Module 7: State Machines (Queue based at end)

Overview

The state machine is one of the fundamental architectures NI LabVIEW developers frequently use to build applications quickly. Developers use state machines in applications where distinguishable states exist. Each state can lead to one or multiple states and can 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 “initialize” state followed by a default state, where you can perform many different actions. These actions depend on previous and current inputs as well as states. You can use a “shutdown” state to perform cleanup actions.

In LabVIEW software, you can create a basic state machine with a while loop, a shift register, a case statement, and some form of case selector (case selectors are discussed in a later section).

The while loop is the main program loop, which executes until the conditions for exiting the program are met.

The while loop’s main responsibility is to call the case selector and then execute the appropriate case.

The shift register keeps track of which case should execute next.

Finally, each case of the case statement contains the action for one specific use action.

Often the default case is used as the place to check the case selector (in other words, if the user did nothing, check again to see if he has done something yet).
When designing state machines, you can create a state diagram to graphically represent the different states and how they interact. Use state diagrams, the design frameworks for state machines, to model the control algorithms you need with discrete logical states. State Diagrams make it easy to develop and understand the functionality of an application that uses a state machine.

The figure below is an example of a state diagram. The ovals represent the states and the arrows represent the possible transitions between states.

All applications require an initial state, or starting point, followed by transition states that perform different actions. A terminal state, or ending point, is the final state executed and performs cleanup actions.

State diagrams are useful in simplifying the design process of applications that use complex decision-making algorithms. To create an effective state diagram, you must know the various states of the application and how they relate to one another. By visualizing the various execution states of the application, you improve the overall design of the application.

**Design a State Machine with a State Diagram**

Imagine a vending machine that accepts combinations of nickels and dimes to get a coke. The cost of a coke is 15 cents and the machine does not return change. First, establish the states that the vending machine might be in:

- Start: No money inserted
- 5 cents
- 10 cents
- Done: 15 cents

Now think about the possible ways or paths that the vending machine can take to get into these states. Don’t forget to consider how the vending machine starts in the initial state.
• Nickel is inserted
• Dime is inserted
• Default: Nothing is inserted

From these lists, you see that there are four states and three possible paths from each state. You need to depict which states are connected by which paths. For example, when the vending machine is in the initial start state, the total change inserted is 0 cents. When a nickel is inserted, the vending machine must go to the 5 cent state. Therefore, the start state leads to the 5 cent state by the nickel path. By considering all states and paths, you can create a state diagram for the vending machine:

With a state diagram, you can better understand how to create a state machine.

**Building a State Machine**

Using the state diagram above, create a state machine.

1. Create a new blank VI.
2. On the front panel place:
   • Two text buttons named “Nickel” and “Dime”
   • Text indicator named “Money Deposited”
   • Boolean indicator named “Dispense coke”
3. Place a while loop on the block diagram.
4. Place a case structure in the while loop.
5. Create a shift register on the while loop.
6. Create an Enum and wire it to the shift register to initialize it.

7. Right-click the Enum, select Edit Items, and add the following “states”:
   - Start
   - 5 cents
   - 10 cents
   - Dispense
8. Wire the shift register to the conditional input of the case structure.

9. Right-click the case box at the top of the case structure and select **Add Case for Every Value**.

10. Wire the different cases as depicted in the following figures.
11. Wire **Dispense coke** to the Boolean output of the case structure.

12. Inside the while loop, place a wait function with a constant.
13. Outside of the while loop, wire a constant of 0 to the **Money Deposited** string indicator.

The finished VI should look like this:
The Dequeue Element VI obtains the next state from the queue. If it is called and there are no elements currently stored in the queue, it returns the default value for the queue's datatype. In our case, the datatype is string, so the default value is an empty string. The state machine is set up so that an empty string returns us to the wait state. This functionality allows us to reset the state machine without passing back a specific "wait" element at the end of each task's execution, giving us more room in each case of our case structure.
Queued State Machine with User Input
Queued State Machine

To add a new state, add the state to the Type Def Enum, the outer case structure, the inner case structure inside of the Done Event, and the case structure in the Calculate State subVI.

