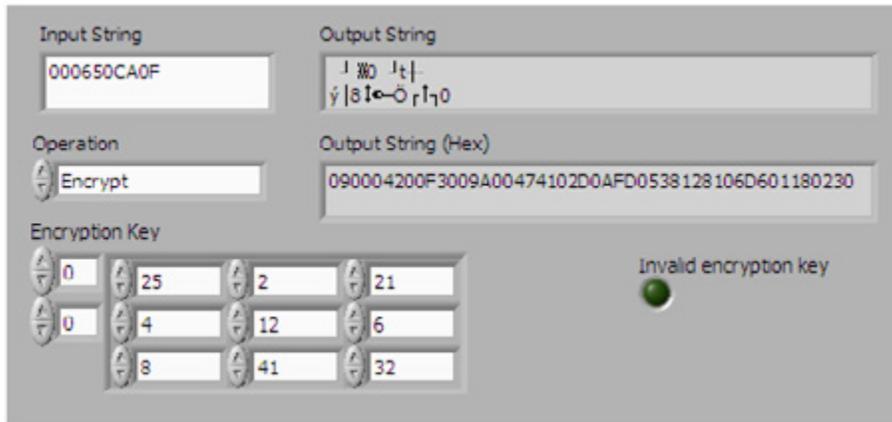


Figure 11.32. Encryption of the System ID Into the License Key

The encryption VI provides two versions of the license key. The default algorithm returns a stream of bytes that can have any value between 0 and 255; therefore, these bytes may not represent printable characters and may not be easily stored in a text file or passed along using other mechanisms such as email. To simplify the process of storing and transferring the license key, the VI provides a hexadecimal version of the encrypted string made up of the ASCII representation of the hexadecimal value of each of the byte values in the original string. This string is stored in a file as the license key.

Store the License Key on the Hardware Target

The reference example stores the license key in a simple INI file on the CompactRIO controller hard drive.

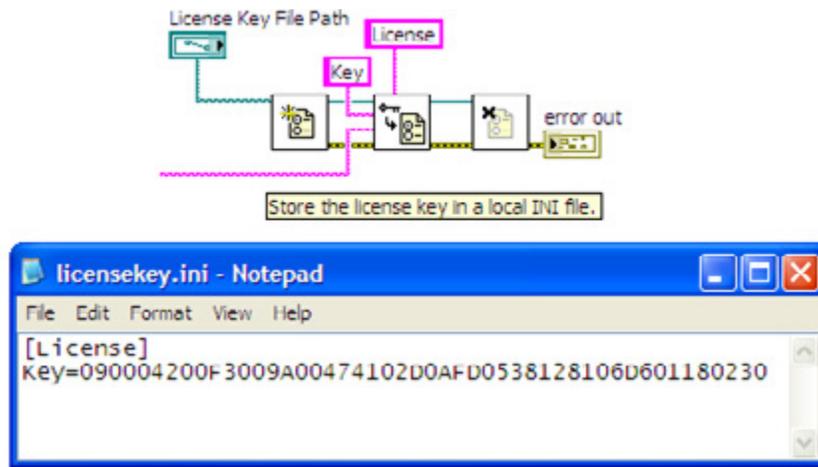


Figure 11.33. Storing the License Key in an INI File on the CompactRIO Controller

If the license key and file are generated away from the actual CompactRIO system, then you must copy the license file to the CompactRIO system when you deploy the application to the controller.

Verify the License Key During Application Startup

When the deployed application is starting up, it needs to verify the license key and, based on the results of the verification process, adjust its behavior according to the license model. If the license key is verified correctly, the application runs normally, but if the key is not verified, it may not run at all or run in an evaluation mode.

The reference example provides a basic verification VI that is added before the actual application.

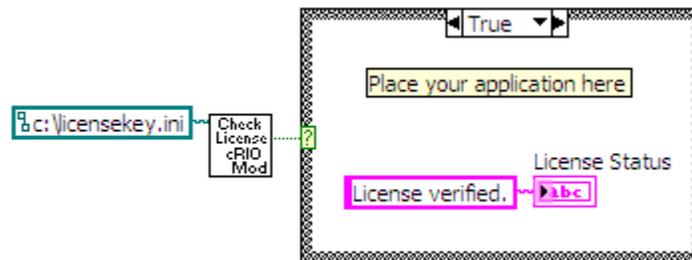


Figure 11.34. Adding the License Key Verification to an Application

You can choose from two different methods to verify a license key. The first and preferred method is to re-create the license key on the target system (described in this section). The second method, which consists of decrypting the license key, is described in the NI Developer Zone white paper [Reference Design for Adding Licensing to LabVIEW Real-Time Applications](#) under the section titled "Enabling Features based on the License Key".

The more basic and more secure method to verify the license key is to run the same algorithm you use to create the license key on the target system and then compare the new license key with the license key stored in the license file. If the two match, then the license key is verified and the application may run.

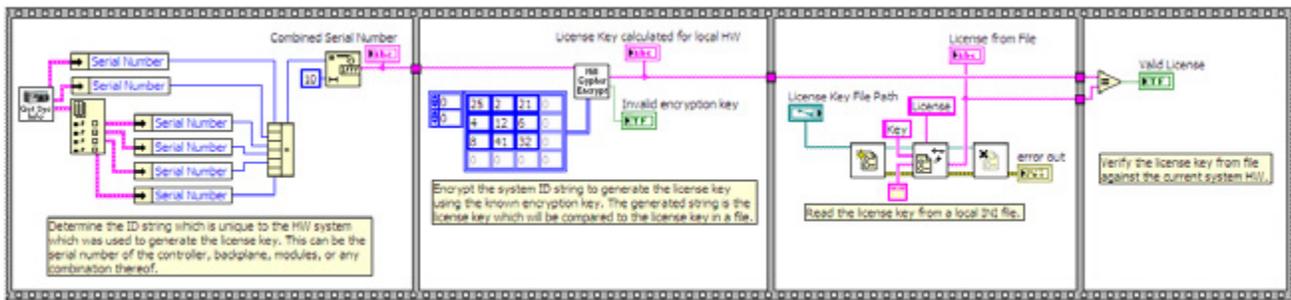


Figure 11.35. Block Diagram to Verify the License Key Stored in the File on the System

Figure 11.35 shows that this process is almost identical to the process of generating the license key. Instead of writing the license file, however, the license file is read and compared to the newly generated license key.

This method of verifying the license key works well if you do not need any partial information from the license key such as information about enabling or disabling individual features or the individual serial numbers of system components. For licensing models that require more detailed information about the license key, the key itself must be decrypted. For more information on this type of licensing, see the NI Developer Zone document [Reference Design for Adding Licensing to LabVIEW Real-Time Applications](#).

Deploying Applications to a Touch Panel

Configure the Connection to the Touch Panel

Although you can manually copy built applications to a touch panel device, you should use Ethernet and allow the LabVIEW project to automatically download the application. NI touch panels are all shipped 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 NI TPC Service connection and enter the IP address of the touch panel. Test the connection to make sure the service is running.

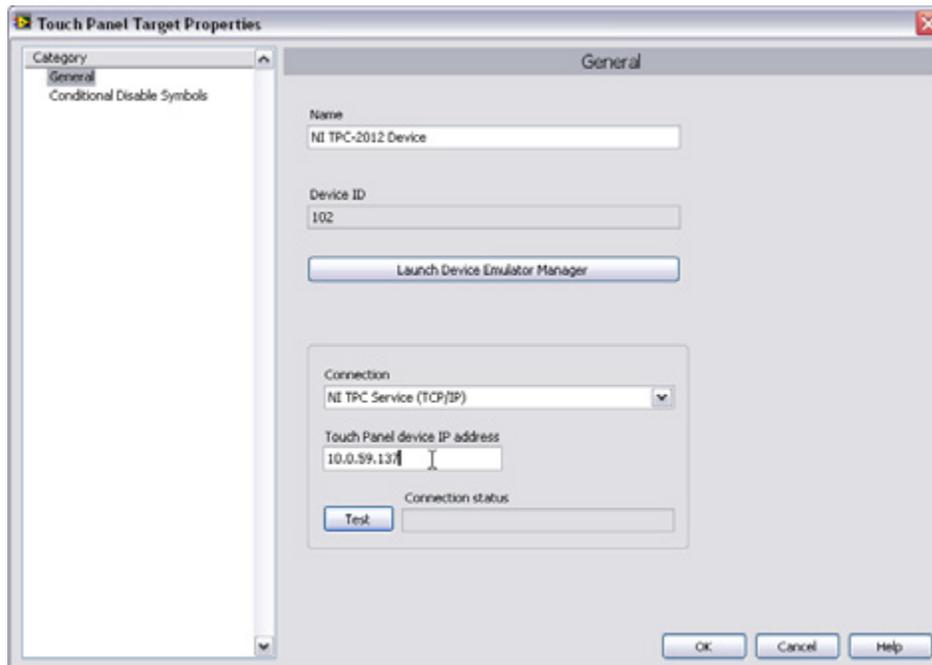


Figure 11.36. Connect to a touch panel through Ethernet using the NI TPC Service.

You can find the IP address of the touch panel by going to the command prompt on the TPC and typing ipconfig. To get to the command prompt, go to the Start menu and select Run... In the pop-up window, enter cmd.

Deploy a LabVIEW VI to Volatile or Nonvolatile Memory

The steps to deploy an application to a Windows XP Embedded touch panel and to a Windows CE touch panel are nearly identical. The only difference is on an XP Embedded touch panel, you can deploy an application to only the nonvolatile memory, and, on a Windows CE touch panel, you can deploy to 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 do this, 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.

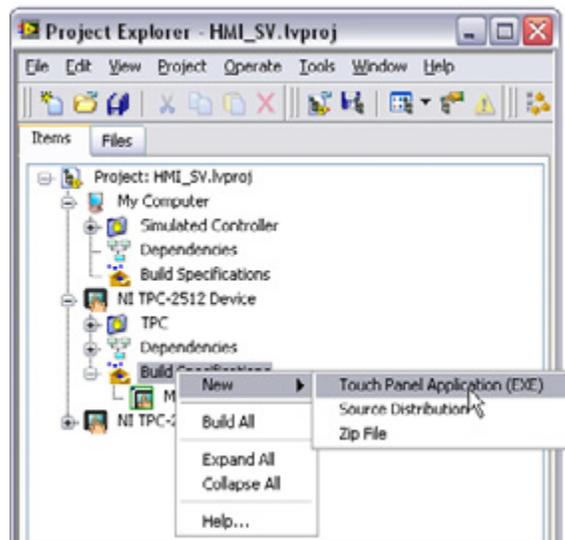


Figure 11.37. Create a touch panel application using the LabVIEW project.

After selecting the Touch Panel Application, you are presented with a dialog box. The two 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.

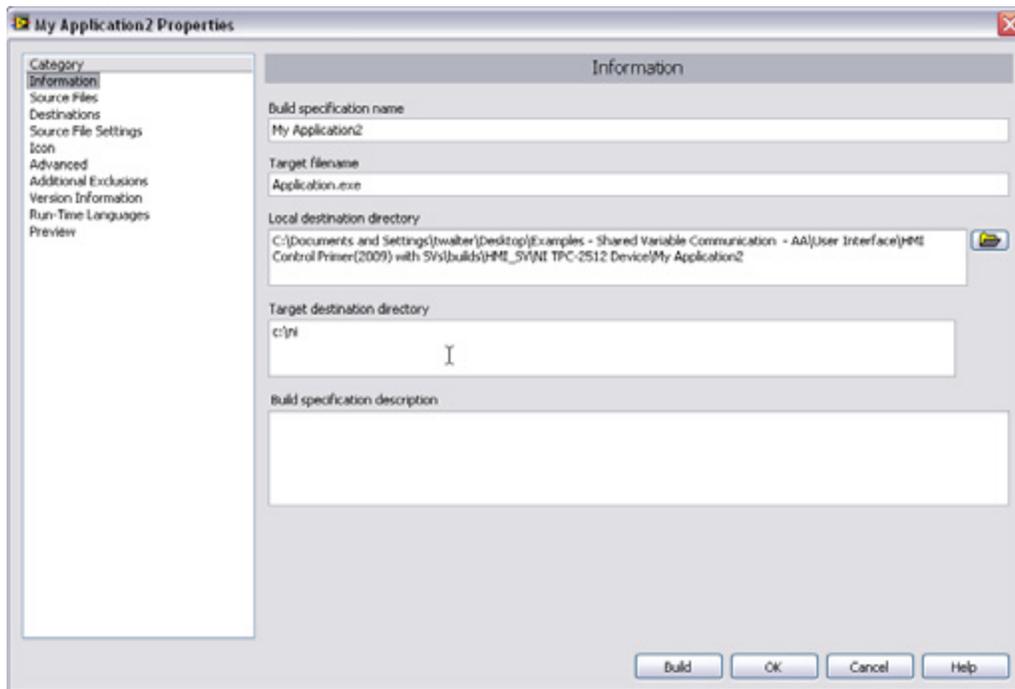


Figure 11.38. The Information Category in the Touch Panel Application Properties

You use the Source Files category to set the startup VIs and obtain 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 lplib or set subVIs as Startup VIs or Always Included unless they are called dynamically in your application.

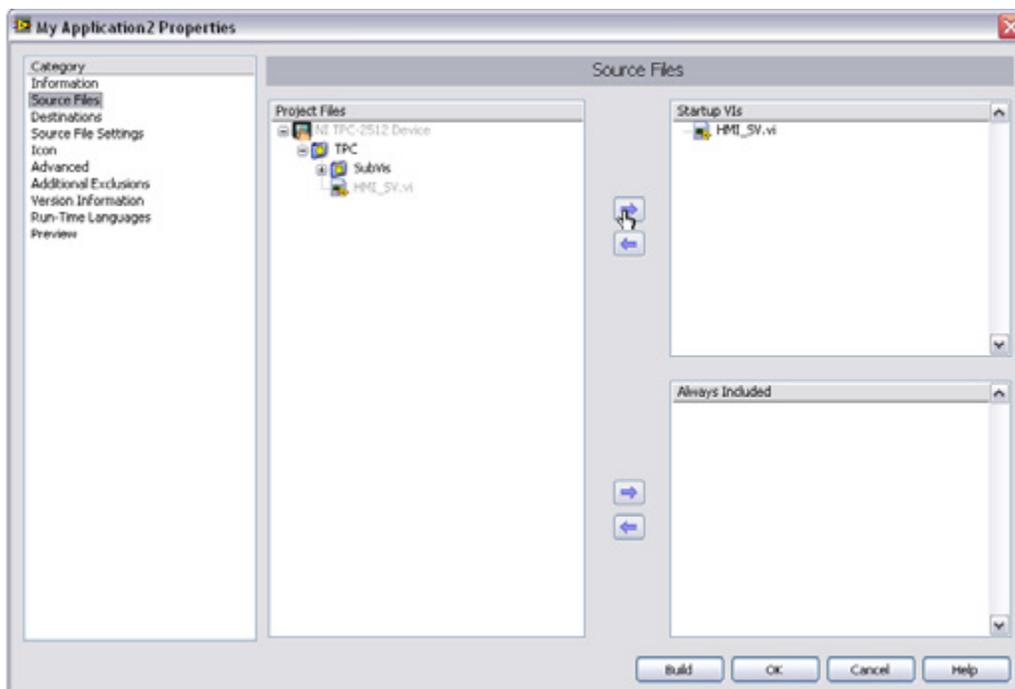


Figure 11.39. Source Files Category in the Touch Panel Application Properties
(In this example, the HMI_SV.vi was selected to be a Startup VI.)

After you have entered all of the information 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.

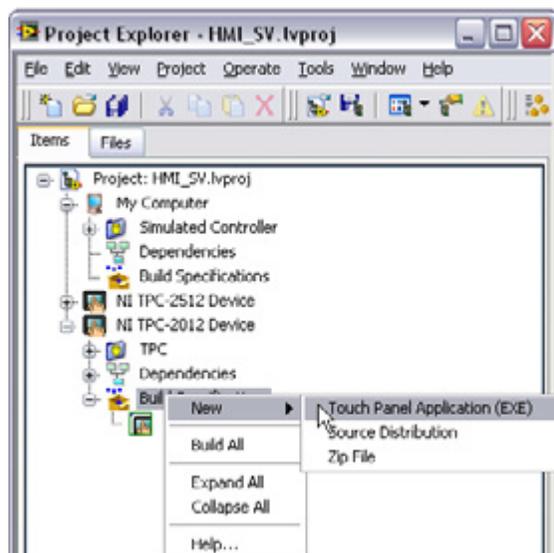


Figure 11.40. Creating a Touch Panel Application Using the LabVIEW Project

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.

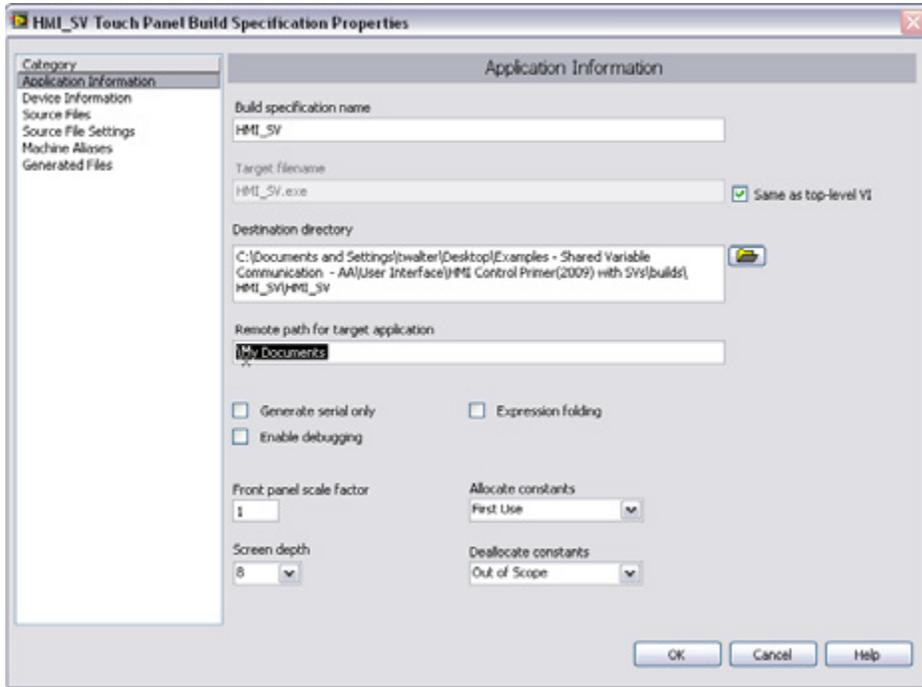


Figure 11.41. The Information Category in the Touch Panel Application Properties

Use the Source Files category to set the startup VI and obtain 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 lplib or subVIs as Always Included.

