A Primer for Machine Control
Using NI LabVIEW Real-Time and CompactRIO

Contents

Overview........................................................................................................................................... 3
Terminology........................................................................................................................................ 4
Machine Control Architecture Overview.......................................................................................... 4
  Control System Configurations....................................................................................................... 5
  Control System Block Diagrams ..................................................................................................... 6
Introduction to CompactRIO ............................................................................................................... 8
  Hardware Architecture ................................................................................................................... 8
    Real-Time Controller .................................................................................................................... 8
    Reconfigurable FPGA Chassis .................................................................................................... 9
    Industrial I/O Modules ............................................................................................................... 9
  CompactRIO Specifications ........................................................................................................... 10
Basic Controller Architecture Background...................................................................................... 11
  I/O, Communications, and the Memory Table .............................................................................. 12
  Control and Measurement Tasks ............................................................................................... 12
Basic Controller Architecture Example in LabVIEW ...................................................................... 13
State-Based Designs........................................................................................................................... 18
  State Machine Overview ............................................................................................................. 19
    Example Scenario of Developing with State Machines ............................................................. 19
  State Machine Example in LabVIEW .......................................................................................... 20
  Introduction to Statecharts .......................................................................................................... 24
LabVIEW Statechart Module ......................................................................................................... 26
  Statechart Regions ....................................................................................................................... 26
  Statechart States ............................................................................................................................ 27
  Orthogonal Regions and Concurrency ......................................................................................... 29
  Transitions .................................................................................................................................... 30
  Pseudostates .................................................................................................................................. 31
  Connectors ...................................................................................................................................... 31
Statechart Example in LabVIEW ....................................................................................................... 31
  Using LabVIEW Statecharts ....................................................................................................... 32
  Design the Caller VI ...................................................................................................................... 33
  Define the Inputs, Outputs, Triggers ......................................................................................... 34
Develop a Statechart Diagram .......................................................... 35
Place the Statechart in the Caller VI ............................................. 35
Quick Start – Modifying an Example ............................................ 37
Overview ....................................................................................... 37
Modify the IO Library .................................................................. 37
Modify the Shutdown Routine ...................................................... 38
Modify Task 1 to Map the I/O ....................................................... 38
Modify/Rewrite the Statechart ...................................................... 39
Reusable Functions ...................................................................... 39
Overview ....................................................................................... 39
Building Reusable Code in LabVIEW ........................................... 40
Example of Building Reusable Code in LabVIEW ....................... 41
Other Reusable Code in LabVIEW ................................................. 44
IEC 61131 Function Blocks ............................................................ 44
Configurable Terminal Variables .................................................. 45
Multiple Tasks .............................................................................. 45
Overview ....................................................................................... 45
Setting Task Priority and Synchronizing Tasks ......................... 46
Passing Data between Tasks .......................................................... 47
Triggering Tasks .......................................................................... 49
Software-Triggered Timing Source .............................................. 49
Errors and Faults ......................................................................... 51
Overview ....................................................................................... 51
The Fault Engine .......................................................................... 51
Recording Errors .......................................................................... 51
Error Logic ..................................................................................... 52
Fault-Handling Loop ................................................................. 52

©2009 National Instruments. All rights reserved. CompactRIO, FieldPoint, LabVIEW, National Instruments, NI, and ni.com are trademarks of National Instruments. Other product and company names listed are trademarks or trade names of their respective companies.
Overview

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

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

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

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

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

**Machine Control Architecture Overview**

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

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

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

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

You can program HMIs on a PC running Windows or a touch panel computer running an embedded OS such as Windows XP Embedded. HMI features typically include the following:
• Touch screen operation
• A paged display system with navigation controls
• Data entry objects (buttons, keypads, and so on)
• Alarm/event displays and logs

Control System Configurations
The simplest machine control system consists of a single controller running in a “headless” configuration (see Figure 1). This configuration is used in applications that do not need an HMI except for maintenance or diagnostic purposes.