1. Initialize array of 50 states and set Selected States to empty.
2. Stop selecting states and finalize State array. Move to first State in the Selected States array.
3. "Stop", Default
4. "Select States"
Queue Basics

To demonstrate the queue VIs, type in some elements in the "Array of strings to be placed in the queue" array, and set the "Number of items to place in the queue" Control to some value. When the VI is run, you will see the array elements inserted into the queue, and then removed. Note that if you specify more items to place in the queue than you have entered in the array, the VI does not finish. Instead, it waits for another item to be placed in the queue before continuing. We will use this behavior in the queued state machine VIs.

The Get Queue Status VI returns the number of items currently in the queue, and an array of items in the queue.

The Track Queue Status While Loop is used to allow the Get Queue Status VI to run continuously. This enables it to update its outputs while the two For Loops are running.

The Not A Queue VI returns a Boolean True if the queue is not valid. In this instance, it is used to end the While Loop after the queue is destroyed by the Destroy Queue VI.

The Insert Elements For Loop selects one item from the string array on each iteration, and inserts that element into the queue.

The Remove Elements For Loop removes one element from the queue on each iteration. The removed element is displayed on the Front Panel.
The state machine stays in state 0 until the Run States switch is pressed. It then executes states 1, 2, 4, and 3 before being routed back to State 0.

As long as the Run States switch is false, the State 0 Case places a string value of "State0" onto the queue. This ensures the state machine stays in the State 0 state until the switch is pressed.
Overview

This document shows one software design architecture for machine control. Their overall software design architecture is documented in an Example program in the Developer Zone article Machine Control Software Design Examples.

This architecture can be used to build control applications on LabVIEW Real-Time controllers including CompactRIO, Compact FieldPoint, and PXI. The same basic architecture could also work on other platforms such as Windows based controllers. LabVIEW Real-Time is a full programming language providing developers numerous options of how to construct a controller and enabling them to create very flexible and complex systems. LabVIEW Real-Time controllers are being used in applications ranging from control of nuclear power plant rods, to hardware in the loop testing for engine ECUs, to adaptive control for oil well drilling, to high speed vibration monitoring for predictive maintenance. However, the flexibility of LabVIEW Real-Time can make starting a new application daunting, especially for programmers new to LabVIEW or new to programming real-time systems. This document does not attempt to provide an architecture for every application and is not intended as a replacement for LabVIEW Real-Time training. It is written to help provide a framework for engineers designing industrial control applications, especially engineers who are familiar with the use of PLCs. This document is designed to explain how to easily construct LabVIEW Real-Time applications that also provide the flexibility to handle non-traditional applications such as high speed buffered I/O, data logging, or machine vision.

Table of Contents

1. Terminology
2. Basic Controller Architecture Background
3. The Control Routine Details
4. User Logic Tasks
5. Basic Controller Architecture Example in LabVIEW
6. State Machines
7. State Machine Example in LabVIEW
8. Quick Start – Modifying an Example

Terminology

Although there are many ways to build a control application in LabVIEW Real-Time, there are fundamental concepts of real-time programming and control applications that need to be understood.
**Responsiveness:** A control application will need the ability to take action 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 will have different tolerances for responsiveness varying from uS to minutes. Most industrial applications have responsiveness requirements in the mS to seconds range. An important design criterion for a control application is the required responsiveness as this will determine the control loop rates and will affect decisions on I/O, processor, and software.

**Determinism and jitter:** Determinism is the repeatability of the timing of a control loop. Jitter is how you measure determinism and is the error in timing. For example, if a loop is set to run once every 50mS but it sometimes runs at 50.5 mS then the jitter is 0.5mS. 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 will lead to poor analog control and can make a system not responsive.

**Priority:** Most controllers use a single processor to handle all control, monitoring, and communication tasks. Since there is a single resource (processor) with multiple parallel demands, the programmer needs a way to manage what demands 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, the control loop can be set to a high priority to preempt the logging operation and provide deterministic temperature control. This assures that lower priority tasks such as logging, a web server, HMI, etc do not negatively affect analog controls or digital logic.

**Basic Controller Architecture Background**

Most programmers who are new to LabVIEW start with a very basic application and then expand their program to add more functions. One benefit of LabVIEW is that it is very easy to build and run a simple program. For instance here is the logic for a simple PID loop running on a Compact FieldPoint system.
Unfortunately building more complex systems in LabVIEW requires a more sophisticated architecture that allows better code reuse, scalability, and management of execution. This section will describe and build a basic architecture for control applications and will perform the same PID loop using this architecture.