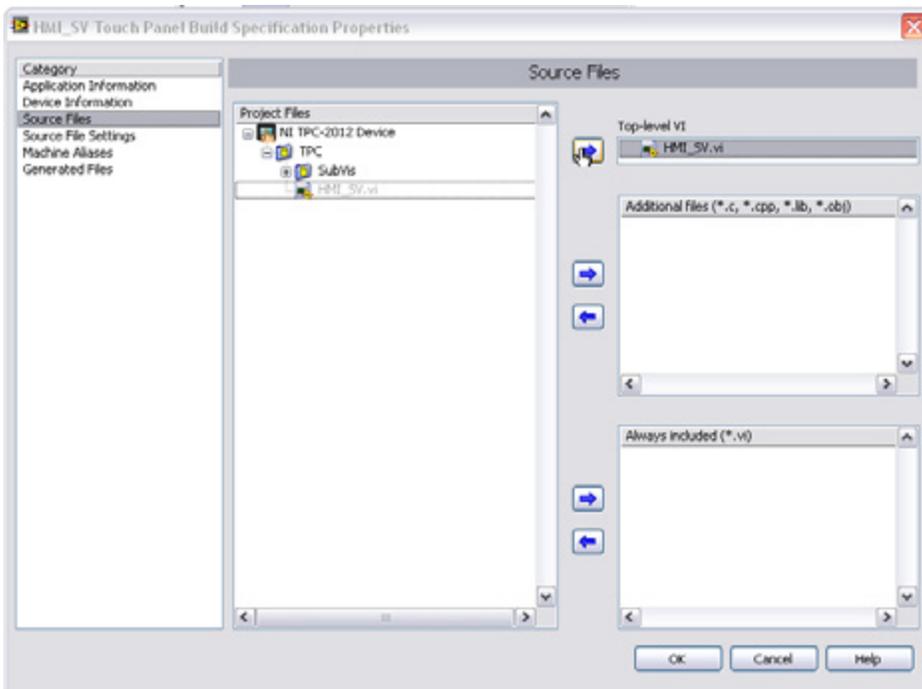


Figure 11.42. The Source Files Category in the Touch Panel Application Properties
(In this example, the HMI_SV.vi was selected to be the top-level VI.)

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 checkbox. The alias list

Misc tab in the Configuration Utility (**Start»Programs»Utilities»Configuration Utilities**) to configure a program to start up on boot. This utility modifies the startup.ini file for you.

Porting to Other Platforms

This guide has focused on architectures for building embedded control systems using CompactRIO systems. The same basic techniques and structures also work on other NI 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.

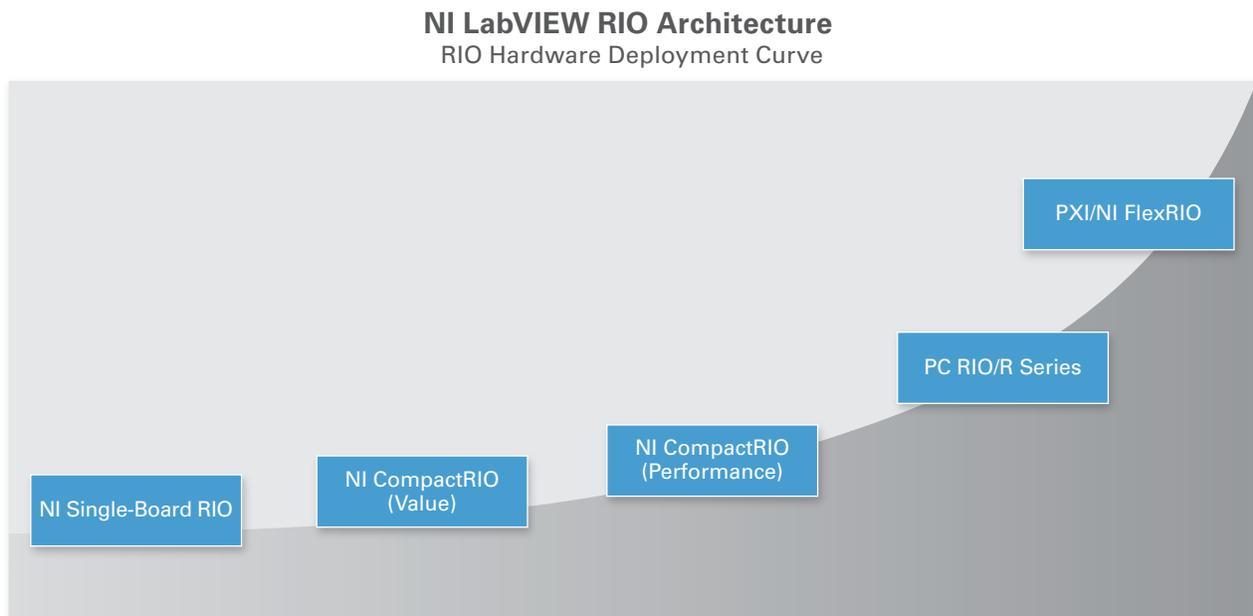


Figure 11.44. With LabVIEW, you can use the same architecture for applications ranging from CompactRIO to high-performance PXI to board-level NI Single-Board RIO.

LabVIEW Code Portability

LabVIEW is a cross-platform programming language capable of compiling for multiple processor architectures and OSs. In most cases, algorithms written in LabVIEW are portable among all LabVIEW targets. In fact, you can even take LabVIEW code and compile it for any arbitrary 32-bit processor 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

NI Single-Board RIO is a board-only version of CompactRIO designed for applications requiring a bare board form factor. While it is physically a different design, NI Single-Board RIO uses the processor and FPGA, and most models

accept up to three C Series modules. NI Single-Board RIO differs from CompactRIO because it includes I/O built directly into the board. NI offers two families of NI-Single-Board RIO products:

Digital I/O With RIO Mezzanine Card Connector

The smallest option for NI Single-Board RIO combines the highest performance real-time processor with a Xilinx Spartan-6 FPGA and built-in peripherals such as USB, RS232, CAN, and Ethernet. In addition to the peripherals, the system includes 96 FPGA digital I/O lines that are accessed through the RIO Mezzanine Card (RMC) connector, which is a high-density, high-bandwidth connector that allows for direct access to the FPGA and processor. With this type of NI Single-Board RIO, you can create a customized daughter card designed specifically for your application that accesses the digital I/O lines and processor I/O including CAN and USB. This NI Single-Board RIO family currently does not support C Series connectivity.

Digital I/O Only or Digital and Analog I/O With Direct C Series Connectivity

NI also offers NI Single-Board RIO devices with both built-in digital and analog I/O on a single board. All I/O is connected directly to the FPGA, providing low-level customization of timing and I/O signal processing. These devices feature 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. They can directly connect up to three C Series I/O and communication modules for further I/O expansion and flexibility.

LabVIEW FPGA Programming

Not all NI Single-Board RIO products currently support Scan Mode. Specifically, the NI Single-Board RIO products with a 1 million gate FPGA (sbRIO-9601, sbRIO-9611, sbRIO-9631, and sbRIO-9641) are not supported, in addition to the NI Single-Board RIO products with the RMC connector (sbRIO-9605 and sbRIO-9606). Instead of using 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 an I/O memory table. This section examines an effective FPGA architecture for single-point I/O communication similar to Scan Mode and shows how to convert an application using Scan Mode.

Built-In I/O and I/O Modules

Depending on your application I/O requirements, 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 with direct C Series connectivity 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)

This NI Single-Board RIO family 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 Spartan-6 LX45. CompactRIO targets offer versions using both the Virtex-5 FPGAs and the Spartan-6 FPGAs as large as the LX150. 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-6 LX45.

Port CompactRIO Applications to NI Single-Board RIO or R Series Devices

Follow these four main steps to port a CompactRIO application to NI Single-Board RIO or PXI/PCI R Series FPGA I/O devices.

5. Build an NI Single-Board RIO or R Series project with equivalent I/O channels.
6. If using the NI RIO Scan Interface, build a LabVIEW FPGA-based scan API if porting to unsupported NI Single-Board RIO devices or to PXI/PCI R Series FPGA I/O Devices
 - Build LabVIEW FPGA I/O scan (analog in, analog out, digital I/O, specialty digital I/O).
 - Convert I/O variable aliases to single-process shared variables with real-time FIFO enabled.
 - Build a real-time I/O scan with scaling and a shared variable-based current value table.
7. Compile LabVIEW FPGA VI for new target.
8. Test and validate updated real-time and FPGA code.

The first step in porting an application from CompactRIO to NI Single-Board RIO or an 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 DAQ device.

Step two is necessary only if the application being ported was originally written using the NI RIO Scan Interface, and if porting to unsupported NI Single-Board RIO or PXI/PCI R Series FPGA I/O Devices. If you need to replace the NI RIO Scan Interface portion of an application with an I/O method supported on all RIO targets, an example is included below to guide you through the process.

If the application you are migrating to NI Single-Board RIO or PXI/PCI R Series did not use the RIO Scan Interface, 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 on the RIO architecture and reusable modular C Series I/O modules, porting applications between these two targets is simple.

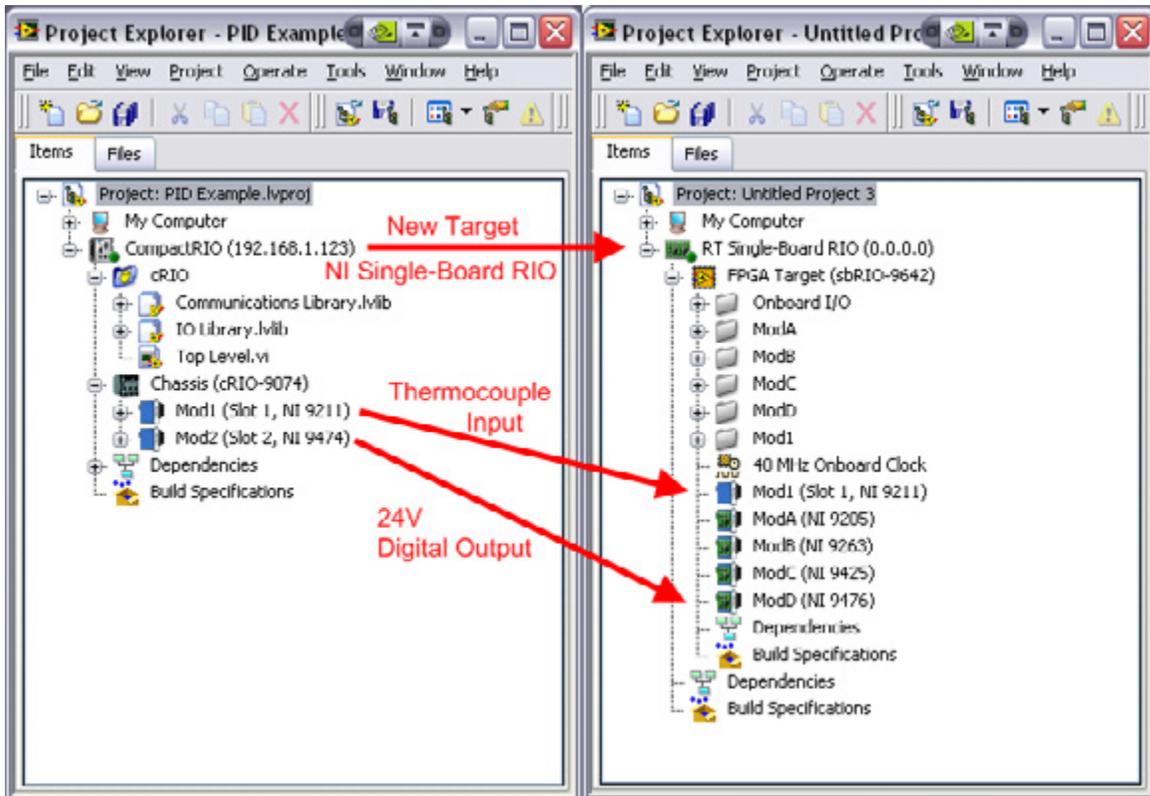


Figure 11.45. The first step in porting an application from CompactRIO to an alternate target is finding replacement I/O on the future target.

Example of Porting a RIO Scan Interface-Based Application to Use LabVIEW FPGA

If you used the RIO Scan Interface in your original application, you might need to create a simplified FPGA version of the Scan Engine. Use these three steps to replace the RIO Scan Interface with a similar FPGA-based scan engine and current value table:

9. Build a LabVIEW FPGA I/O scan engine
10. Replace scan engine I/O variables with single-process shared variables
11. Write FPGA data to a 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 encoders. Next, create an FPGA Scan Loop that synchronizes the I/O updates by updating all outputs and reading the current value of all inputs in sequence.

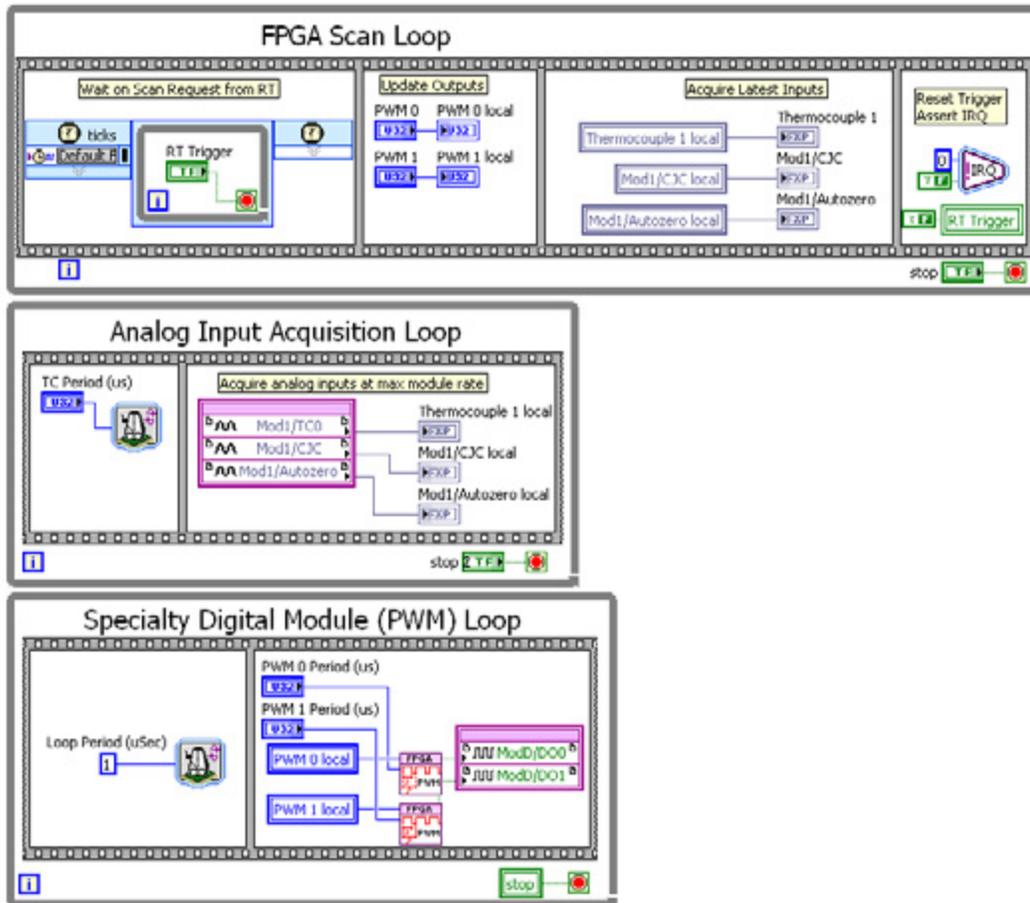


Figure 11.46. Develop a simple FPGA application to act as an FPGA scan engine.

After you have implemented a simple scan engine in the FPGA, you need to port the real-time portion of the application to communicate with the custom FPGA scan engine rather than the scan engine I/O variables. 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, single-process shared variables are not updated by a driver. You can change the type by going to the properties page for each I/O variable alias and changing it to single-process.

Tip: If you have numerous variables to convert, you can easily convert a library of I/O variable aliases to shared variables by exporting to a text editor and changing the properties. To make sure you get the properties correct, you should first create one "dummy" single-process shared variable with the single-element real-time FIFO enabled in the library and then export the library to a spreadsheet editor. While in the spreadsheet editor, delete the columns exclusive to I/O variables and copy the data exclusive to the shared variables to the I/O variable rows. Then import the modified library into your new project. The I/O variable aliases are imported as single-process shared variables. Because LabVIEW references shared variables and I/O variable aliases by the name of the library and the name of the variable, all instances of I/O variable aliases in your VI are automatically updated. Finally, delete the dummy shared variable that you created before the migration process.