![Figure 1. A Headless Controller](image)

The next level of system capability and complexity adds an HMI and/or additional controller nodes (see Figure 2). This configuration is typical for machines controlled by a local operator.

![Figure 2. A Local Machine Control System](image)

Complex machine control applications may involve many controllers and HMIs (Figure 3). They often involve a high-end server that acts as a data-logging and forwarding engine. This system configuration supports physically large or complex machines. With it, you can interact with the machine from various locations or distribute specific monitoring and control responsibilities among a group of operators.
Control System Block Diagrams

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

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

An HMI has similar components except instead of performing control, it provides a user interface (UI). You can perform additional tasks such as alarm and event detection and logging on both the controller and the HMI.
By analyzing controller operations, you can break down the system into smaller components, each responsible for a specific task in the overall application. Figure 5 shows the controller architecture and individual components of the machine control application. Some of these components are ready-to-run as part of the machine control reference architecture, while others must be developed as part of the design and implementation of a specific machine control application.

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

**Hardware Architecture**

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

*Figure 6. Reconfigurable Embedded System Architecture*

**Real-Time Controller**

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

*Figure 7. NI cRIO-9014 Real-Time Controller*
Reconfigurable FPGA Chassis
The reconfigurable FPGA chassis is the center of the embedded system architecture. The reconfigurable I/O (RIO) FPGA is directly connected to the I/O modules for high-performance access to the I/O circuitry of each module and unlimited timing, triggering, and synchronization flexibility. Because each module is connected directly to the FPGA rather than through a bus, you experience almost no control latency for system response compared to other industrial controllers. By default, this FPGA automatically communicates with I/O modules and provides deterministic I/O to the real-time processor. Out of the box, the FPGA enables programs on the real-time controller to access I/O with less than 500 nS of jitter between loops. You can also directly program this FPGA to run custom code. Because of the FPGA speed, this chassis is frequently used to create controller systems that incorporate high-speed buffered I/O, very fast control loops, or custom signal filtering. For instance, using the FPGA, a single chassis can execute more than 20 analog proportional integral derivative (PID) control loops simultaneously at a rate of 100 kHz. Additionally, because the FPGA runs all code in hardware, it provides the high reliability and determinism that is ideal for hardware-based interlocks, custom timing and triggering, or eliminating the custom circuitry normally required with custom sensors.

Industrial I/O Modules
I/O modules contain isolation, conversion circuitry, signal conditioning, and built-in connectivity for direct connection to industrial sensors/actuators. By offering a variety of wiring options and integrating the connector junction box into the modules, the CompactRIO system significantly reduces space requirements and field-wiring costs. You can choose from more than 50 NI C Series I/O modules for CompactRIO to connect to almost any sensor or actuator. Module types include thermocouple inputs; ±10 V simultaneous sampling, 24-bit analog I/O; 24 V industrial digital I/O with up to 1 A current drive; differential/TTL digital inputs; 24-bit IEPE accelerometer inputs; strain measurements; RTD measurements; analog outputs; power measurements; controller area network (CAN) connectivity; and secure digital (SD) cards for logging. Additionally, the platform is open and you can build your own modules or purchase modules from other vendors. With the NI cRIO-9951 CompactRIO Module Development Kit, you can develop custom modules to meet application-specific needs. The kit provides access to the low-level electrical CompactRIO embedded system architecture for designing specialized I/O, communication, and control modules. It includes LabVIEW FPGA libraries to interface with your custom module circuitry.
Figure 9. You can choose from more than 50 I/O modules for CompactRIO to connect to almost any sensor or actuator.