The most basic controller has three main sequences or routines:

1. Initialization Routines
2. Control Routines
3. Shutdown Routines

These routines deal with the state of the controller and are not dependent on the user logic. Because National Instruments real-time controllers allow you to define the full capabilities of your system you will need to build the sequences in LabVIEW.
The initialization routine:

Before executing the main control loop the program needs to perform an initialization routine. The initialization routine will prepare the controller for execution. This 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 will:

1. Set all internal variables, including output variables in the I/O memory table, to default states
2. Create all necessary queues, RT FIFOs, VI Refnums and download any FPGA bit files.
3. Perform any additional user defined logic to prepare the controller for operation such as preparing log files.

The control routine:

This is the heart of the controller and where all the user logic executes. It executes a scanning architecture with multiple parallel loops. Although you can diagram it as one loop performing the I/O Scan Task, high priority tasks, and then low priority tasks in reality it is multiple synchronized loops.

![Control Routines Diagram]

The first loop will write and read physical IO to a memory table and will execute first. Then subsequent tasks will execute user logic from highest priority to lowest priority ending with housekeeping and communication tasks that will operate at the lowest priority. You must verify that you have programmed your controller with enough idle processor time to service housekeeping and communication tasks.

The shutdown routine:

When the controller is commanded to stop running or experiences a fault condition, it will stop running the main control loop and will run a shutdown routine. The shutdown routine is for shutting down the controller and putting it in a safe state. It is only used for controller shutdown, and is not the place for machine shutdown routines which should go in the main control loop. The shutdown routine will:

1. Set all outputs to safe states
2. Stop any parallel loops that are running
3. Perform any additional logic such as notifying operator of a controller fault or logging state information

The Control Routine Details

Most LabVIEW programmers are familiar with direct I/O access where 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. (Note that this architecture can also support waveform measurements and waveforms are covered in a later section.) Accessing I/O introduces overhead on the system and can be relatively slow, 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 a scanning I/O architecture the physical hardware is only accessed once per loop iteration. Input and output values are stored in memory in a construct known as an I/O memory table and the control code accesses the memory space instead of the direct hardware. This architecture provides numerous benefits:

- I/O abstraction so subVIs and functions can be reused (no hard coding of I/O)
- Low overhead
- Deterministic operation
- Easy transition between HW platforms
- Support for simulation
- Support for interlocks and “forcing” (covered later)
- Elimination risk of I/O changes during logic execution
I/O Memory Table:

The I/O memory table is intended to pass data between the I/O scan loop and other parallel control or measurement loops or tasks. There are numerous methods for inter-loop communication, but one of the simplest is to use a global variable. To make updating the program simple we will use a global with multiple clusters. Each cluster must only have one “writer” to avoid race conditions where multiple data sources overwrite data. In this case each task that writes data to the global has a dedicated cluster.
**I/O Scan:**

The main control loop will be the highest priority loop so it will run first each cycle. It will have two steps occurring repeatedly at the desired control loop rate.

1. Write to outputs - The I/O memory table is read and output modules are updated.
2. Read from inputs - Input modules are read and the values are written to the I/O memory table.

*Note: Writing to the outputs first helps assure deterministic operation and provides a cleaner program for performing error handling and transitioning to a shutdown state.*

**User Logic Tasks**

The user logic is the machine specific logic that defines the control application. In many cases it is based on a state machine to handle complex machine control with multiple states. A later section will explore how to use state machines to design the user logic. To execute in the control architecture, user logic must:

- Execute in less time than timed loop 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”, instead use timer functions or “Tick Count” for timing logic
- Not perform waveform, logging, or non-deterministic operations (the use of parallel loops for these operations is explained later)

The user logic can:
• Include single point operations such as PID or point-by-point analysis
• Use a state machine to structure the code

Each logic task will be structured using a timed loop. The priority of the timed loop must be lower than the I/O scan loop and the timing should be an integer multiple of the I/O scan loop rate. The logic task will have three steps:

1. Copy I/O memory table to local cluster - The I/O memory table is read and a local cluster (strict type def) is written. A local cluster is used to provide scalability for multiple tasks and to enable the use of the LabVIEW StateChart module.
2. Execute logic – I/O is read and written to local clusters for inputs and outputs
3. Copy local cluster to I/O memory table outputs – The local output cluster (strict type def) is read and written to the I/O memory table. A local cluster is used to provide scalability for multiple tasks and to enable use of the LabVIEW StateChart module.