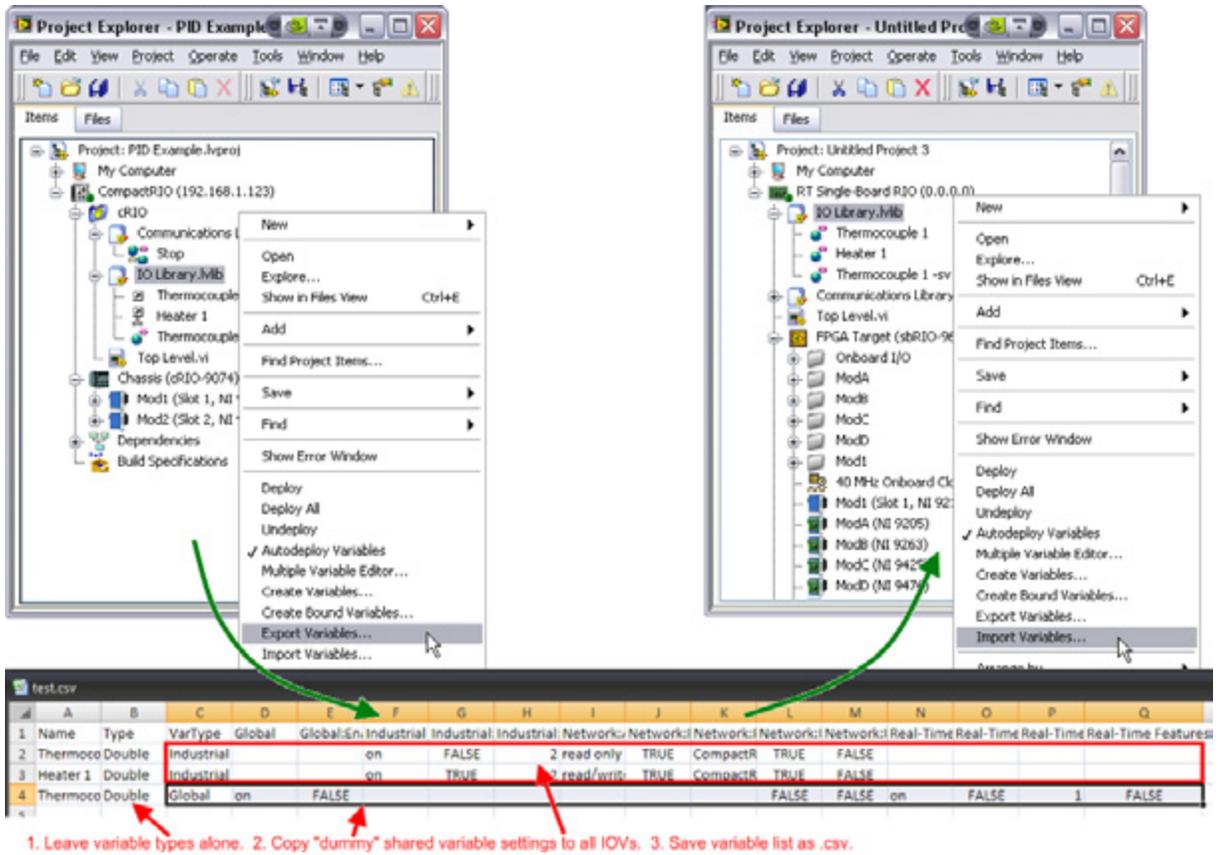


Figure 11.47. You can easily convert an I/O variable (IOV) alias library to shared variables by exporting the variables to a spreadsheet, modifying the parameters, and importing them into your new target.

The final step for implementing an FPGA scan engine is adding a real-time process to read data from the FPGA and constantly update the current value table. The FPGA I/O you are adding to the shared variables is deterministic, so you can use a Timed Loop for implementing this process.

To read data from the FPGA 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 you should set it 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. You also need to change the timing sources of any other task loops in your application that were previously synchronized to the Scan Mode to the 1 kHz clock and set them to the same rate as the I/O scan 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 the scaling and calibration of analog and specialty digital I/O.

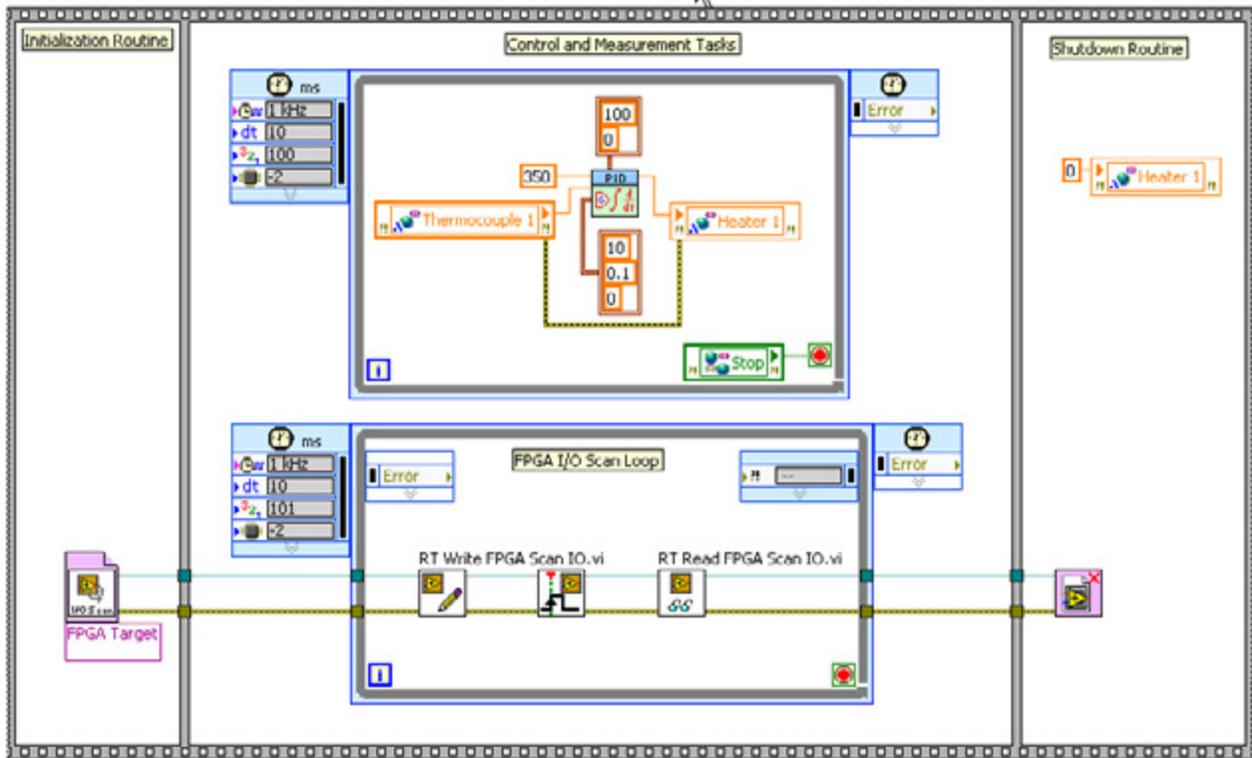


Figure 11.48. The FPGA I/O scan loop mimics the RIO Scan Interface feature by deterministically communicating the most recent input and output values to and from the FPGA I/O and inserting the data into a current value table.

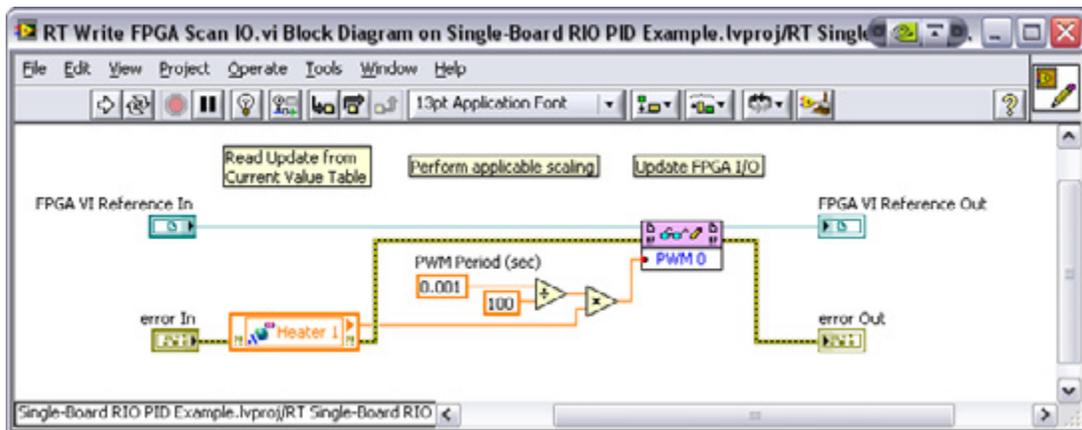


Figure 11.49. The RT Write FPGA Scan IO VI pulls current data using a real-time FIFO single-process shared variable, scales values with appropriate conversion for the FPGA Scan VI, and pushes values to the FPGA VI.

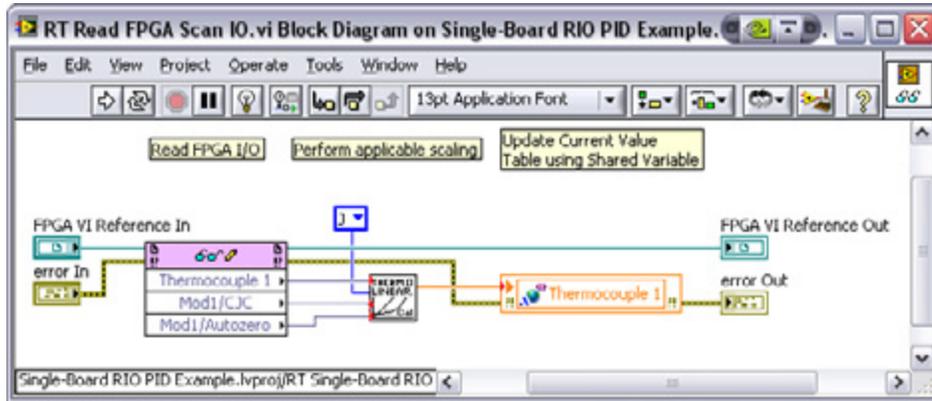


Figure 11.50. The RT Read FPGA Scan IO VI pulls all updates from the FPGA Scan IO VI, performs applicable conversions and scaling, and publishes data to a current value table using a real-time FIFO single-process shared variable.

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 the specialized features of one platform, such as the RIO Scan Interface, the porting process is more involved, but only the I/O portions of the code require change for migration. In both situations, all of the LabVIEW processing and control algorithms are completely portable and reusable across RIO hardware platforms.

CHAPTER 12

Security on NI RIO Systems

Security Concerns on RIO Systems

Reconfigurable I/O (RIO) systems (NI Single-Board RIO, NI CompactRIO, and so on) are often used in critical applications. While the application space for the RIO platform is expansive, security concerns with RIO systems can be more narrowly defined.

One key security concern is functional correctness, wherein the I/O of the RIO device is proper. This is a security concern because if functional correctness is compromised, the hardware that the RIO device is connected to can get damaged or malfunction. For example, failed products on an assembly line may pass if the I/O of the RIO device has been compromised, or alternatively, motors and centrifuges controlled by the RIO device may get permanently damaged if they are ramped improperly.

In addition to functional correctness, a key security concern is sensitive data protection. RIO devices often compute, carry, or transmit sensitive data. It's important to ensure that this data is properly protected. Besides the data itself, valuable algorithms are often programmed onto RIO devices as well. Keeping these algorithms safe from theft is also crucial.

The three key security concerns, functional correctness, sensitive data protection, and algorithm protection are seldom considered independently on a RIO system. Often, a compromise in one of these areas leads to a compromise in another.

The Nature of Security on RIO Systems

Security is a complex challenge regardless of the system. It's not a measure that can be addressed once and then forgotten. It must be continually managed. Additionally, definitions of "secure" can vastly differ because they are extremely application specific. Security, fundamentally, comes at a trade-off; more secure systems require greater time and cost investments and sacrifice ease of use.

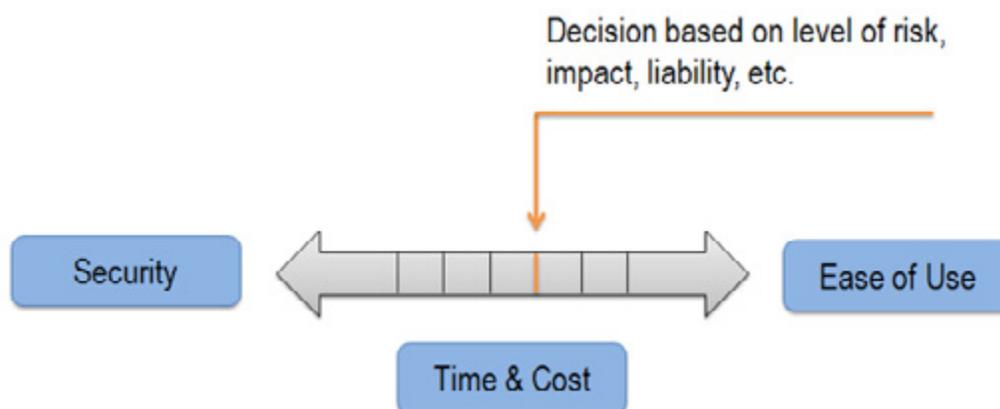


Figure 12.1. The Trade-Off Between Security and Time, Cost, and Ease of Use

Given this trade-off, you need to evaluate the proper investment in security for each application. The security required by an application depends on the liability and scope of the application. National Instruments provides best practices and open channels of communication to help you overcome the challenges that security presents.

Layers of Security

Security can be defined at several different levels in most systems. For a RIO system, security is defined at the following key levels; physical, network, operating system, and application. It's important to have some protection at each level, otherwise, a compromise in one layer can easily lead to a compromise in another. If you invest a lot of time and effort protecting one layer and fail to consider other layers, an intruder may find a way around the well-protected layer and manage to compromise your system with little effort.

First, you need to understand how the four RIO system layers are related. The layers farthest away from the application are the physical and network layers. Security threats to a RIO system can emerge from these two layers. The next affected layer is the operating system layer, which is closest to the application. Lastly, the application layer, which resides at the core, requires the most protection. Steps can be taken at each of the layers to ensure that the overall application is not compromised.

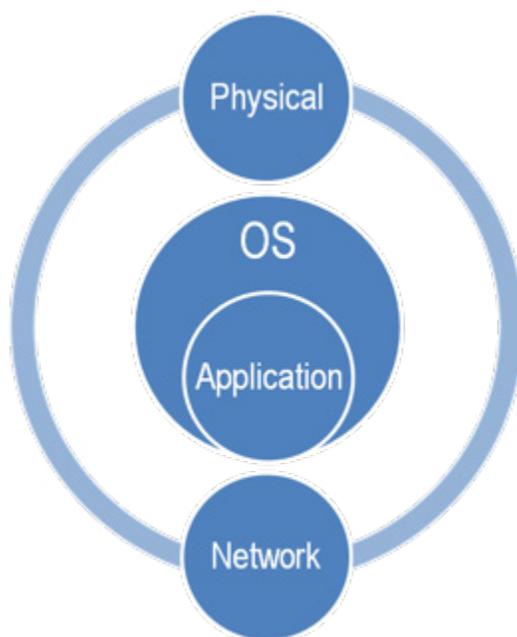


Figure 12.2. The Layers of Security Important to RIO Systems

Security Vectors on RIO Systems

A typical RIO system comprises a host PC and a RIO target, connected over a network, as shown in Figure 13.3. The host PC can be either the development or deployment machine—in some cases, these are the same. The network is often a private local area network (LAN), but it can also be as simple as a crossover connection. As mentioned, threats to the RIO system can originate at the physical or network level. The targets can be either the host PC or the RIO device. Thus you need to consider the threats that can affect both the host PC and the RIO device, and not focus solely on the RIO device itself. To avoid redundancy, the best practices documentation focuses on the threats to the RIO device that originate in the network between the host PC and the RIO and not with the threats to the host PC.

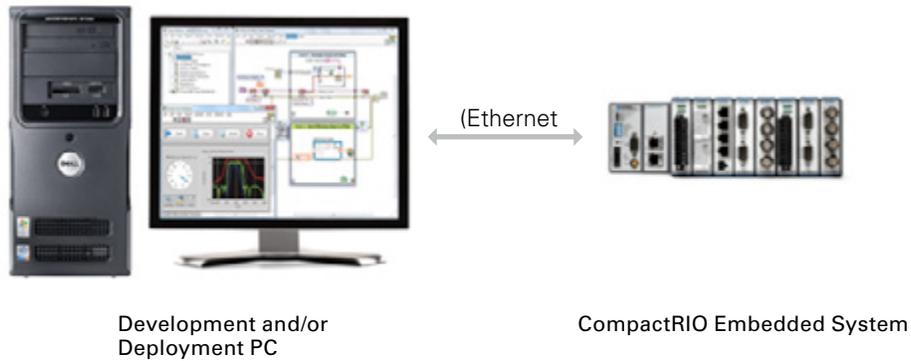


Figure 12.3. A Typical RIO Deployment System

Best Practices for Security on RIO Systems

The best practices for security on RIO systems are organized into three groups: recommended, optional, and extreme. The best practices presented online are not exhaustive; there are further measures you can take, but these are reserved for extremely advanced users. To learn about what you can do beyond the best practices presented in these online articles, contact National Instruments either through support or your local sales or account representative.

The recommended practices should be adopted by all users before deploying the RIO system. The optional practices take more time and effort to implement, but they provide the added security required by some applications. Lastly, the extreme section highlights a few measures that you can take to further secure a RIO system. Note that the extreme measures require a significant investment in time and effort and may not be appropriate for all applications.

Best Practices Resources

[Best Practices for Security on RIO Systems: Part 1 Recommended](#)

[Best Practices for Security on RIO Systems: Part 2 Optional](#)

[Best Practices for Security on RIO Systems: Part 3 Extreme](#)

CHAPTER 13

Using the LabVIEW for CompactRIO Sample Projects

LabVIEW 2012 and later provides several fully functioning project templates and sample projects to use as starting points for your own applications. All of the sample projects are built on existing LabVIEW project templates, like the Simple State Machine or the Queued Message Handler. You can spend less time developing your own application by using one of the sample projects that already has a lot of the application written for you. Using a sample project also ensures that the design your application is based on is an architecture recommended by National Instruments.

LabVIEW 2012 and later includes three sample projects that were written for CompactRIO and NI Single-Board RIO targets:

- LabVIEW FPGA Control on CompactRIO
- LabVIEW Real-Time Control on CompactRIO (NI Scan Engine)
- LabVIEW FPGA Waveform Acquisition and Logging
- LabVIEW 2013 and later includes an additional sample project designed for SCADA systems:
- LabVIEW Supervisory Control and Data Acquisition
- LabVIEW Real-Time Sequencer Sample Project

Overview of Sample Projects

This document examines the architecture of the LabVIEW FPGA Control on CompactRIO sample project, but the architectural drawings for the other sample projects are included below as a reference.