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

<table>
<thead>
<tr>
<th>Description</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Electromagnetic Compatibility (EMC)</td>
<td>89/336/EEC EN 55011 Class A at 10 m</td>
</tr>
<tr>
<td></td>
<td>FCC Part 15A above 1 GHz Industrial levels per EN 61326-1:1997 + A2:2001, Table A.1</td>
</tr>
<tr>
<td></td>
<td>CE, C-Tick, and FCC Part 15 (Class A) Compliant</td>
</tr>
<tr>
<td>Product Safety</td>
<td>73/23/EEC EN 51010-1, IEC 61010-1</td>
</tr>
<tr>
<td></td>
<td>UL 61010-1</td>
</tr>
<tr>
<td></td>
<td>CAN/CSA C22.2 No. 61010-1</td>
</tr>
<tr>
<td>Hazardous Locations, Class I, Division 2</td>
<td>Class I, Division 2, Groups A, B, C, D, T4;</td>
</tr>
<tr>
<td></td>
<td>Class I, Zone 2, AEx nC IIC T4,</td>
</tr>
<tr>
<td></td>
<td>EEx nC IIC T4</td>
</tr>
<tr>
<td>Shock and Vibration</td>
<td>IEC 60068-2-64, IEC 60068-2-27,IEC 60068-2-6</td>
</tr>
<tr>
<td>Mean Time Before Failure (MTBF)</td>
<td>Bellcore Issue 6, Method 1, Case 3</td>
</tr>
<tr>
<td></td>
<td>MIL-HDBK-217F</td>
</tr>
<tr>
<td>Marine</td>
<td>Lloyds Register (LR Type Approval System Test Spec No. 1)</td>
</tr>
<tr>
<td>Quality/Environmental Management System (QMS/EMS)</td>
<td>ISO 9001/14001</td>
</tr>
</tbody>
</table>

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

Figure 10. CompactRIO Specifications
Basic Controller Architecture Background

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

A basic controller architecture has three main states:
1. Initialization (Housekeeping)
2. Control (IO and Comm Drivers, Memory Table, Control and Meas Tasks)
3. Shutdown (Housekeeping)

Figure 11. The Three Main States of a Basic Controller Architecture

The Initialization Routine

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

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

Many programmers are familiar with direct I/O access, during which subroutines directly send and receive inputs and outputs from the hardware. This method is ideal for waveform acquisition and signal processing and for smaller single-point applications. However, control applications normally use single-point reads and writes and can become very large with multiple states— all of which need access to the I/O. Accessing I/O introduces overhead in the system and can slow it down. Additionally, managing multiple I/O accesses throughout all levels of a program makes it very difficult to change I/O and implement features such as simulation or forcing. To avoid these problems, the control routine uses a scanning I/O architecture. In this type of architecture, you access the physical hardware only once per loop iteration using I/O and communication drivers (labeled as IO and Comm Drivers in Figure 11). Input and output values are stored in a memory table, and control and measurement tasks access the memory space instead of directly accessing the hardware. This architecture provides numerous benefits:

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

Control and Measurement Tasks

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

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

The user logic can:

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

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

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

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

Basic Controller Architecture Example in LabVIEW

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

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

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

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

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

Figure 13. Example PID Controller Architecture

Figure 14. Manage this controller sequence with three frames: initialization routines, control and measurement tasks, and shutdown routine.
3. Add any initialization or shutdown logic. In this application, no initialization is required for the controller. By default, the controller leaves output values at last state. In this application, at shutdown, you need to set outputs to an off state. You could also add other logic in the shutdown such as error logging into the structure.

![Figure 15. Add a caption here.](image)

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

**I/O Scan and Memory Table**

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

![Figure 16. Starting in LabVIEW 8.6 you can program CompactRIO controllers using the Scan Interface.](image)

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

![Image](image.png)

**Figure 17. Block Diagram Description of the CompactRIO Scan Interface Software Components**

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

![Image](image.png)

**Figure 18. Creating an I/O Alias**

1. To create an I/O alias, right-click on the controller and select a new variable. Select the variable type as I/O Alias and bind it to the physical I/O.
2. For this example, create two I/O aliases for Thermocouple 1 (bound to a TC module) and Heater 1 (bound to a digital output module configured for PWM output), and put both into a library called IO Library.
**Control and Measurement Tasks**