Basic Controller Architecture Example in LabVIEW

To demonstrate this control architecture, we will build a basic PID control application. This simple application controls a temperature chamber to maintain 350 degrees. It has one analog input from a thermocouple, one analog output (0-10V) that is connected to the input on a heater controller, and will use a PID algorithm for control. This application is overly simplistic and is used here to explain the architecture components without adding the complexity of an intricate control example. More detailed control examples are explored later which demonstrate using this architecture for more complex control applications.

I/O Memory Table:

To create the I/O memory table:

1. Create a global variable called IO Table Global.vi with two clusters, one for the data from the IO task and one for data from the Task 1. Create two controls, Input_TC and Output_Heater.
I/O Scan:

1. To create the I/O Scan:
2. Open a top level VI.
3. Drop a timed loop on the diagram, configure the rate to match the loop time required by your application.
4. On the diagram create two subVIs: Write Outputs and Read Inputs.

5. Open the Write Outputs. On the block diagram drop the I/O Table Global and logic to write to the physical outputs. This example is programmed for FieldPoint so it uses an FP Write VI. The same architecture could easily be used to program CompactRIO, PXI, or DAQ. For error handling this example included an error input and output on the front panel with connections on the VI.
connector pane.

6. Open the Read Inputs. On the block diagram drop the I/O Table Global and logic to read the physical inputs. Also add error handling.

7. Save and close the two subVIs. On the top level vi place a case structure around the Write Outputs and Read Inputs subVIs. In the false case make a new subVI called Shutdown Outputs. You will enter this case if the controller needs to shutdown.

8. Open the Shutdown Outputs.vi and add logic to write safe values to the outputs on shutdown and save the subvi. Instead of writing the output values from the I/O memory table you will write safe output values. In this case the heater should go off if we exit the program so the Output_Heater value is replaced with a constant of zero.

9. On the top level vi you now need to add logic to decide when to shutdown. Remember that your logic can shutdown the machine. This shutdown stops the controller. In most cases this should only happen if there is an error or if the controller needs to be reprogrammed. For this example we will shutdown if there is an error. We created a functional global to collect and trap errors from multiple parallel tasks.
Note: Shut down behavior is defined by the programmer. You may decide to ignore certain errors or to retry operations before shutting down.

10. In the top level vi use the functional global to detect errors and control the case structure to shutdown on an error. Note that we for each task we need to assign a Task #. We will use 0 for the initialization routine so for the I/O Scan Task we will use Task # one.

Initialization and Shutdown Routines:

1. Now we need to 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. The initialization routine will set the default values for the outputs and perform error trapping. Remember the main control loop starts by writing outputs to the hardware so we need to load startup values for the outputs into the I/O memory table.
3. Create a subVI called Default Outputs.vi and write code to set the I/O memory table to the default values. We want the default behavior of the heater output to be off so we will create a 0 constant and wire it to the Output_Heater shared variable.
4. Add error trapping with the functional global and use Task # 0.
5. On the top level vi add a sequence structure to the left of the I/O Scan Task. Wire the error cluster from the Default Outputs.vi to the border of the timed loop to assure execution order.

![Image](image1.png)

6. For the shutdown routine add a sequence structure to the right of the I/O Scan Task and wire the error cluster to the border of the sequence structure. Add any shutdown logic into the structure. In this example we have no shutdown routines.

![Image](image2.png)

7. You now have a complete initialization, I/O Scan, and Shutdown routine. Now you need to add your logic tasks.

**User Logic Tasks:**

Each logic task will be structured using a timed loop with a priority lower than the I/O scan loop and the timing an integer multiple of the I/O scan loop rate. To make your application scale and to allow you to run loops different rates than the I/O Scan task you will create a local version of the I/O. To do this you will copy I/O memory table to two local clusters (input and output) that are strict type def. A strict type def will update all instances throughout the program when you change the control. This makes it easier to modify your inputs and outputs if your program grows.