LabVIEW FPGA Control on CompactRIO Sample Project

The LabVIEW FPGA Control on CompactRIO sample project implements deterministic, hardware-based control of a plant. The control algorithm, which was written with the LabVIEW FPGA Module, runs on the FPGA inside the CompactRIO device. You send commands and setpoint changes to the FPGA from the user interface, running on a desktop computer, by way of the real-time controller in the device. This controller also monitors the status of the application, such as CPU load and memory usage. This sample project has the following features:

- **High-performance control**—The control loop can run faster than 10 kHz and features four control algorithms operating in parallel, all with minimal jitter.
- **Hardware-based control**—Running the control algorithm and safety logic on the FPGA provides maximum reliability.
- **User interface with headless option**—The user interface VI interacts with the CompactRIO device and displays data. This VI can connect and disconnect from the device at any time without affecting the control loop.
- **Error handling**—The application reports and logs all errors from the CompactRIO device and then shuts down on critical errors.

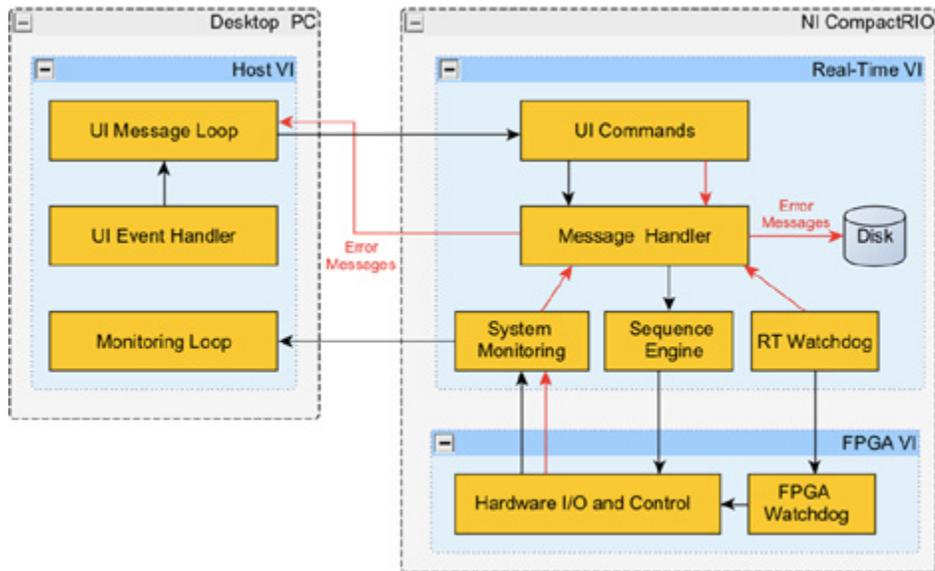


Figure 13.1. LabVIEW FPGA Control on CompactRIO Sample Project Architecture

LabVIEW Real-Time Sequencer on CompactRIO Sample Project

The LabVIEW Real-Time Sequencer on CompactRIO sample project is similar to the LabVIEW FPGA Control on CompactRIO sample project, but it implements a user-customizable control sequence to perform the control.

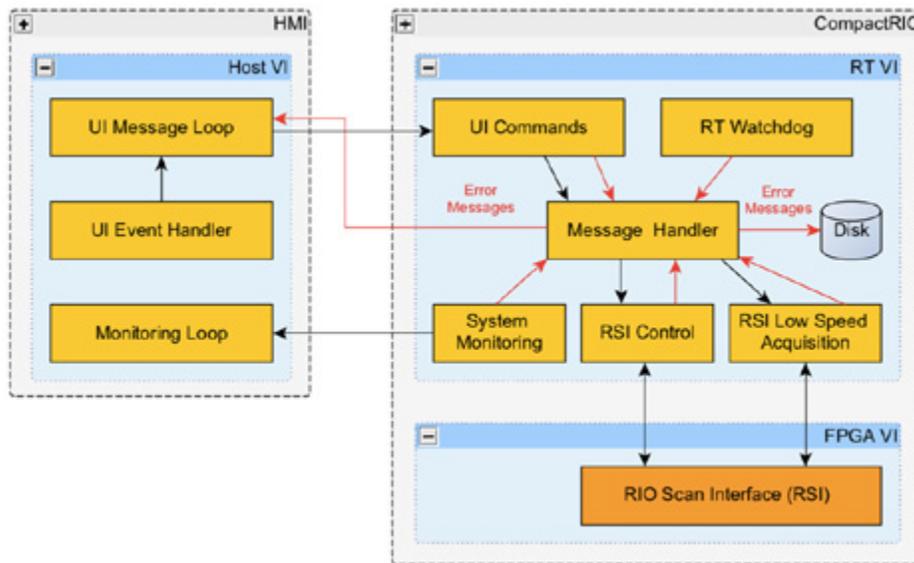


Figure 13.2. LabVIEW Real-Time Sequencer on CompactRIO Sample Project Architecture

LabVIEW Waveform Acquisition and Logging on CompactRIO Sample Project

The LabVIEW Waveform Acquisition and Logging on CompactRIO sample project acquires continuous waveform data from C Series I/O modules and logs it to disk on the real-time controller. This sample project has the following features:

- **Headless operation with optional user interface**—The user interface VI interacts with the CompactRIO device and displays data. This VI can connect and disconnect from the device at any time without affecting the acquisition and logging loop.

- **Triggered data logging**—The real-time VI logs acquired data to disk as TDMS files when a trigger condition is met. This sample project also manages the amount of disk space being used.
- **Error handling**—The application reports and logs all errors from the CompactRIO device, shutting down on any critical error.

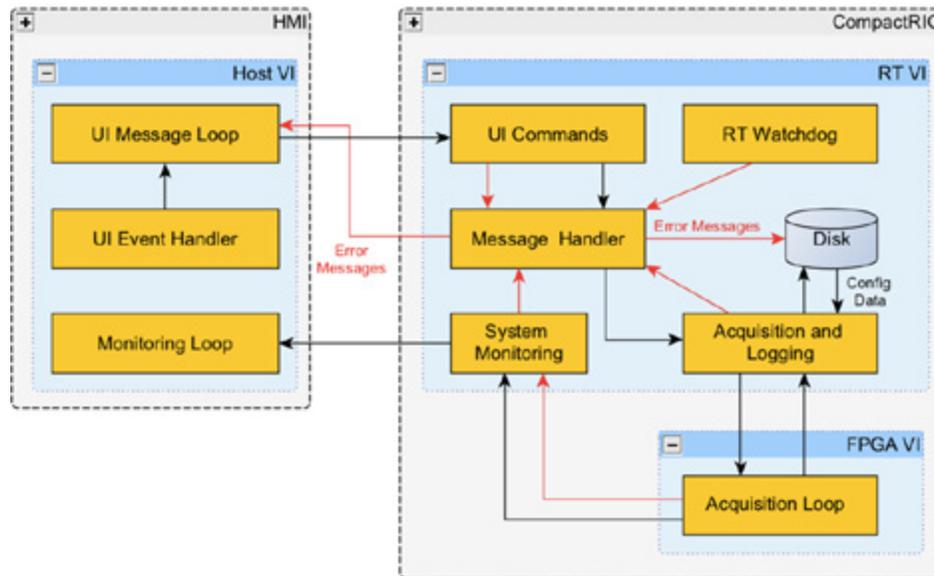


Figure 13.3. LabVIEW Waveform Acquisition and Logging Sample Project Architecture

LabVIEW Real-Time Control on CompactRIO (RIO Scan Interface) Sample Project

The LabVIEW Real-Time Control on CompactRIO (RIO Scan Interface) sample project implements deterministic, software-based control of a plant. This sample project uses the NI RIO Scan Interface, which is an alternative to programming with the LabVIEW FPGA Module, to perform I/O on the FPGA. The RIO Scan Interface programming mode is available on most CompactRIO and NI Single-Board RIO devices and is sufficient for applications that require single-point access to I/O at rates of a few hundred hertz. This sample project has the following features:

- **Deterministic, software-based control**—The real-time controller executes parallel control algorithms that run at different rates.
- **RIO Scan Interface**—The RIO Scan Interface provides single-point updates to I/O variables at rates up to a few hundred hertz without requiring FPGA programming.
- **User interface**—The user interface VI interacts with the CompactRIO device and displays data. This VI can disconnect from the device and reconnect at any time without affecting the control loop.
- **Error handling**—The application reports and logs all errors from the CompactRIO device and shuts down on critical errors.

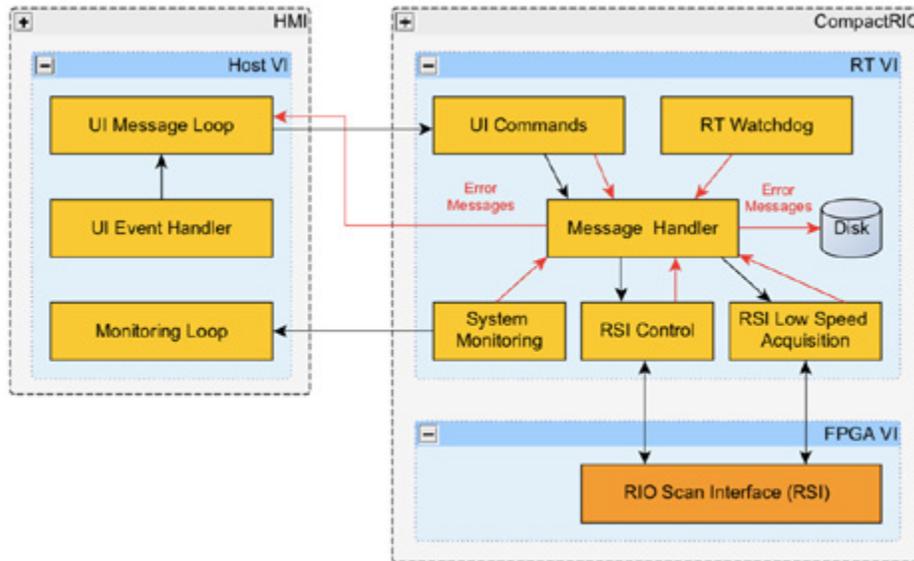


Figure 13.4. LabVIEW Real-Time Control (RIO Scan Interface) on CompactRIO Sample Project Architecture

LabVIEW Supervisory Control and Data Acquisition System Sample Project

The LabVIEW Supervisory Control and Data Acquisition System sample project demonstrates how to implement a supervisory control and data acquisition (SCADA) system with scalable architecture for building systems with many I/O points. This sample project has the following features:

- **SCADA server**—The SCADA server manages I/O points in the system, logs data and alarms in the historical database, and regularly archives data from the logging database to the archive database.
- **Administrator console**—The administrator console allows the system administrator to configure and manage the SCADA server.
- **Client**—The client allows the operators to monitor the system status and I/O points. The operator can view the values of the I/O points, alarms, and historical trends in different detailed views.
- **Simulated CompactRIO system and simulated PLC-based system**—A simulated CompactRIO system and a simulated PLC-based system are included in this sample project. The systems demonstrate how to connect to third-party devices. You can replace the simulated systems with a real CompactRIO system and a real PLC-based system.
- **Scalability**—You can use this sample project as a starting point to scale to a large system by adding a large number of I/O points or adding many subsystems in this sample project.

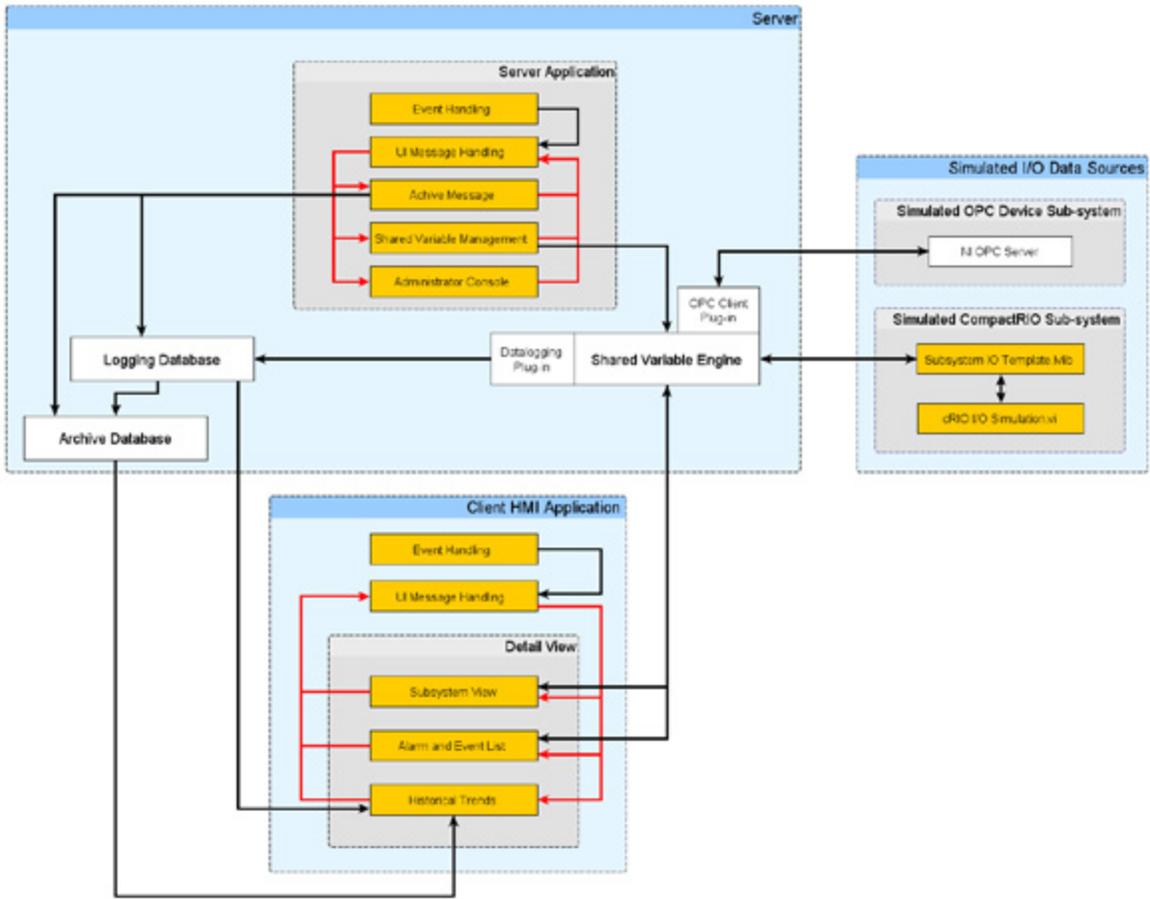


Figure 13.5. LabVIEW Supervisory Control and Data Acquisition Sample Project Architecture

Walk-Through of the LabVIEW FPGA Control on CompactRIO Sample Project

You can see from the communication diagram in Figure 13.1 that this sample project has VIs running on three different targets: the desktop PC, the CompactRIO real-time target, and the CompactRIO FPGA target. The FPGA target features two tasks running in parallel: one executes the control loop and one executes the Watchdog Loop. On the real-time target, tasks are receiving commands from the desktop, monitoring the health of the real-time system, handling messages from all the other tasks, and monitoring the Watchdog Loop. The desktop PC features tasks responsible for sending and receiving commands over the network to the real-time target, handling actions on the user interface, and updating the user interface with data from the real-time target.

Note: The LabVIEW FPGA Control on CompactRIO sample project works for all CompactRIO and NI Single-Board RIO hardware. You also can use this sample project with a real-time PXI controller and R Series device, but the System Reset node on the LabVIEW FPGA VI is not supported on R Series hardware. You can delete this node or put it inside a Disable Diagram structure.

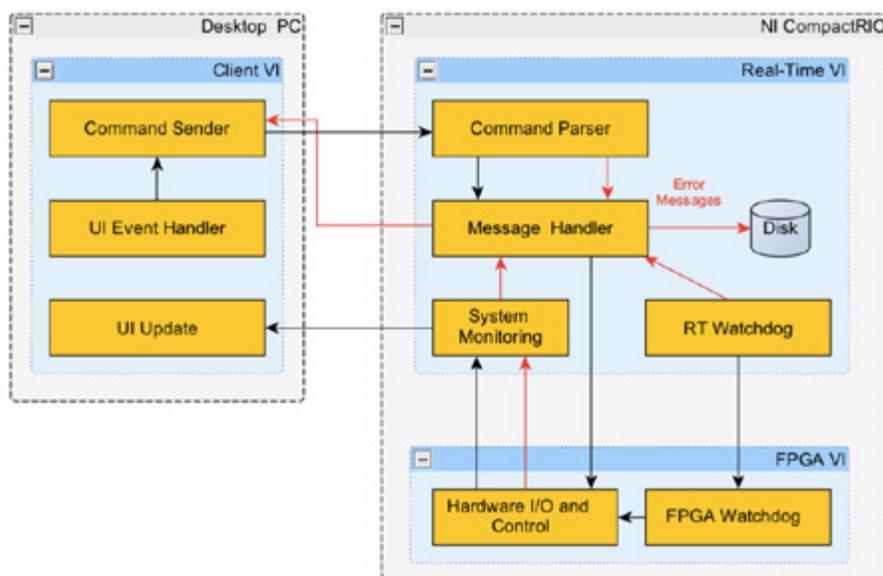


Figure 13.6. Data Communication Diagram for the LabVIEW FPGA Control on CompactRIO Sample Project

For an introduction to creating data communication diagrams for CompactRIO applications, see [Chapter 1: Designing a CompactRIO Software Architecture](#).

Create a New Sample Project

To create your own application based on the LabVIEW FPGA Control on CompactRIO sample project, click the Create Project button in the Getting Started Window. This launches the Create Project dialog.