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

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

![Configure Timed Loop](image)

**Figure 19. Synchronizing the NI Scan Engine**

2. Write the control logic to read the inputs from the I/O aliases, run the logic, and write to the I/O aliases. Normally, you would create subVIs for encapsulation to enable code reuse; however, because this example is intended to show the overall architecture, the code is trivial. Further encapsulation of the code would be redundant; in later examples, you will learn about appropriate code encapsulation. For this simple example, drop a PID VI on the block diagram and wire constants so the output range is [100, 0], the PID gains are [10, 0.1, 0], and the setpoint is 350. Instead of constants, these can also be variables that you can reconfigure while the program is executing. Wire the “Temperature 1” I/O alias to the “Process Variable” terminal and wire the “Heater 1” I/O alias to the “Output” terminal. Add the appropriate error-handling components.
Figure 20. In this example, use a network published variable to stop the loop.

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

Figure 21. Fundamental Architecture for Machine Control

**State-Based Designs**

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

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

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

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

Example Scenario of Developing with State Machines

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

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

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

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

<table>
<thead>
<tr>
<th>I/O Signals</th>
<th>I/O Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operator push button</td>
<td>Input_Operator_PB</td>
</tr>
<tr>
<td>Pump A</td>
<td>Output_PumpA</td>
</tr>
<tr>
<td>Pump B</td>
<td>Output_PumpB</td>
</tr>
<tr>
<td>Chemical A Flow</td>
<td>Input_ChemA_Flow</td>
</tr>
<tr>
<td>Chemical B Flow</td>
<td>Input_ChemB_Flow</td>
</tr>
<tr>
<td>Stirrer</td>
<td>Output_Stirrer</td>
</tr>
<tr>
<td>Heater</td>
<td>Output_Heater</td>
</tr>
<tr>
<td>Thermocouple</td>
<td>Input_TC</td>
</tr>
<tr>
<td>Drain Pump</td>
<td>Output_PumpDrain</td>
</tr>
<tr>
<td>Tank Empty Level Sensor</td>
<td>Input_TankEmpty_LS</td>
</tr>
</tbody>
</table>

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

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

Once you have defined the states within a system, use a case structure in LabVIEW to represent and contain the logic for each state.
By adding a shift register to the application, you can keep track of and pass current state information each time the state machine executes. The case structure input terminal is connected to the shift register.

Each state within the case structure now can contain some LabVIEW code that executes each time the state becomes active. The code in Figure 25 executes a PID function for mixing and heating your chemical reacting vessel.
Each state must then determine which state to transition to next depending on the conditions within the system. Add transition logic to determine which state should be executed next.

For this example, use simulated I/O instead of physical I/O so you can test your logic. To do this, use a global variable instead of hardware read/write VIs. This is a convenient way to test your logic with interactive controls and indicators before deploying to actual hardware. The ability to easily simulate I/O is one of the benefits of this architecture.
Because one state in your machine has parallel processes, you need to build a second state machine to represent the parallel logic and call parallel processes in that state. Exit the state only when both parallel processes have completed.
Introduction to Statecharts

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

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

To begin understanding statecharts, it is best to start with the classic state diagram. The classic state diagram consists of two main constructs: states and transitions. In Figure 29, the state diagram describes a simple soda vending machine with five states and seven transitions to illustrate how the machine operates. The machine starts in the “idle” state and transitions to the “count coins” state when coins are inserted. The state diagram shows additional states and transitions when the machine waits for a selection, then dispenses a soda, and finally gives change.

Figure 29. State Diagram of a Simple Soda Vending Machine

Figure 30 shows a statechart that describes the behavior of the same machine. In the statechart, you can nest the “count coins” and “dispense” states within a superstate by using a statechart feature called hierarchy, allowing states to be embedded within another. With hierarchy, you can simplify your designs. Now you have to define only one transition (T3) from either of these two states to the “give change” state. You can configure the T3 transition to respond to three events: soda dispensed, change requested, or coins rejected. Notice how the “select soda” state in the classic state diagram has been removed. This is accomplished using a “guard” condition to transition T2. With guard conditions, you can embed logic within a transition. Guard conditions
must evaluate to “true” for the transition to occur. If the result of the guard condition is “false,”
the event is ignored and the transition does not take place.