1. First a strict type def custom control is created called Input Cluster.ctl. In the custom control we add a cluster and create one controls called Input_TC.
2. Next a strict type def custom control is created called Output Cluster.ctl. In the custom control we add a cluster and create one controls called Output_Heater. **Tip: If you are going to use the StateChart Module it will create a strict type def Input Cluster.ctl and Output Cluster.ctl**
3. On the diagram create two subVIs, one called Read Inputs Local and one called Write Outputs Local.

4. Open the Read Inputs Local. On the block diagram drop the I/O Table Global and logic to read the inputs and write them to the cluster. Add error handling and modify the connector to write out the cluster and error and input an error cluster.

5. Open the Write Outputs Local. On the block diagram drop the I/O Table Global and logic to read the inputs and write them to the cluster. Add error handling and modify the connector to write
out the cluster and error and input an error cluster.

6. Save and close the two subVIs. On the top level vi place a case structure and error handling around the subVIs, note that for the Fault Merge.vi this is Task # two.

7. Now create a subVI called Logic.vi. In this vi you will wire in and out the error clusters and place your logic. Remember logic must:
   - Execute in less time than timed loop rate
   - Not directly access hardware for 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”, instead use timer functions or “Tick Count” for timing logic
   - Not perform waveform, logging, or non-deterministic operations, use of parallel loops for these operations is explained later

8. Normally we would create a subVI for encapsulation to enable reuse, however since this example is so simple PID is already a reusable subVI this step is already done for us. Simply drop a PID VI on the block diagram and wire constants so the output range is [10, 0] the PID gains are [10, 0.1, 0] and the setpoint is 350. The PID gains and setpoint could have also been wired to elements from the I/O memory table and published to an operator terminal if they needed to be changed later. Wire the Input_TC value to the “Process Variable” terminal and wire the Output_Heater value to the “Output” terminal. Add the appropriate error handling components.
Tip: StateCharts can be used for sequencing, batch, or machine control applications

9. On your top level vi wire the error cluster from the initialization routine to the border of the timed loop and save and run the application.

We can now 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.

We will now look at a more complicated example which highlights the benefits of this type of architecture.

**State Machines**
Most real control applications are far more complex than the simple PID application built in the previous example. While PID may be used to control a temperature setpoint, this will only be a small component in the application. A slightly more realistic example would be a chemical reacting vessel. In this imaginary 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
4. Once the temperature has reached 200 degrees the system will turn off the stirrers and hold the temperature constant for 10 seconds
5. Pump the contents to a holding tank
6. 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, and the time to 10 seconds. In a real application these values could be loaded from a recipe or entered by an operator.

**State Machine Overview**

The best way to represent this type of control system is to use a state machine. The state machine design pattern is one of the most widely recognized and highly useful design patterns for LabVIEW. 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.

A state machine, which is more precisely defined as a finite state machine, consists of a set of states and a transition function that maps to the next state. A full state machine should be designed to execute actions on entry, while it exists in the state, or on exit. Because we are using the state machine inside our larger machine control architecture it cannot use wait statements and cannot use loops except to retain state or perform algorithms such as a for loop used for array manipulation.

Use state machines in applications where distinguishable states exist. 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 many different actions can be performed. The actions performed can depend on previous and current inputs as well as states. A shutdown state is commonly used to perform clean up actions.

**State Machine Example in LabVIEW**
To build this application the first step is to map out the logic and I/O points. Because this application involves a sequence of steps a flow chart is a good tool for planning the application. Below is a flow chart for this application and a list of I/O signals.

Each state in a state machine performs a unique action and calls other states. State transitions depends 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

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

<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_DrainPump</td>
</tr>
<tr>
<td>Tank Empty Level Sensor</td>
<td>Input_TankEmpty_LS</td>
</tr>
</tbody>
</table>
Because one state in our machine has parallel processes we will build a second state machine to represent the parallel logic and call these in that state. We will only exit the state when both parallel processes have completed.