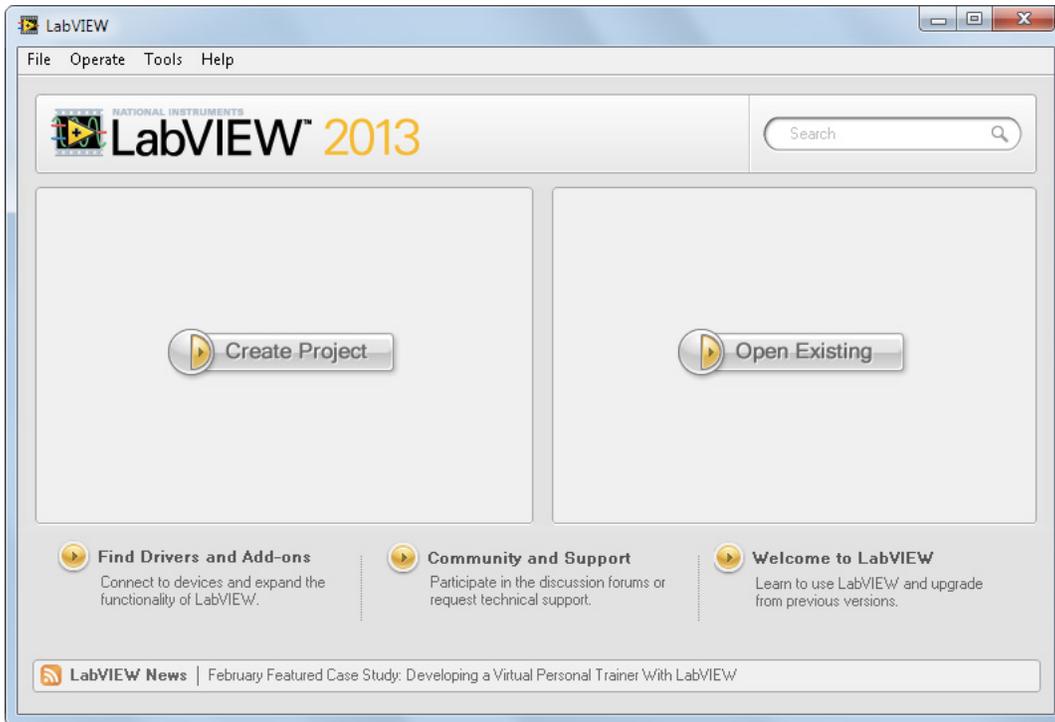


Figure 13.7. Create a sample project by clicking the Create Project button.

If you choose the CompactRIO filter under the Sample Projects heading, you can see the LabVIEW FPGA Control on CompactRIO sample project. Follow the More Information link to launch the documentation for this sample project.

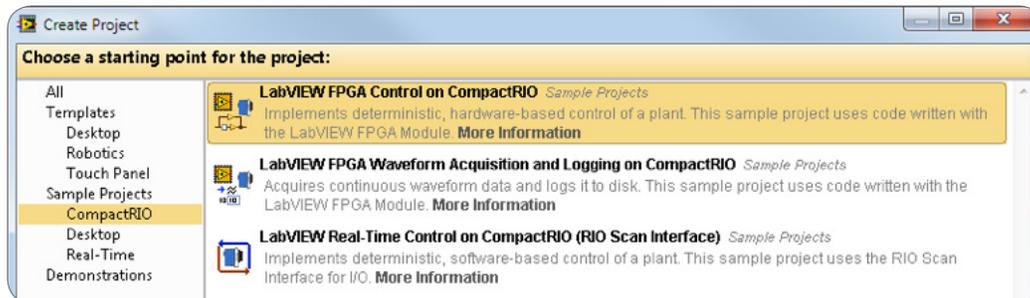


Figure 13.8. Filter Sample Projects by CompactRIO

Before opening the documentation, start the process of creating your own copy of this project by clicking Next in the Create Project dialog.

You can see several options for customizing your own copy of the LabVIEW FPGA Control on CompactRIO sample project. In this case, change "Project Name" to "My Control Application." Specify a new folder titled My Control Application within your preferred directory. Leave all the other options as default. Click the Finish button to start copying the project. While the project is being made, examine the documentation.

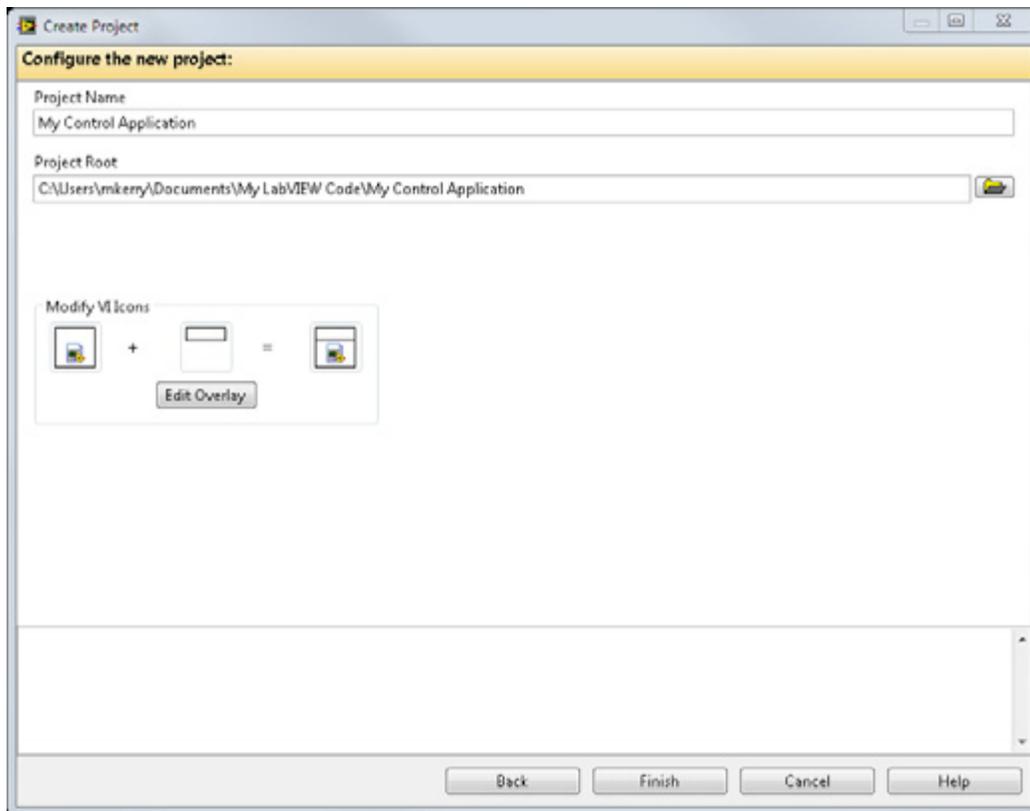


Figure 13.9. Configure your sample project.

Review Documentation

You can see in the overview of this documentation that the sample project is based on the Simple State Machine and Queued Message Handler project templates. You can also see a list of features, including the high-performance and hardware-based control of this application, because the control loop is running on the FPGA target.

If you scroll down, you can see the system requirements for this sample project, the required software and hardware, and an overview of all the parallel tasks running across the execution targets in this application.

Further down, a diagram illustrates what happens when you change a control setpoint on the user interface of this application. You can see that a message is sent over the network to the real-time target containing the new setpoints, and the real-time target sends those setpoints via read/write controls to the FPGA target. Explore how this works later when you open up the VIs.

Sending PID Setpoints to the FPGA

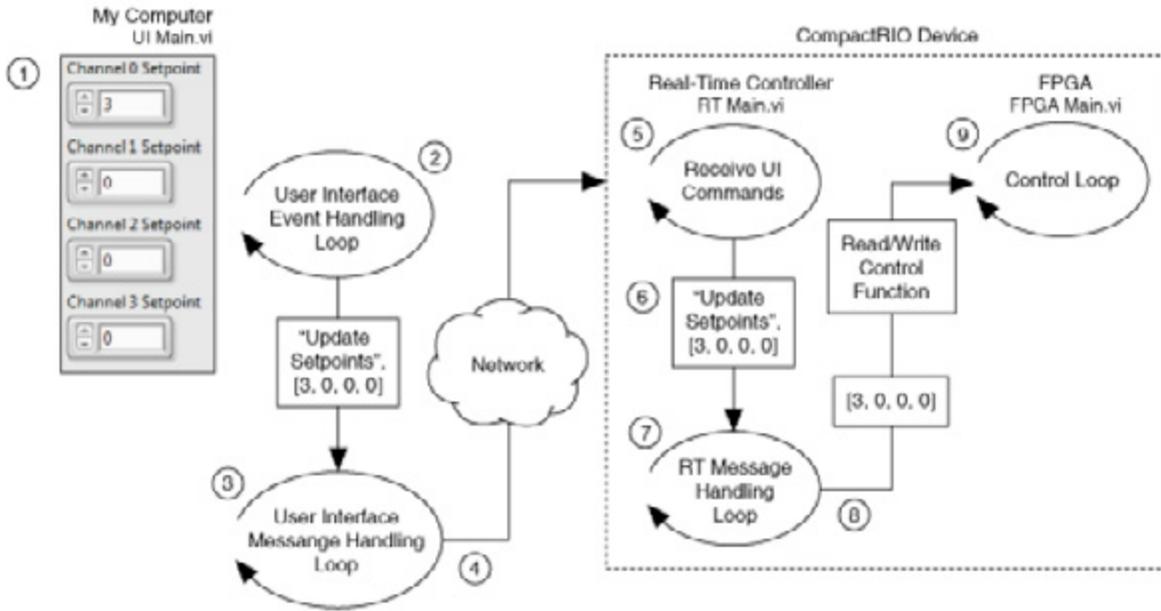


Figure 13.10. Review sample project documentation.

The documentation also contains instructions on how to modify the sample project to include your own hardware under “Modifying this Sample Project—Adapting the Sample Project to Your Hardware.”

Now look at the completed project.

Review LabVIEW FPGA Control on CompactRIO Architecture

You can see you now have your own copy of the LabVIEW FPGA Control on CompactRIO application called My Control Application. The project features three execution targets: My Computer, the real-time CompactRIO target, and the FPGA target. For more documentation about the project, the documentation discussed previously is included in the project under the Project Documentation folder.

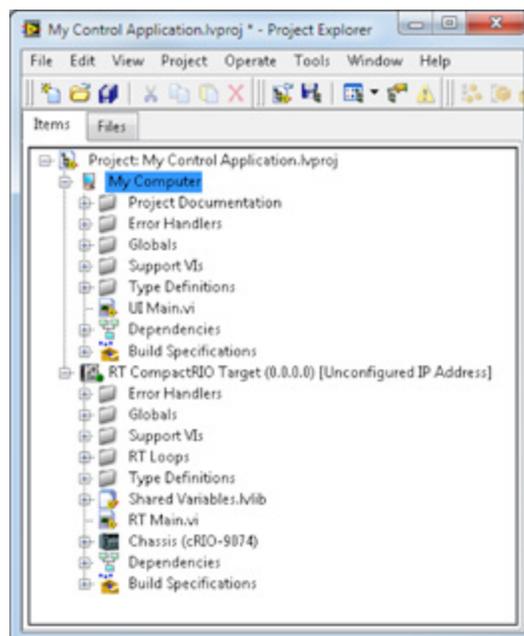


Figure 13.11. LabVIEW FPGA Control Sample Project

FPGA Main VI

The front panel of the FPGA Main VI features many controls and indicators. The Real-Time Main VI uses read/write controls to set and get the values of these controls and indicators. Now look at the block diagram.

Note: This FPGA VI was written to be loaded to flash memory using the NI RIO Device Setup utility and configured to reload the FPGA bitfile only upon cycling the power. After reviewing the LabVIEW FPGA code, see instructions for properly configuring the FPGA bitfile in the section "Downloading the FPGA Bitfile to Flash Memory."

Control Loop

Right away, you see multiple blue labels indicating places where you are required to add your own code for your specific application. You can see the control loop is a state machine that is initialized in the safe state. The only way for the control loop to move out of the safe state is by receiving a command from the Real-Time Main VI. While in the safe state, a safe state value is sent to all of the output controls for your control algorithm.

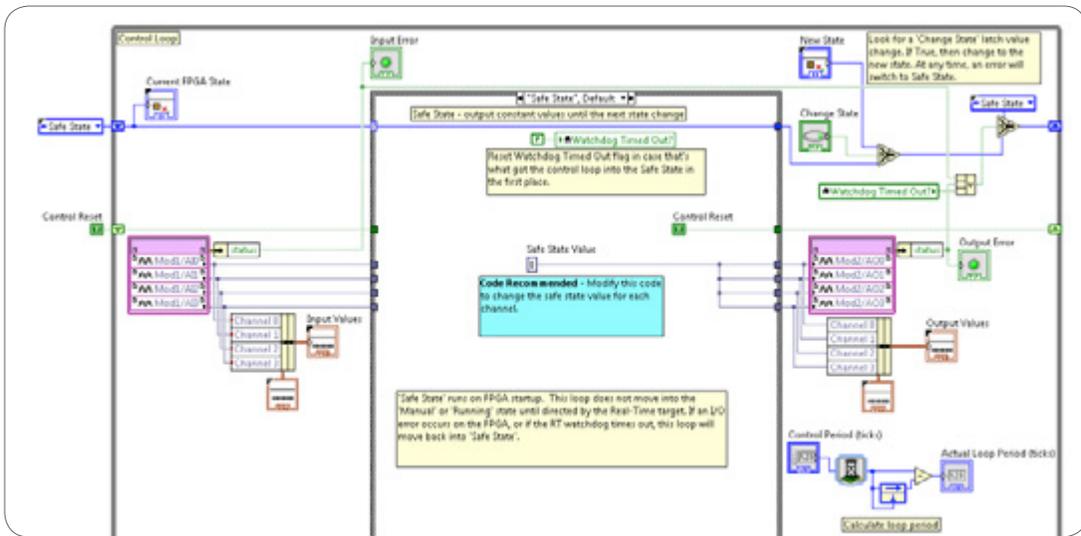


Figure 13.12. The FPGA Main VI starts up in a safe state.

If the user moves the control algorithm into the Manual state on the user interface, then the Manual state runs. In this state, users can specify which manual values they wish to write to each of the output channels.

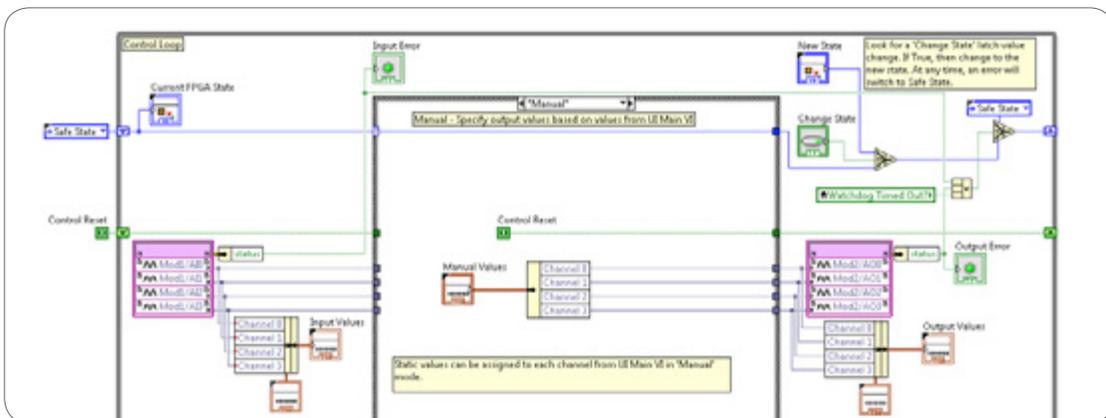


Figure 13.13. Move into Manual state to manually control the hardware outputs.

If users run this application in the Control state (see Figure 13.14), then they are required to add a function, like the PID block that is available with the LabVIEW PID and Fuzzy Logic Toolkit, to actually perform the control algorithm because this code has no control function. The PID block installs with the LabVIEW Real-Time Module, and can be easily dropped into this VI. You can also create your own custom control algorithm.

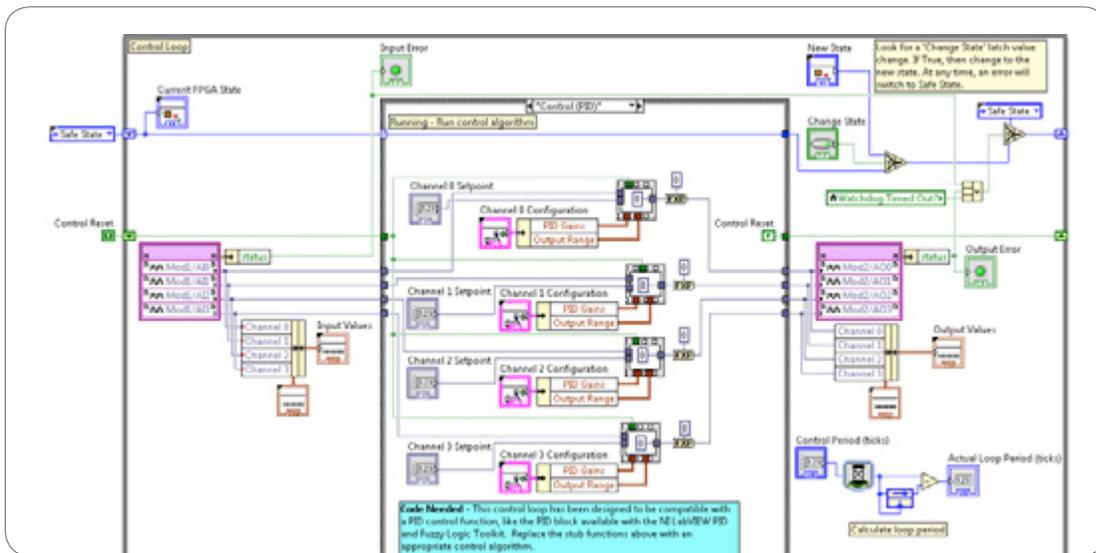


Figure 13.14. Move into the Control state to execute PID control on all four channels.

Watchdog Loop

In addition to the control loop, you can use the Watchdog Loop. This loop, once enabled by the RT Main VI, periodically “pets” the watchdog and resets the system if a timeout occurs. The timeout occurs when the real-time Watchdog Loop does not execute within a timeout period defined by the user. The Watchdog Loop runs on the FPGA VI instead of the real-time target because this code allows the FPGA to switch into a safe state and write known values to the outputs before resetting the system. For more information on watchdog timers, see the “Ensuring Reliability With Watchdog Timers” section of [Chapter 3: Designing a LabVIEW Real-Time Application](#).

More information on configuring the watchdog timer is included in the real-time Watchdog Loop discussion later in the chapter.

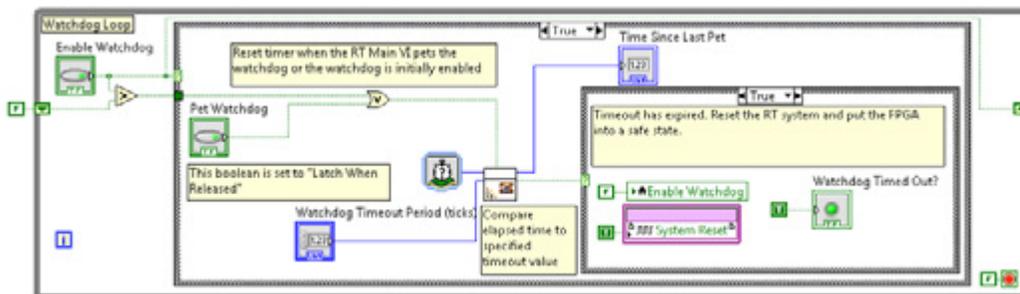


Figure 13.15. The watchdog timer is implemented in LabVIEW FPGA.