![Image: Vending Machine Statechart]

**Figure 30. Statechart of a Simple Soda Vending Machine**

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

![Image: Vending Machine Statechart]

**Figure 31. The T7 transition shows how statecharts can define an exit that applies to both substatecharts.**
In addition to hierarchy and concurrency, statecharts have features that make them valuable for complex systems. They have a concept of history, allowing a superstate to “remember” which substate within it was previously active. For example, consider a superstate that describes a machine that pours a substance and then heats it. A halt event may pause the execution of the machine while it is pouring. When a resume event occurs, the machine remembers to resume pouring.

**LabVIEW Statechart Module**

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

**Statechart Regions**

A region is an area that contains states. The top-level statechart diagram is a region within which states are placed. Additionally, you can create regions within states to take advantage of hierarchical designs by creating states within another state. This ability is illustrated in Figure 32, where a substate has been created within a state using a region. Each region must contain an initial pseudostate.

![Figure 32. Create a substate within a state using a region.](image)
Statechart States
A state is a condition of a statechart. You must place states within regions and have at least one incoming transition.

Figure 33. A state is a condition of a statechart.

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

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

Figure 34. You can access entry and exit code through the Configure State dialog box.

You can further configure states to have static reactions, which are the actions a state performs when it is not taking any incoming or outgoing transitions. An individual state can have multiple static reactions that can execute at each iteration of the statechart.
Each static reaction comprises three components – trigger, guard, and action.

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

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

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

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

![Configure State dialog](image)

*Figure 35. You can create static reactions through the Configure State dialog by creating a new reaction.*
Orthogonal Regions and Concurrency

When a state contains two or more regions, the regions are said to be orthogonal. Regions 1 and 2 in Figure 36 are orthogonal.

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

Transitions define the conditions that statecharts move between states.

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

Figure 37. Transitions define the conditions that statecharts move between states.

Figure 38. Configure transition nodes through the Configure Transition dialog.
Triggers, guards, and actions behave the same way in transitions that they do in states. A transition responds to a trigger, and if the guard code evaluates to true, the action is executed and the statechart moves to the next state. If the guard code does not evaluate to true, the action code is not executed and the statechart does not move to the state indicated by that transition.

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

- **Initial state** – Represents the first state that occurs when entering a region. An initial state must be present in each region.

- **Terminal state** – Represents the final state of a region and ends the execution of all states within that region.

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

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

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

- **Fork** – Splits one transition segment into multiple segments.

- **Join** – Merges multiple transition segments into one segment.

- **Junction** – Connects multiple transition segments.

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

1. Wait for an operator start command from a push button.
2. Meter two chemical flows into a tank based on output from a flow totalizer (two parallel processes – one for each chemical flow).
3. After filling the tank, turn on a stirrer and raise the temperature in the tank. Once the temperature has reached 200 °F, the system turns off the stirrers and holds the temperature constant for 10 seconds.
4. Pump the contents to a holding tank.
5. Go back to wait state.
Note that for simplicity in this application, the chemical flow rates have been hard coded to 850, the temperature to 200 °F, and the time to 10 seconds. In a real application, you can load these values from a recipe or an operator can enter them.

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

![Image of LabVIEW Statechart]

Figure 39. Create a library with I/O aliases to each of the I/O signals.