LabVIEW StateChart Module:

The previous example was built using basic LabVIEW structures of loops, sequences, and strict type def enums. Starting in LabVIEW 8.5 there is a new module, the LabVIEW StateChart Module. This module is a new editor for LabVIEW that allows you to quickly build full state based machine logic. It is hierarchical so that multiple state charts and LabVIEW VIs can be used together in a complex application. It also runs on Windows, real-time targets, and FPGA targets. The StateChart Module is especially useful if you need to have multiple possible transitions from a state or if you want to have specific events trigger a transition from multiple
states. For instance based on an operator command you may want to abort production from any state. The StateChart Module allows you to easily program the system to transition from any state and to give priorities to transitions so that the abort command executes even if other transitions are triggered at the same time. I will not go into a tutorial on the StateChart Module and did not use it in this example so that engineers who do not have the module can still open and run the examples. All of the features of the StateChart Module can be programmed in LabVIEW, however the StateChart module dramatically improves development time and is much easier to read and debug compared to traditional LabVIEW state machines. I strongly recommend it for anyone building machine control applications.

Quick Start – Modifying an Example

Overview

The easiest way to get started with this design is to modify an existing example. In this section we will walk through modifying the previous Chemical Mixing example that used the state chart to build our own application. There are 4 main steps

1. Modify the I/O Scan Task to read and write to the physical I/O for your application.
2. Modify the Initialization Routine to write the default values for your physical outputs.
3. Modify the Task 1 to map the I/O. You will change the StateChart, Read Inputs Local.vi, and Write Inputs Local.vi.
4. Modify /rewrite the StateChart to fit your application.

Modify the I/O Scan Task

Open the Chemical Mixing.lvproj. Since your application will use different I/O you will need to modify the I/O Scan Task to read and write to your I/O. Remember that I/O can be physical I/O, networked I/O, or simulated I/O.
1. Open the IO Table Global. This defines all the I/O for your system.

2. Change the controls in the clusters to those needed for your application. Save the IO table Global.vi with these changes.

3. Open the Write Outputs.vi and the Read Inputs.vi.

4. Modify the VIs so they write and read to your physical I/O instead of the simulated I/O. If your hardware target requires initialization to start the IO (for instance CompactRIO requires calling the Open FPGA Reference.VI) you should use the initialization routine. In your write and read you would then access the IO via the Read/Write control.

5. Open the Shutdown Outputs.vi. Modify the VI so it writes to your physical I/O instead of the simulated I/O

Modify the Initialization Routine

Since your application will use different I/O you will need to modify the Initialization Routine to set the default values for your outputs.
1. Open the Default Outputs.vi and modify it to set the default values for your outputs.

Modify the Task 1 to Map the I/O

You now have a complete I/O Scan Task and Initialization and Shutdown Routines. Now you need to modify your logic. Since each logic task creates a local copy of I/O for its execution you need to remap the I/O.

1. Under the StateChart folder open the Outputs.ctl and Inputs.ctl. Modify these to match the I/O for your application.

2. Update the Write Outputs Local.vi and Read Outputs Local.vi to read and write your I/O.
A Machine Controller Architecture Overview

Overview

The heart of most machine control applications is a controller such as a programmable automation controller (PAC) or programmable logic controller (PLC). The controller is used to receive sensor data and to control the machine using analog and digital I/O signals. Traditionally PLCs have been used in machine control and programmed using tools such as ladder logic or sequential function charts. Increasingly machine control is moving to PACs because they offer higher performance and more functionality, allowing for operations such as high-speed data acquisition and processing, as well as motion control and vision which are not supported in traditional PLCs. In order to take advantage of all the features and capabilities of a PAC, the application code must be well designed, balancing and coordinating various controller processes such as I/O, process control logic, communication to a HMI and other tasks. This document provides an overview to the architecture for a machine controller running on a CompactRIO PAC using LabVIEW Real-Time as the development tool. Most of this information is equally applicable to other LabVIEW Real-Time based PACs such as Compact FieldPoint and PXI. The controller architecture described in this document is a subsystem of the larger Machine Control Reference Architecture described in other Developer Zone Articles.