Note: If the FPGA bitfile is loaded to flash and set to reload only upon cycling the power, then when you call the System Reset node, the real-time controller reboots, but the FPGA bitfile continues to run. This ensures that the FPGA hardware outputs immediately go into a safe state and stay there until the real-time controller is back online. After reviewing the LabVIEW FPGA code, see instructions for properly configuring the FPGA bitfile in the section “Downloading the FPGA Bitfile to Flash Memory.”

If you choose to not download the FPGA bitfile to flash memory, or if you choose to download it to flash but with the option to “autoload upon a device reboot,” then you need to add some logic to this TRUE case to ensure that the hardware outputs go into a safe state prior to calling the System Reset node. Otherwise, a race condition could occur and you could end up resetting the FPGA prior to putting the hardware outputs into a safe state.

RT Main VI

Now look at the RT Main VI, which features several loops running in parallel. A comment on the diagram explains the functionality of each of the parallel loops. This VI opens with a broken run arrow since the FPGA VI is not compiled.

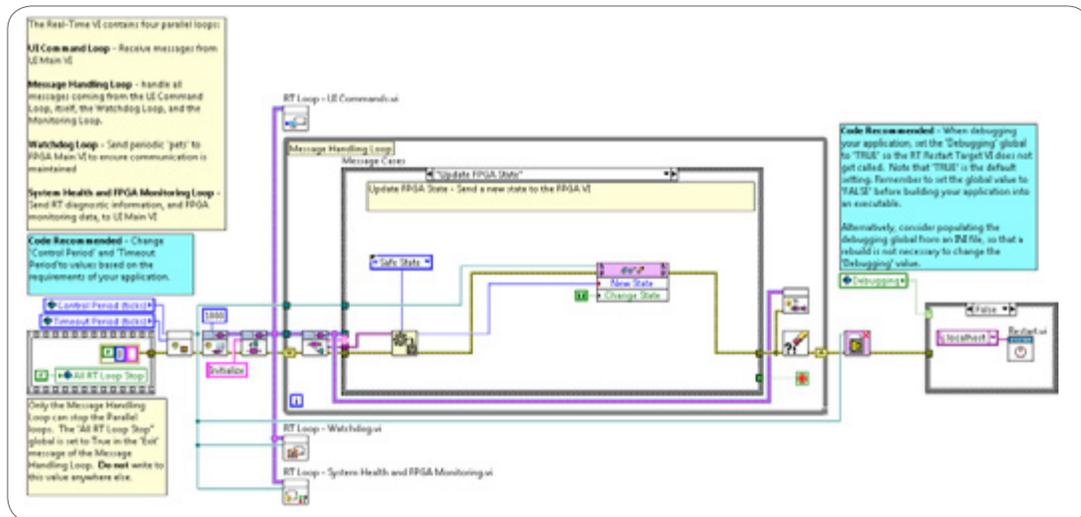


Figure 13.16. The RT Main VI features four loops running in parallel.

UI Commands Loop

The top loop is the UI Commands VI. Open this VI.



Figure 13.17. The UI Commands Loop

This VI is based on the Simple State Machine project template that is available with LabVIEW. You can see that this VI is using Network Streams to communicate with the user interface VI under My Computer. The main purpose of this loop is to receive UI commands and manage the connection of any Network Streams that are used within the application. This sample project uses two streams:

- **UI Writer Stream**—RT application receives commands from the UI
- **RT Writer Stream**—RT application sends error messages to the UI for display

Network Streams are used for this type of data transfer because when sending commands from a user interface to an embedded system, you need a reliable data communication mechanism. Network Streams were designed and optimized for lossless, high-throughput data communication. They feature enhanced connection management that automatically restores network connectivity if a disconnection occurs due to a network outage or other system failure. Streams use a buffered, lossless communication strategy that ensures data written to the stream is never lost, even in environments that have intermittent network connectivity.

The VI initializes by going into the Listen for UI Writer state and waits until it makes a connection with the UI, at which point it moves into the Receive UI Commands state.

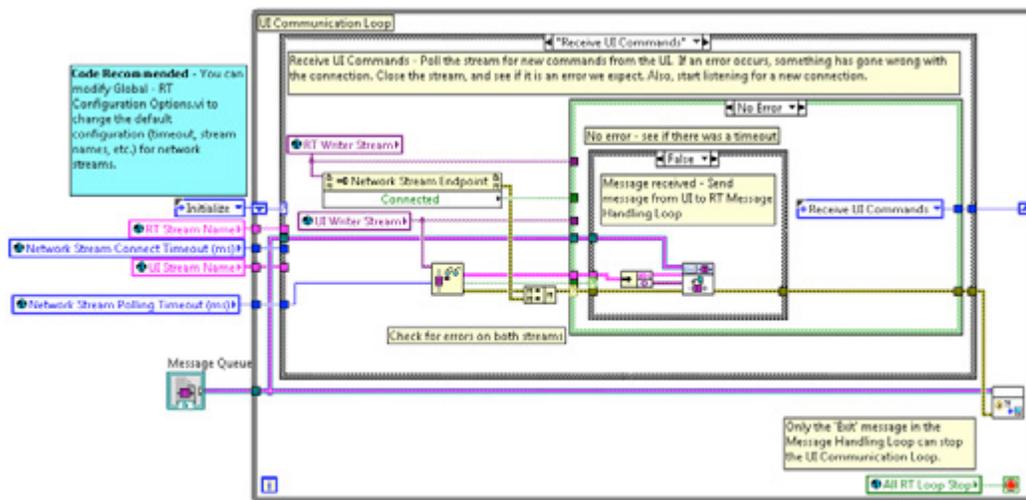


Figure 13.18. The UI Commands loop receives UI commands and manages the network connection of Network Streams.

Upon receiving an error in both the Listen for UI Writer and Receive UI Commands cases, destroy the Network Stream and attempt to create a new one. This allows the client to disconnect and reconnect at any time while the RT application continues to run. When the client disconnects, the reference is no longer valid, so you need to create a new one.

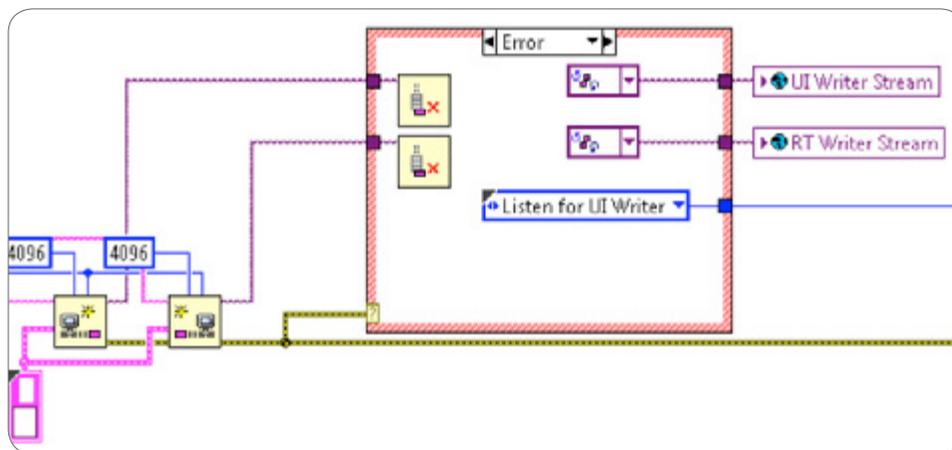


Figure 13.19. The UI Commands Loop is designed to allow the client to disconnect and reconnect at anytime.

You write data received from the Network Stream into the Enqueue Message VI shown in Figure 13.20. The Enqueue Message VI passes all messages or commands to the Message Handling Loop shown in Figure 13.21. This VI is borrowed from the Queued Message Handler Template available for the LabVIEW Core software option. The Message Queue API is basically a wrapper around Queue functions with usability tweaks for communicating messages between two or more loops. You can double-click the Enqueue Message VI to see how it works. This function forces the user to send messages as a cluster consisting of a string (the message or command) and a variant (any associated data). For example, the command might be "Change Setpoint" and the associated data might be "27 °C." This is a scalable and efficient method for communicating messages within a large application. You use the variant data type so that you can use the same API to send multiple data types.

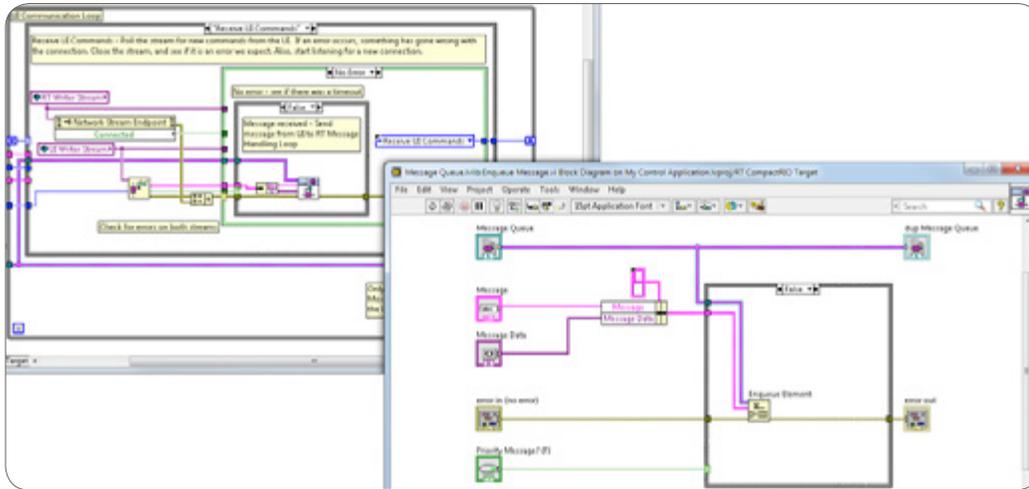


Figure 13.20. The Message Queue API is a wrapper around Queue functions.

The Message Handling Loop is part of the RT Main VI. The RT Main VI was built from the Queued Message Handler project template and thus uses a Queued Message Handler architecture for receiving and sending messages. The RT Main VI is set up to handle different messages like updating the FPGA state on the FPGA, handling errors, and sending data to the UI.

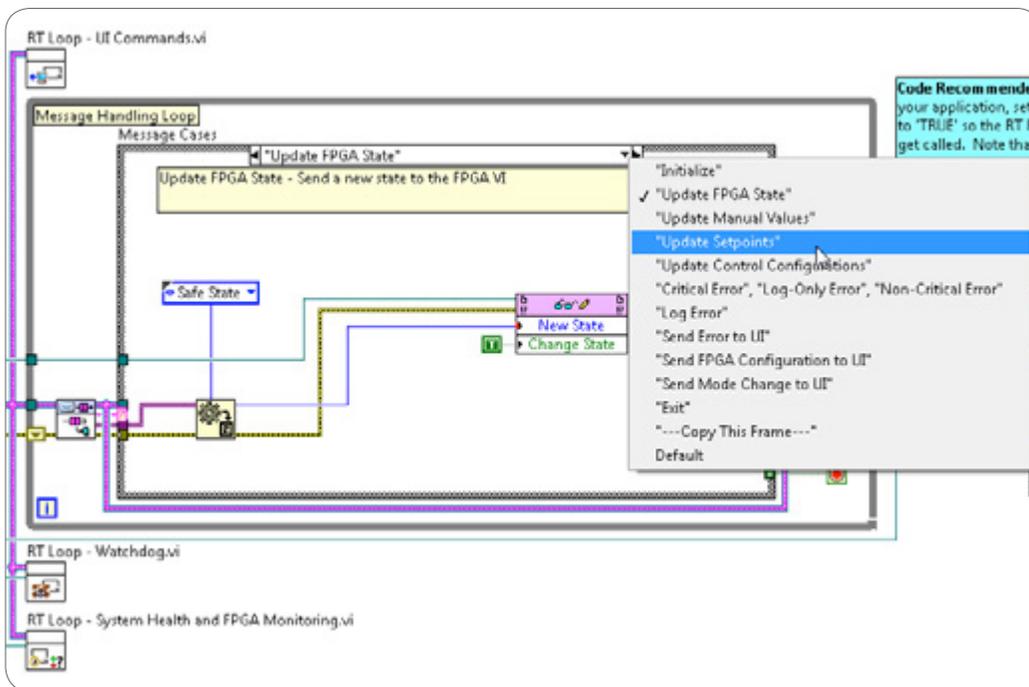


Figure 13.21. The Message Handling Loop receives messages such as Update Setpoints and Send Error to UI.

Note that every parallel loop on the RT Main VI has its own unique error handler VI, shown in Figure 13.22.

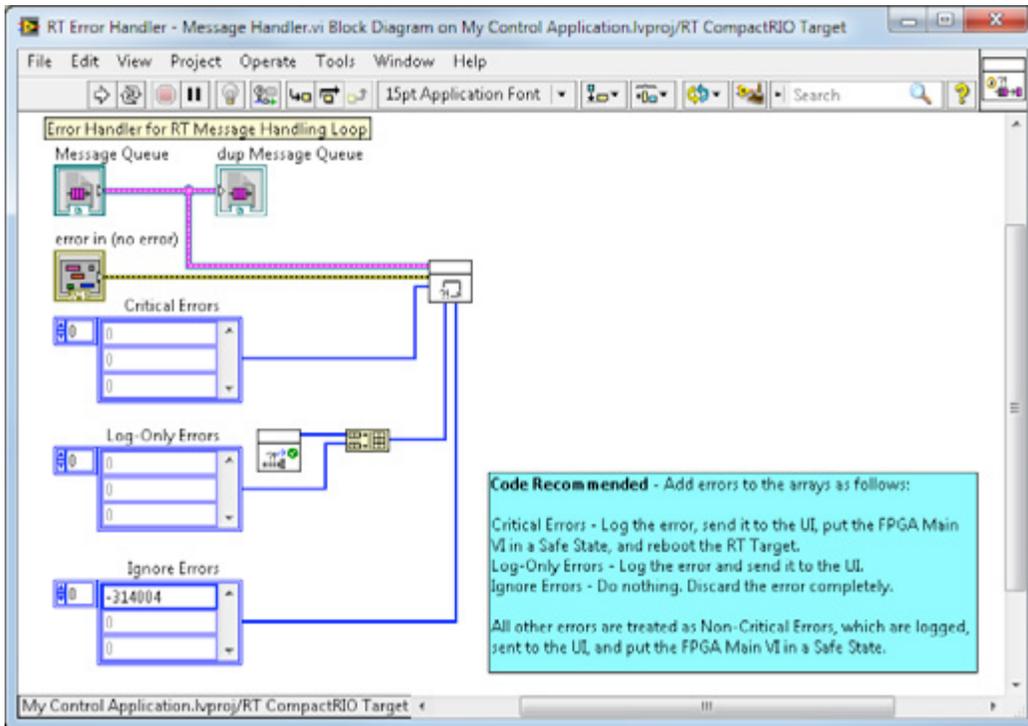


Figure 13.22. Each loop has its own error handler subVI.

By using its own error handler, each loop can specify unique errors that the particular loop wants to handle in a certain way. Any error message that is generated within the real-time application can be classified by the user as critical, log-only, and ignore. Any error that isn't classified by the user is automatically classified as non-critical. All error messages are sent to the Message Handling Loop and are processed within the "Critical Error," "Log-Only Error," "Non-Critical Error" case, which is shown in Figure 13.23. Note that a feature of the Queued Message Handler is that in addition to receiving messages from the producer loop (UI Command Loop), the consumer loop can send messages to itself and make decisions on which state executes next.

In Figure 13.23, all error messages are logged to the local drive on the real-time controller and sent to the UI for display. In addition, upon a non-critical error, the FPGA goes into a safe state. Send a message to the UI notifying the user that the FPGA state has changed. Upon a critical error, exit the application.

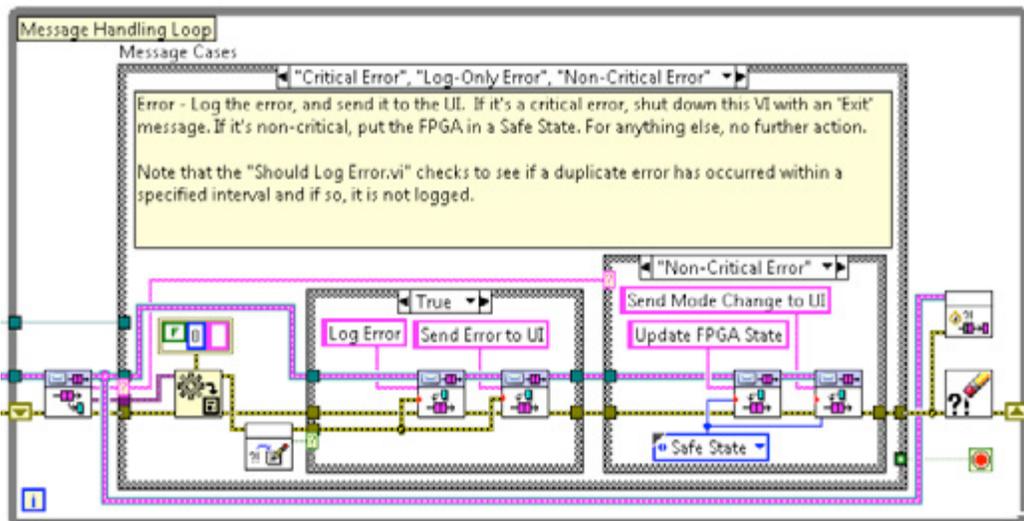


Figure 13.23. All error messages are processed in the Message Handling Loop.

Watchdog Loop

The next VI on the RT Main VI is the Watchdog Loop.



Figure 13.24. Watchdog Loop

The only purpose of this VI is to use a read/write control to periodically pet the watchdog in the Watchdog Loop on the FPGA that you already looked at.