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

![Image of Shutdown Outputs VI]

Figure 40. Create the shutdown task to set the default output states for the output I/O aliases.
The process for developing a statechart-based application involves the following steps:

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

**Design the Caller VI**

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

![Statechart Diagram](Image)

*Figure 41. The top-level VI includes a timed loop that continuously calls your statechart. It also includes code sections for startup and shutdown, which are encapsulated within a sequence structure.*

Now add a new statechart into the LabVIEW Project. Each LabVIEW statechart has several components that you can use to configure the context of the design.
The diagram.vi file contains the actual statechart diagram. The inputs.ctl and outputs.ctl are clusters that define the inputs and outputs to the statechart. The statedata.ctl is for internal state information only used in the statechart. For this example, do not use the triggers, statedata.ctl, or customdatadisplay.vi.

**Define the Inputs, Outputs, Triggers**

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

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

Place the Statechart in the Caller VI

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

![Statechart](image.png)

**Figure 44. Click on the icon on the upper left to have LabVIEW generate code for the statechart.**

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

![Statechart Diagram](image.png)

**Figure 45. Create subVIs to read the I/O aliases and pass them in and out of statechart clusters.**

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

Figure 46. If there is no error the statechart will execute.

Figure 47. If an error occurs the statechart will not execute.
Quick Start – Modifying an Example

Overview

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

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

Modify the IO Library

Open the Chemical Mixing.lvproj. Because your application uses different I/O, you need to modify the IO Library to create IOV aliases for your physical I/O. If you have finalized your wiring, you can map the IOV aliases to the physical I/O now. If you have not finalized the wiring, you can remap the IOV aliases later.

1. Expand the IO Library in the project. You can edit the variables one at a time by double-clicking on the alias. For a faster method, use the Multiple Variable Editor. You can open the editor by right-clicking on the library and selecting “Multiple Variable Editor…”

![Project Explorer](image)

*Figure 48. You can edit variables faster with the Multiple Variable Editor.*
2. In the Multiple Variable Editor, change the names, data types, and physical bindings (alias path) of existing variables. You can also quickly create new variables by copying and pasting existing variables.

![Multiple Variable Editor Options](image)

**Figure 49. Multiple Variable Editor Options**

**Modify the Shutdown Routine**

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

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

**Modify Task 1 to Map the I/O**

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

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

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

**Modify/Rewrite the Statechart**

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

**Reusable Functions**

**Overview**

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

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

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

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

   ![Figure 51](image)

   *Figure 51. On the connector pane wire inputs and outputs to front panel controls and indicators.*

2. The code must maintain its own memory space so it can retain state (this may not be required on some functions).
   - In LabVIEW, you can do this one of two ways. You can use a while loop with uninitialized shift registers to hold memory or you can create local variables. Local variables have slightly more overhead but are more flexible and easier to understand. You can create local variables from front panel controls and indicators. Right click on the control and create a local variable. This variable can be referenced multiple times on the block diagram.
3. The code must be capable of having multiple instances in one program.
   - In LabVIEW, do this by making the VI reentrant. A reentrant VI has a separate memory space for each instance it is called in a program. To make a subVI reentrant, go to the VI Properties page (under File), select Execution on the Category pull-down menu, and check the box for Reentrant execution.

**Figure 53. Making a VI Reentrant**

**Example of Building Reusable Code in LabVIEW**

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

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

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

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

![Elapeled Timer Locals.vi Front Panel](image)

*Figure 54. Create a new VI to call the code and provide input and output data.*
2. The code must maintain its own memory space.
   - On the front panel, create controls and indicators for the three locals: Previous Tick Count, Wrapped, and Initial Tick Count.

   ![Image](image1.png)

   **Figure 55.** Create controls and indicators for the three locals: Previous Tick Count, Wrapped, and Initial Tick Count.

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

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

![Image](image2.png)

**Figure 56.** The finished code.
Other Reusable Code in LabVIEW

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

IEC 61131 Function Blocks

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

Figure 57. New LabVIEW Function Blocks Based on the IEC 61131-3 International Standard for Programming Industrial Control Systems
Configurable Terminal Variables
Function blocks differ from standard subVIs by offering configuration pages and providing the option to directly connect the inputs and outputs to entries in the global memory table that are visible in the LabVIEW Project. You also can access these entries through the network. You can configure the terminals and variables from the function block Properties window.