Table of Contents

1. Introduction
2. Controller Operations
3. Tag Engine
4. Machine Process Control
5. Reference Application
6. Where to Go From Here

Introduction

The controller is one of two subsystems in a machine control application, the other being the HMI. The purpose of the HMI is to display machine status information and to accept machine control inputs from the operator. Additional tasks associated with the HMI in the Machine Control Reference Architecture are alarm and event detection and logging. The controller’s responsibilities include communication and interfacing to the machine and sensors, control of the current machine state and process, and communication with the HMI. Both systems are built around a tag engine which stores current data and handles communication to other systems and the machine.
Analyzing the operations of the controller we can break down the system into smaller components, each responsible for a specific task in the overall application. The following diagram shows the controller architecture and individual components of the machine control application. Some of these components are available ready-to-run as part of the machine control reference architecture, while others must be developed as part of designing and implementing a specific machine control application.
Controller Operations

The core operations of a machine controller include I/O scans and process control. After an initialization phase, the controller will continuously read inputs, calculate control parameters for the machine process using application-specific logic based on the input data, and then update outputs to the machine. If the controller is turned off, some shutdown process may be completed.
To simplify the I/O process and access to data from different process control systems, a central data storage is commonly used that contains the current value of all tags or variables used in the application. This tag table is sometimes called a real-time database as it provides current instead of historical data. The following diagram looks at the basic controller in a more component/operations hybrid view. It adds a block for HMI communication so that data stored in the tag table on the controller can be accessed and viewed on the HMI. It also allows for multiple parallel control processes. Individual components such as I/O Scan Engine, HMI communication and process control blocks run in parallel in a multi-threaded environment. If necessary synchronization between components can be added, though for most applications this is not required if the I/O scan frequency is faster than the required control rate (loop rate of the process control blocks).

In the following section we will look in more detail at the individual components and operations of the controller and how they map to the elements of the machine control reference architecture. For more advanced information and installers for the individual components look in Developer Zone for their individual documents.

**Tag Engine**

The tag engine is the core controller subsystem. It combines the functionality of the tag table, I/O scan engine, and HMI communication components in the previous diagram. These operations in combination provide the data to the controller process logic, automating communication with the
machine and HMI. As these operations are common to most machine control applications, the specific operations of these components are typically configured for an application rather than requiring custom programming from the application developer.

**Current Value Table (CVT)**

The Current Value Table (CVT) reference library is used to implement the tag table in the controller. It provides a common mechanism for storing data that can be accessed from other system elements. Tags in the Current Value Table are name-based to make the process control logic easy to design and implement, and more readable for maintenance and support. The list of tags stored in the Current Value Table is configured using the Tag Configuration Editor (TCE), an intuitive configuration tool.

The CVT is directly accessed by the cRIO I/O Engine (CIE) and CVT Client Communication (CCC) reference libraries to automate communication with the machine and HMI respectively.

**I/O Scan Engine - cRIO I/O Engine**

In the current implementation, the I/O scan engine is provided by the cRIO I/O Engine (CIE). This reference library is used to exchange data between the NI CompactRIO Scan Engine and the Current Value Table. Channels to be used by the cRIO I/O Engine can be configured for a specific application using the Tag Configuration Editor, which is also used to configure the CVT. The FPGA can be customized with additional code to support other I/O operations in parallel to the Scan Engine and cRIO I/O Engine.

**HMI Communication - CVT Communication Client**

During the controller operation, sensor readings, machine control parameters, and other intermediate values are placed in the CVT by the I/O engine and process control blocks for use by other sub-systems. All of this data is available to the HMI for display, logging and event/alarm detection. HMI communication is provided by the CVT Client Communication reference library, which provides a network interface to the local CVT. On the HMI, data read from the controller is placed in another Current Value Table hosted on the HMI. The mapping of data from the controller CVT to the HMI CVT is configured using the Tag Configuration Editor.

**Machine Process Control**

Once the framework to handle data and I/O in the controller is in place, developing an application is a matter of configuring the Tag Engine and implementing the process control logic. This portion of the controller is customized for each application as it greatly depends on the individual machine, the processes being automated, etc. The process control blocks interface to the Current Value Table for input and output data and therefore can be developed very quickly and easily without the need to interact directly with physical I/O. Even though this portion of the controller application is very much customized to each individual application, certain common design patterns emerge across different application implementations.
State Machine – State Chart

The most common design pattern for process control implementation for machine control is some form of state machine. In a traditional state machine the process control block runs in a defined set of states that correspond to the state of the machine. As conditions change, time elapses or operator commands are received from the HMI, the process control block transitions from one state to another, updating the machine through the output values.