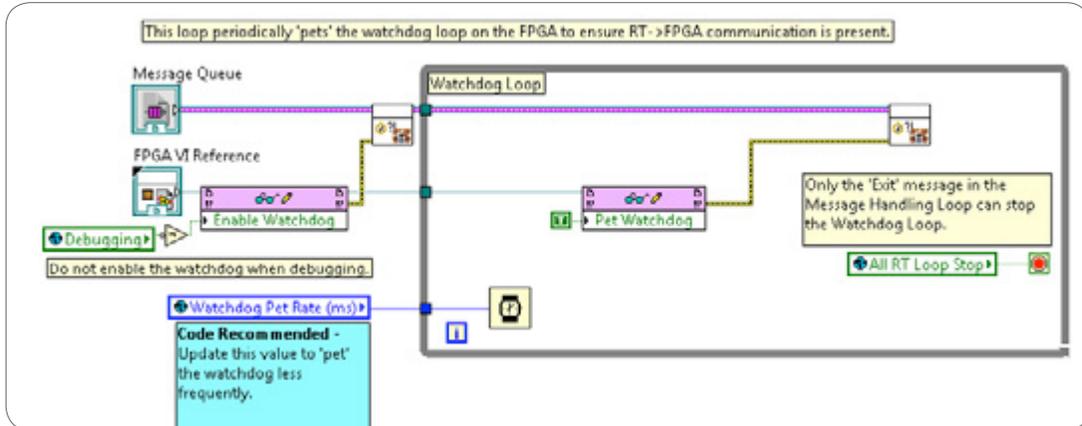


Figure 13.25. The RT Watchdog Loop periodically pets the watchdog loop on the FPGA to ensure that RT to FPGA communication is present.

Several global variables are referenced throughout the RT Main VI. The RT Main VI includes one global variable that stores all configuration options titled Global-Configuration Options.vi. This global variable is located in the LabVIEW project under the **RT Target»Globals** folder and is shown in Figure 13.26. This is where users can specify whether they are in debugging mode, in addition to timeouts and loop rates.

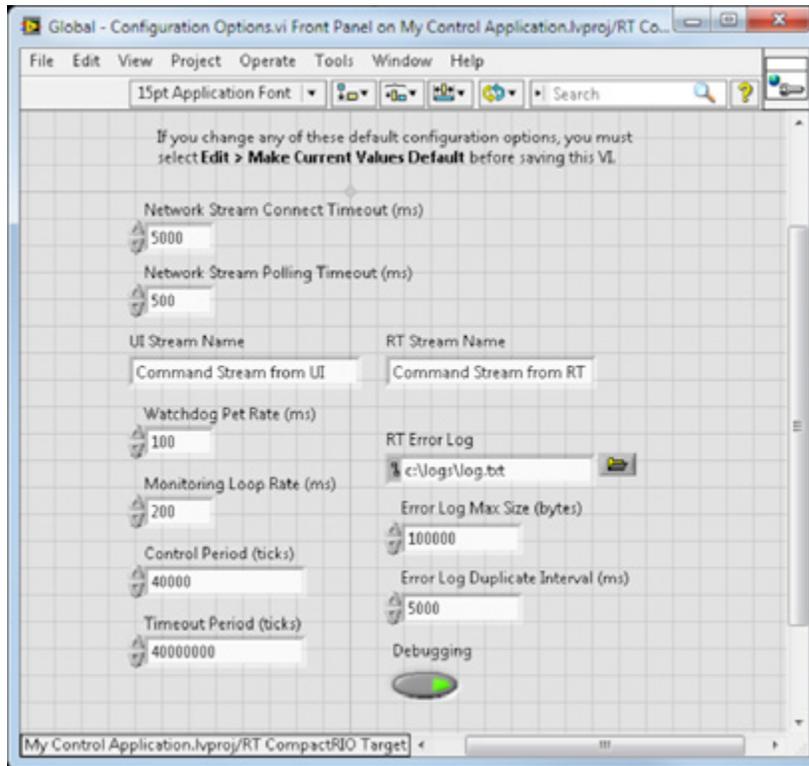


Figure 13.26. Configuration options used within the RT Main VI are stored as a global variable.

The watchdog timeout by default is set to 40,000,000 ticks of the FPGA clock or 1 second. This means that if the Watchdog Loop fails to execute within 1 second on a given iteration, a watchdog timeout occurs and the FPGA goes into a safe state. Keep in mind that a real-time OS is extremely reliable and the only reason a loop might not finish on time is typically because of CPU starvation—your system runs out of memory and the software is hanging or is about to crash. You can ensure that the CPU bandwidth stays low and that the largest block of contiguous memory does not decrease by monitoring them in the System Health and FPGA Monitoring Loop.

You should disable the watchdog timer in debugging mode since debugging tools such as highlight execution cause the application to run much slower.

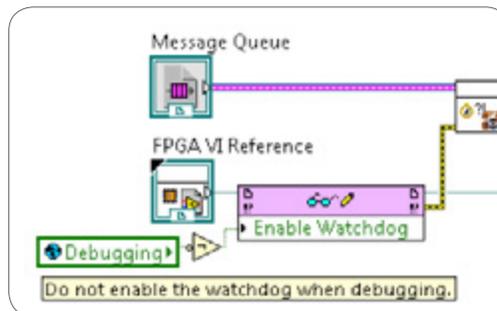


Figure 13.27. The Watchdog Loop checks to see if debugging mode is enabled before executing the watchdog.

System Health and FPGA Monitoring Loop

The last loop in RT Main VI is the System Health and FPGA Monitoring Loop.



Figure 13.28. System Health and FPGA Monitoring Loop

This loop uses network-published shared variables to send monitoring information to the UI Main VI. You monitor input values (analog input) and output values (analog output) from the FPGA VI by using read/write controls and monitor some real-time system data, such as largest contiguous memory block and CPU usage percentage.

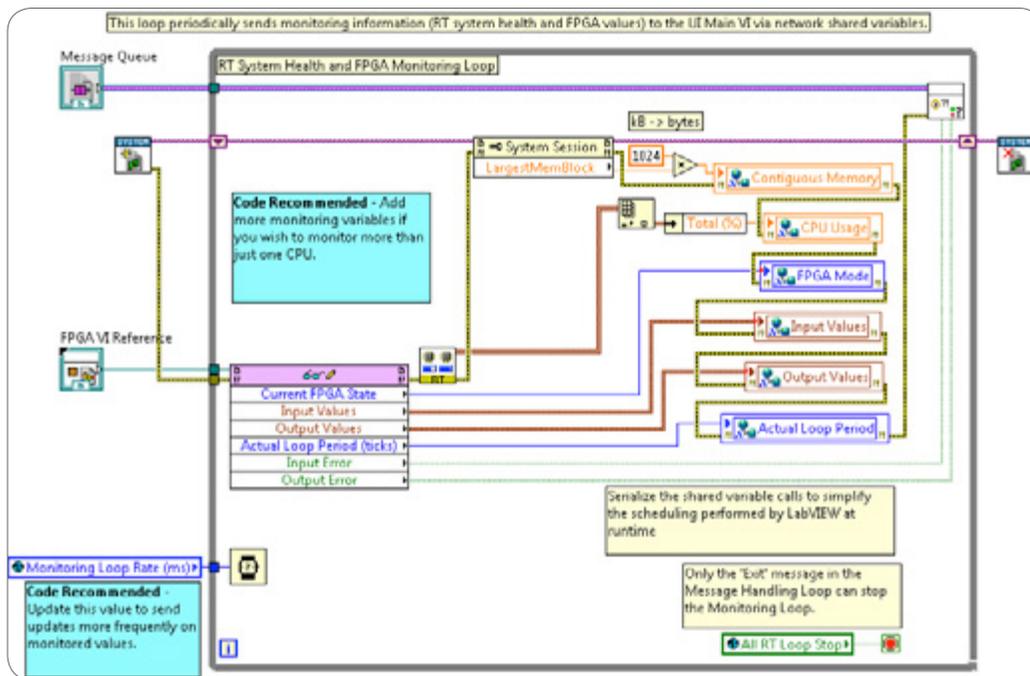


Figure 13.29. The System Health and FPGA Monitoring Loop periodically sends information to the user interface.

You need to keep the CPU usage percentage below 80 percent. If the CPU usage reaches 100 percent, you probably have loops that are running slower than your specified rate. If a high-priority Timed Loop is active, the other loops may not be executing at all. If you find that the CPU usage is higher than 80 percent, you should review the LabVIEW Real-Time Help document [Minimize CPU Usage](#) for tips and tricks on how to reduce it.

You also should closely monitor the largest block of contiguous memory. The largest block of contiguous memory should be fairly constant because LabVIEW allocates and releases memory throughout the execution of the RT Main VI. If you see that the contiguous memory falls below a certain limit, go into a safe state and restart the system. When you restart the real-time controller, the memory is defragmented. Often an embedded software application crashes not because it ran out of memory but because the available memory is very fragmented. This is caused by dynamic memory allocations. Some applications cannot be restarted while they are deployed out in the field, and for these applications, the developer should spend extra time removing a majority if not all of the dynamic memory allocations. Some common sources of dynamic memory allocation include Queues without a fixed-sized buffer, variable-sized arrays (and waveforms), variable-sized strings, and variants.

Finally, note that any input or output errors from the LabVIEW FPGA VI are also handled in this loop. If an error occurs on either the analog input node or the analog output node in the LabVIEW FPGA VI, a custom error message is sent to the UI for display.

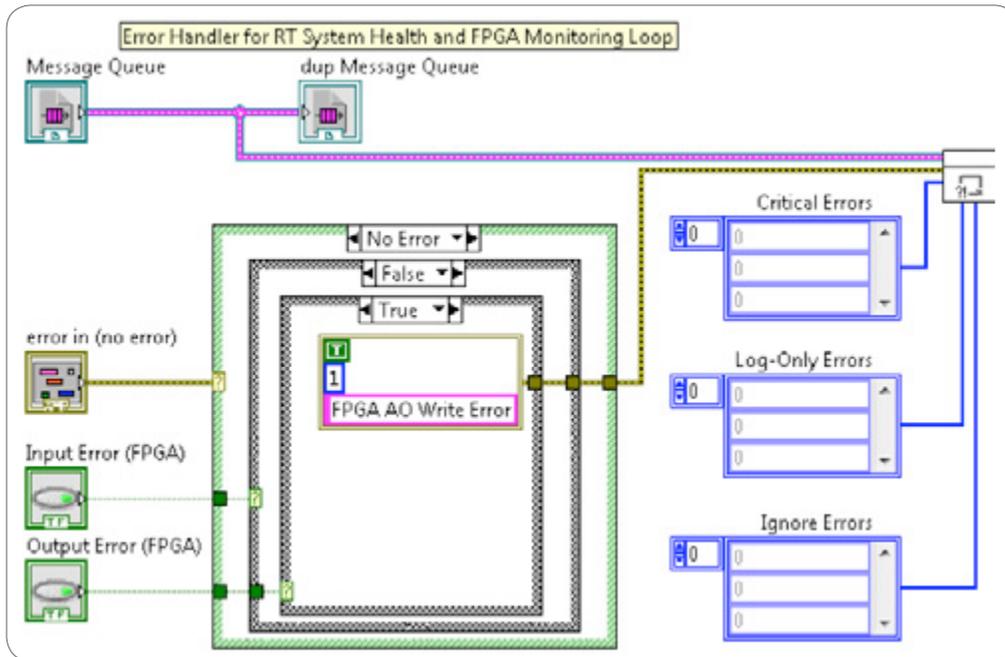


Figure 13.30. A custom error message is sent to the UI if an error occurs on the LabVIEW FPGA I/O nodes.

Stopping the Loops in the RT Main VI

The only way to stop the RT Main VI from running is to execute the Exit case within the RT Message Handling Loop. This case executes only if the user presses the Exit button on the UI Main VI, or if a critical error occurs within the RT Main VI. If either of these conditions occurs, the Exit case executes, which writes TRUE to the All RT Loop Stop global variable.



Figure 13.31. The RT Main VI stops if the user presses the Exit button on UI Main VI or if a critical error occurs within the RT Main VI.

UI Main VI

Open the UI Main VI. This is the user interface that your user interacts with when operating the LabVIEW FPGA Control on CompactRIO application. In the upper-left corner of the user interface, you see the Connection Settings. When this application first runs, these are the only settings that the user can change. All other controls are disabled.

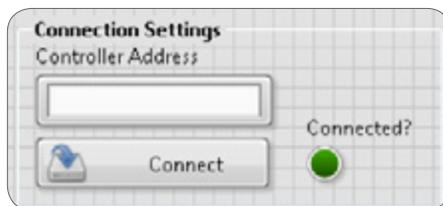


Figure 13.32. Connect to a CompactRIO device within the Connection Settings portion of the user interface.

The user can now change the IP address of the controller and choose to connect to that IP address. Once connected, the rest of the user interface enables.

If you want to run the FPGA control algorithm in the Control state, you can click the Run Control button.

Once the control is running, you can set setpoint values, which are sent to the target automatically. Note that the setpoints automatically coerce to the range of the input module being used for the PID control task.

You typically want to tweak the Control Configuration settings while the control loop is running. To send the updated Control Configuration settings to the target, you can click the Update Control Configurations button.

On the Manual tab, you can set the values of all the FPGA outputs directly. When you click the Run Manual button, the FPGA Main VI switches into the Manual state.

The Data Monitoring tab features the raw channel values of the inputs and the outputs on the FPGA.

Finally, the System Monitoring tab shows monitoring information like CPU usage, largest continuous memory, and the loop period of the VI running on the FPGA. If any errors occur while running this application, they appear in the System Status string on the front panel.

Now look at the block diagram. The UI Main VI has three parallel loops.

Event Handling Loop

The top loop is the Event Handling Loop, which sends messages to the UI Message Loop when any value change occurs on the front panel. By giving the Event Handling task its own loop separate from the UI Message Loop, you can ensure that the user interface is responsive.

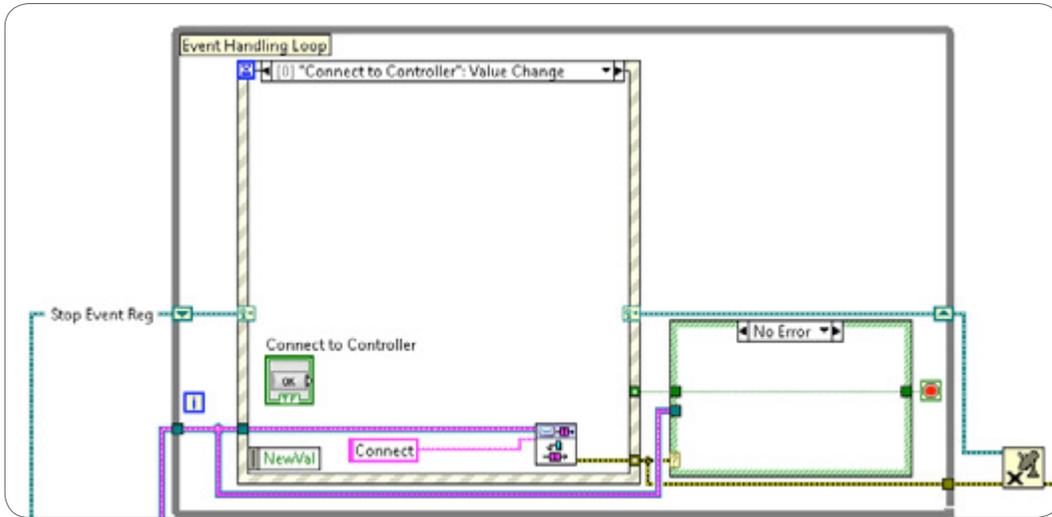


Figure 13.33. Event Handling Loop

Note that this loop is registered for the user event Stop. You use the user event so the Message Handling Loop is in charge of shutting everything down. If an error (as opposed to a UI event) is what causes the UI Main VI to exit, you need a way for the Message Handling Loop to tell the Event Handling Loop to stop. The user event is the NI-recommended way.

When designing applications involving a UI, ensure that the loop handling front panel events is the last one to shut down. In this sample project, when you press the Exit button, you immediately send an Exit command down to the UI Message Loop, as shown in Figure 13.34. Once the Event Handling Loop has received an acknowledgement that the UI Message Loop has stopped (or is executing the Exit case), stop the Event Handling Loop. You send this acknowledgment by sending a user event called Stop from the UI Message Loop to the Event Handling Loop.

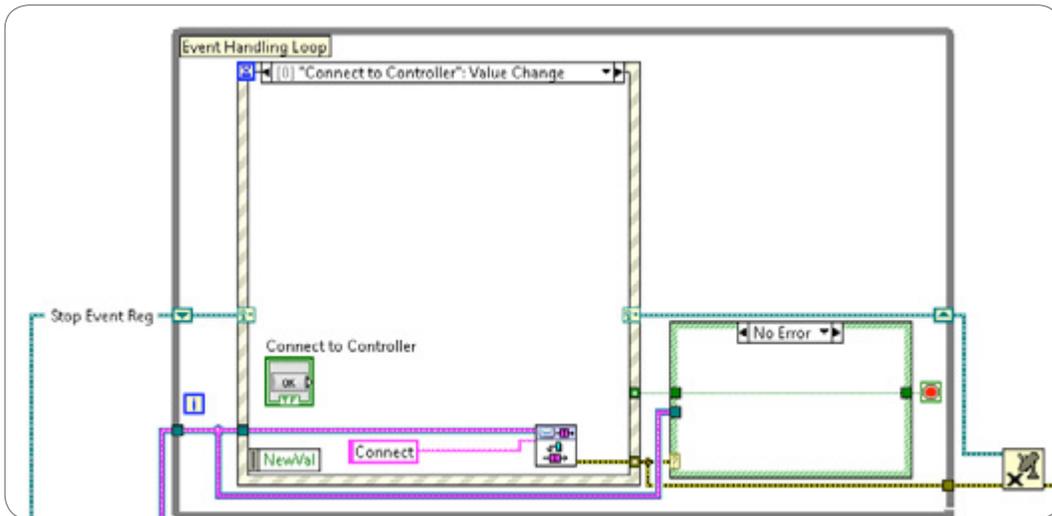


Figure 13.34. The Event Handling Loop sends messages to the UI Message Loop that are generated from front panel events.

UI Message Loop

The UI Message Loop, which is based on the Queued Message Handler project template, handles messages from the user interface and any error messages received from the real-time target via Network Streams.