Figure 58. Configure function blocks from the Properties window and access inputs and outputs from the memory table.

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

Figure 59. The control routine can have multiple tasks that run in parallel and pass data to the memory table.
To manage the execution of your application, you need to be able to:

- Set the priority between the tasks
- Synchronize the tasks
- Pass data between the tasks
- Trigger the tasks

**Setting Task Priority and Synchronizing Tasks**

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

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

![Configure Timed Loop](image)

**Figure 60. Set the priority of a timed loop.**
To synchronize multiple tasks, set them all to synchronize to the NI Scan Engine. All loops synchronized to the scan engine run at the I/O scan rate and execute from highest priority to lowest priority.

If you have background tasks or nondeterministic tasks that do not need synchronization or priority, you can run those tasks in a standard while loop with a wait function to set timing.

![Figure 61. While loops operate at normal priority.](image)

**Passing Data between Tasks**

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

![Figure 62. Shared variables and IO Variables are both elements of the memory table.](image)
Creating a new shared variable is similar to creating an I/O alias. You can organize these variables by using libraries. First create a new library in the project and then create a new variable. Select the data type and set the variable type as Single Process (a single process variable is a global variable).

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

Figure 63. Creating a Shared Variable

Figure 64. Enabling the RT FIFO allows you to read and write to a shared variable from multiple parallel loops without inducing jitter.
You can now read and write to this memory table element from anywhere in your control code, just like I/O aliases.

**Triggering Tasks**

Sometimes it is ideal to trigger a task from another task. For instance, the top-level machine control code may manage the machine and, in one state, need to start a waveform acquisition and analysis operation. This waveform operation needs to take place in a lower-priority parallel task. There are numerous methods to trigger parallel loops in LabVIEW. The simplest is to poll a shared variable to check for a change of state. A better way is to set the variable to multi-element and check to see if a new element has been added. Both of these methods increase processor usage by requiring the processor to poll the element. Additionally, while the loop is in the wait stage, it is unresponsive to triggers. A better method is to directly trigger the parallel timed loop.

**Software-Triggered Timing Source**

LabVIEW offers you the ability to create a software timing source for the timed loops. When you use this timing source, the timed loop sleeps until it gets a trigger or the timeout is exceeded. It is important to set a timeout for timed loops that are triggered so that the program shuts down correctly. By default, the timeout is -1, and the loop will wait forever.

![Figure 65. Setting a timeout assures that a program remains responsive to shut-down commands.](image)
Once you have configured a timed loop, you can create a software-triggered timing source in the initialization routine. The name of the timing source is wired to the timed loop, and the timing source in the shutdown routine is destroyed. Now, from anywhere else in the program, you can trigger this timed loop by firing the software-triggered timing source. The source is triggered by its number. In this example, the top loop is set to run at a higher priority. When it decides to fire the SW trigger (in this case, based on the Boolean trigger), the software trigger is set and the lower loop runs the trigger logic. The lower loop also monitors the wakeup reason to determine if it timed out or if it was triggered. In the timeout case, it does nothing.

*Figure 66. Triggering parallel loops from a main control loop.*
Errors and Faults

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

The Fault Engine

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

To use the fault engine, you need to:
- Record any errors
- Determine appropriate logic for any errors
- Create a fault-handling loop to monitor and respond to any errors

Recording Errors
To enter new errors into the fault engine, use the Set Faults.vi. This takes an error cluster as an input. If an error occurs, the Set Faults.vi automatically logs this as a fault. You can also enter your own user-defined faults by creating and wiring a user-defined error cluster.

Figure 67. The Faults Pallet.
**Error Logic**

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

**Fault-Handling Loop**

You also need to create a loop to monitor any faults and run your fault logic. This loop should run at the highest priority so that it always operates and is not “starved” by lower-priority tasks.

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

*Figure 68. A complete application with fault handling.*