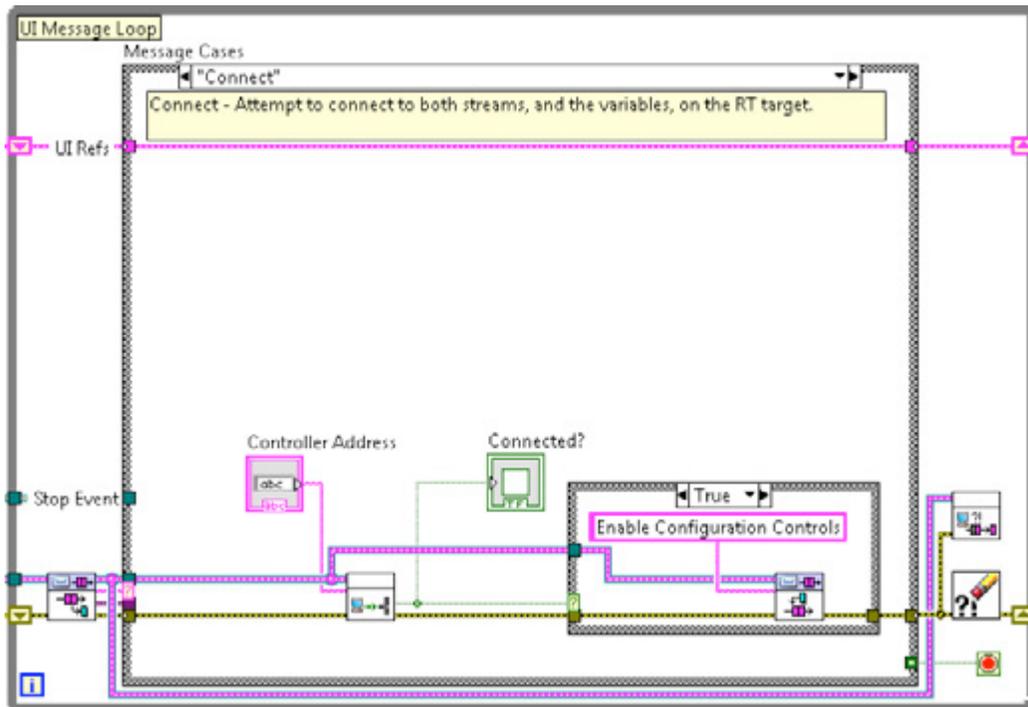


Figure 13.35. The UI Message Loop sends messages to the CompactRIO device using Network Streams and handles the configuration of front panel controls.

Specifically, this loop implements the following tasks:

- "Initialize UI Refs," "Disable Controls," "Enable Configuration Controls"
 - Enables and disables front panel controls based on the connection status.
- "Connect"
 - Creates a Network Stream endpoint for each Network Stream.
 - Initializes network-published shared variables.
 - Updates the Connected? indicator if both the Network Streams and network-published shared variables are successfully created.
- "Close Connections"
 - Closes network connections upon an error in the UI Message Loop and/or the UI Monitoring Loop.
- "Disconnect UI"
 - Disconnects the UI after closing network connections (if something went wrong). User needs to attempt to reconnect and address any networking issues.
- "Get Settings from FPGA"

- Once the connection is established within the “Connect” case, a command is sent to the RT Main VI to retrieve all current FPGA Main VI parameter values and send them back to the UI. This is important since the FPGA Main VI control loop continues to run even when the client disconnects and reconnects.
- “Initial UI Values”
 - Initializes the UI with the current values of the FPGA Main VI’s read/write controls upon successful connection to the real-time target.
- “Run Control,” “Run Manual,” “Run Safe State,” “Stop Control,” “Stop Manual”
 - Sends any of the commands listed above to the RT Main VI, which then sends the commands to the FPGA Main VI.
- “Update Control Configuration”
 - Sends the control configuration on the user interface to the FPGA Main Control Loop when the user hits the “Update Control Configuration” button.
- “Update Manual Values,” “Update Setpoints”
 - When the user modifies the value of any setpoint control or manual value control on the front panel, the new value is sent to the FPGA control loop.
- “Update Display Mode”
 - Enables/disables controls depending on the state of the FPGA (this case is called when the FPGA state changes).
- “Critical Error,” “Log-Only Error,” “Non-Critical Error,” “RT Error”
 - Identifies any incoming errors as “RT” or “UI” and displays the error message to the System Status indicator on the front panel.
 - Exits upon a critical error.
- “Update System Status”
 - Adds a new string to the beginning of the System Status string on the UI.
- “Exit”
 - Fires the user event to stop the Event Handling Loop and toggles the local variable to stop the Monitoring Loop. This message alone causes all three loops to stop.

Monitoring Loop

The Monitoring Loop runs periodically when connected and uses network-published shared variables to display the monitored values from the real-time target. You use network-published shared variables instead of Network Streams because you are concerned only with the current value of the data, and you are OK with missing a few data points here and there.

Note that you use the Dynamic Shared Variable API instead of the static shared variable nodes. This is because you can dynamically change the IP address of the real-time target from the front panel. To ensure that you connect to the Shared Variable Library that is hosted on the real-time target corresponding to the IP address on the front panel, use the IP address on the front panel when opening a connection to each variable.

When you use static shared variable nodes, the IP address is statically configured in the LabVIEW project. If you change the IP address on the front panel, the UI Main VI continues to connect to the IP address defined in the LabVIEW project.

The shared variable connection is created in the UI-Initiate Connection.vi found in the Connect state of the UI Message Loop.

For more information on the LabVIEW FPGA Control on CompactRIO sample project, view the documentation in the created project or follow the More Information link in the Create Project window.

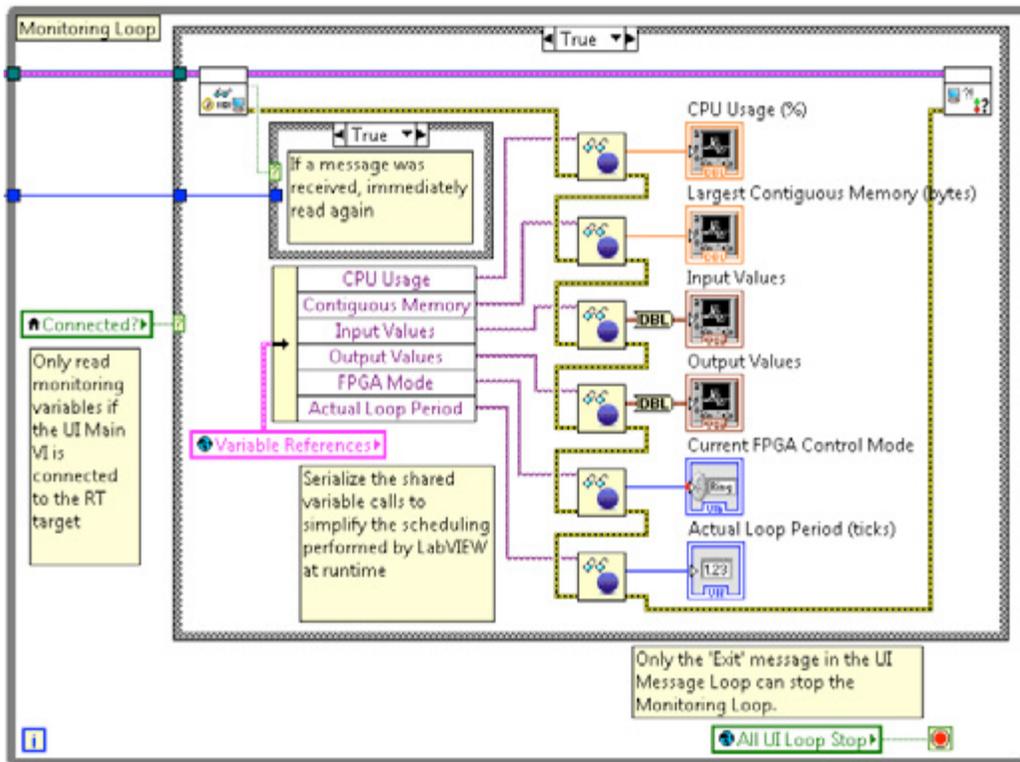


Figure 13.36. The Monitoring Loop receives current value data from the CompactRIO device and displays it to the UI.

Downloading the FPGA Bitfile to Flash Memory

This FPGA VI was written to be loaded to flash memory using the NI RIO Device Setup utility and configured to reload the FPGA bitfile only upon cycling the power. The benefit of loading the bitfile to flash memory is that you ensure that the bitfile is always loaded and running, even if you need to restart the real-time controller. This is especially important for this specific application since the hardware outputs must stay in a safe state upon rebooting the real-time controller. For more information, see the “Deploying Stand-Alone FPGA Applications” section of the white paper Managing FPGA Deployments.

Configure your FPGA bitfile to download directly to flash memory using the following instructions:

1. Right-click the LabVIEW FPGA Build Specification in the LabVIEW project and select Properties.

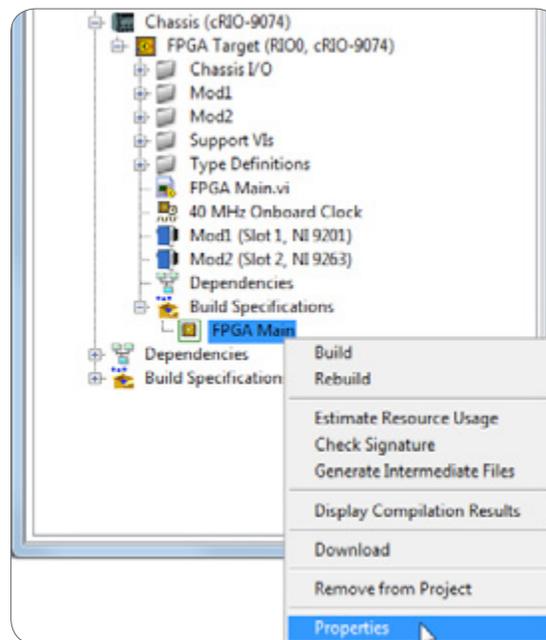


Figure 13.37. To download a LabVIEW FPGA bitfile to flash, right-click the LabVIEW FPGA Build Specification and go to Properties.

2. On the Information page, ensure that “Run when loaded to FPGA” is checked. If it’s not checked, check it and rebuild the bitfile.

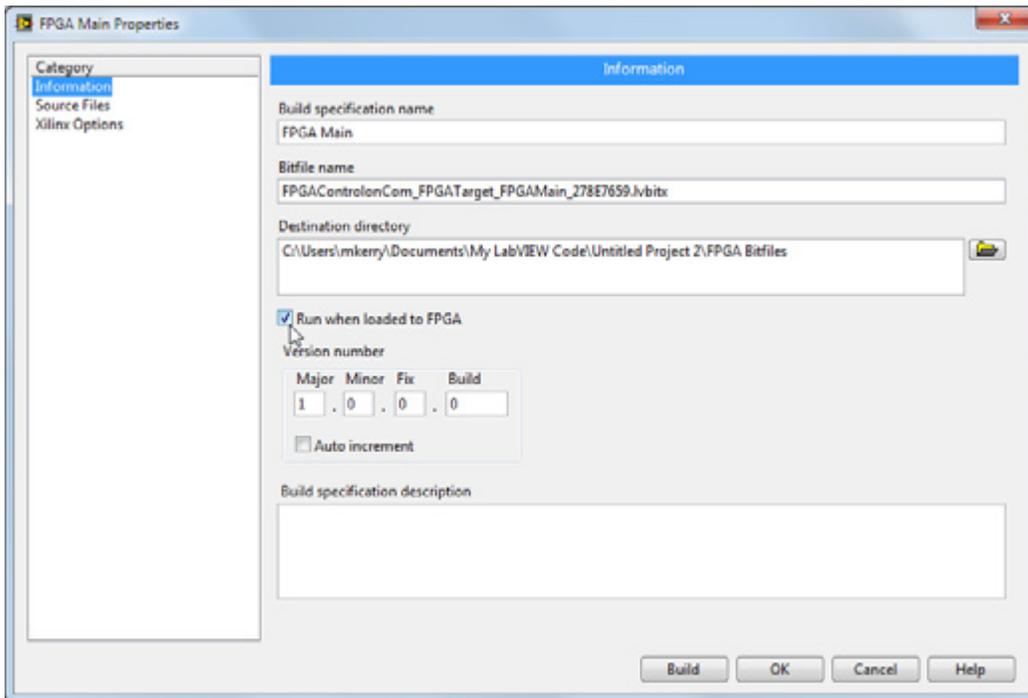


Figure 13.38. Ensure that “Run when loaded to FPGA” is checked.

3. To load your FPGA bitfile to flash memory, launch the RIO Device Setup utility by right-clicking your FPGA target.

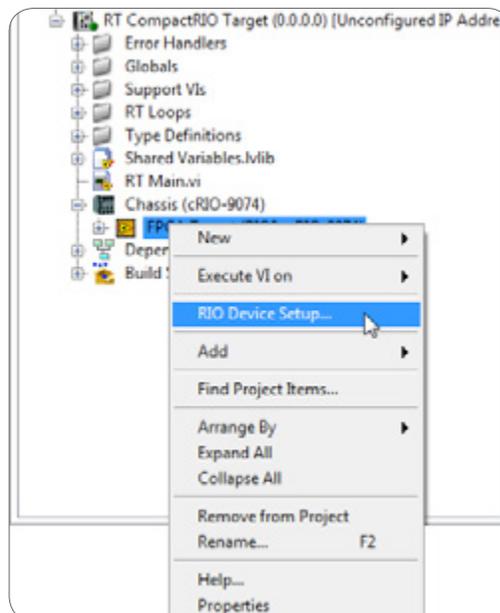


Figure 13.39. Launch the RIO Device Setup Utility from the LabVIEW project.

4. On the Download Bitfile to Flash tab, select the bitfile you want to use and click the Download Bitfile button.

5. On the Device Settings tab, select “Autoload VI on device powerup” and press the Apply Settings button.

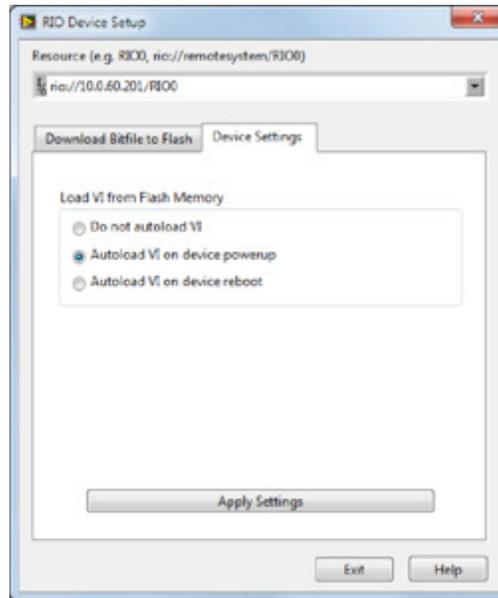


Figure 13.40. Configure your FPGA bitfile to load to your device every time you cycle the power.

6. Now your FPGA bitfile loads to your device every time you cycle the power. If you perform a system reset, the FPGA bitfile stays loaded, and the real-time OS restarts.

7. Finally, open the Initialize and Run FPGA subVI within the RT Main VI and ensure that the Open FPGA VI Reference is configured to not run the FPGA VI when called. Since the FPGA bitfile already is loaded, calling this function resets the FPGA bitfile, which is not necessary.

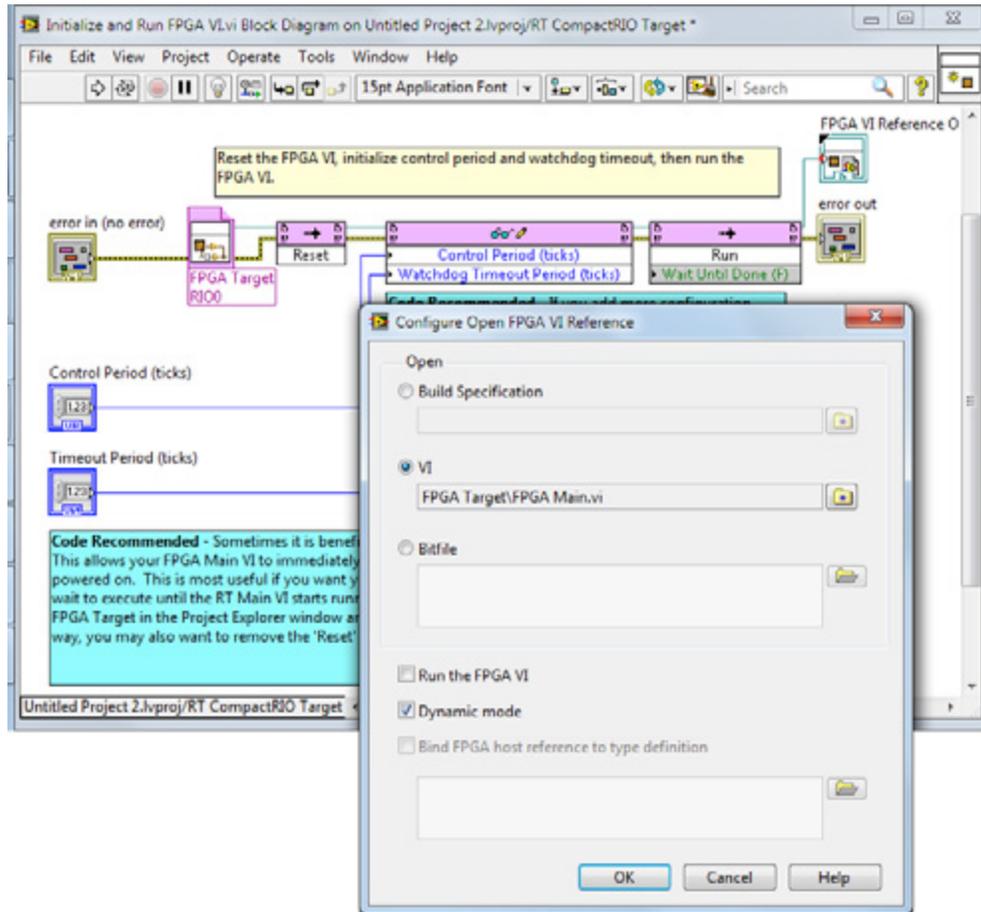


Figure 13.41. Uncheck the Run the FPGA VI option in the Configure Open FPGA VI Reference dialog.