Using the LabVIEW Real-Time development tools you can implement state machines in a number of different ways. The most common methodology is the use of a Case structure inside of a While loop with a shift register to keep track of the state for the next iteration of the state machine. This implementation can be enhanced using a LabVIEW queue or FIFO in place of the shift register, to store multiple states and allow for one state to buffer up a series of subsequent states. Search Developer Zone for more information on building basic state machines in LabVIEW.

A derivative of the buffered state machine is a queued message handler (QMH) in which each entry of the queue or FIFO is considered an event or action and is only processed one time. The processor does not stay in the current state unless the event or action is placed repeatedly into the queue. If no event is present in the queue, no action is performed by the processor. A template of a queued message handler implementation is included in the Asynchronous Message Communication (AMC) reference library.

The LabVIEW State Chart module is an advanced tool for building complex state machines. Using state charts you can easily design parallel and nested state diagrams that can be executed on LabVIEW or LabVIEW Real-Time targets. You can assign LabVIEW code to each transition between states, and also place condition checking (guards) into the states and transitions. This tool can easily handle the implementation of industry standard state diagram such as the OMAC PackML standard for machine control. An implementation of this standard is offered as A State Chart Design Pattern for Machine Control. Additional information on using the LabVIEW State Chart module is available in Developing Applications with the LabVIEW State Chart Module.

High Speed I/O, Control and Processing

One of the biggest benefits of using a PAC is the ability to add other tasks to your application, which could not be handled by a traditional PLC. When using CompactRIO as your controller this may include high-speed data acquisition, motion control, hardware-based closed loop control in the microsecond range and more.

Using CompactRIO and its FPGA technology we have the ability to configure much more advanced functionality on the FPGA and in the LabVIEW real-time application than the cRIO I/O Engine described earlier. In parallel to the Scan Engine and cRIO I/O Engine, you can implement high-speed data acquisition from a different analog input module and stream the data using a DMA channel to a separate task on the controller running in LabVIEW Real-Time. On both the FPGA and the controller this code runs in parallel to the existing architecture and has no effect on the implementation or the configuration of the machine control application. Using high-
speed analog data we can add vibration analysis, machine condition monitoring, frequency analysis and more to the machine control application. These additional tasks can reduce or eliminate the need for additional measurement equipment and by integrating these operations with the main machine control application better and more efficient control of the machine can be implemented.

Adding motion control directly to the machine controller allows further integration of different operations which have been traditionally handled by separate applications and equipment. Using the NI SoftMotion tools combined with the C series motion drive and digital I/O modules, high speed closed loop control of servo and stepper motors can be added. The FPGA target allows implementation of the motion controller with the reliability and determinism only offered by a hardware-based solution, while the LabVIEW RT based SoftMotion trajectory generator can create multi dimensional contours and blended moves to satisfy advanced motion control requirements. Like the high-speed DAQ operation, this functionality works in parallel to the fundamental machine control application.

Any additional task such as high-speed acquisition and motion control can interact with the rest of the machine control application using the Current Value Table and thus have access to the same data and information. Other communication paths can be added between these different components and the machine control application using queues, FIFOs, and notifiers. High-speed communication from these tasks back to the HMI can also be implemented in parallel to the CVT Client Communication interface.

Reference Application

A basic implementation of the machine control reference architecture is shown using a simple bioreactor tank control reference application that is available on Developer Zone as part of the architecture components. The tank control reference application integrates all of the components of the machine control reference architecture, so it can be used as a template for building other machine control applications using the architecture. It serves as a guide for developing an application but as with any development, customization of code and enhancements to the provided architecture and tools may be necessary to implement your specific application. Both the architecture and reference libraries should serve as a starting point and guide in development but should not be considered the only possible solution for a particular application.

Where to Go From Here

To learn more about LabVIEW-based machine control, we suggest that you read the following document:

A Reference Architecture for Local Machine Control - This is a reference architecture for building a local machine control system based on an NI Touchpanel Computer (TPC) HMI and a cRIO controller.
The following is a map of other documents that describe the machine control reference architecture. You can click on the image to navigate directly to each document.

[Your user agent does not support frames or is currently configured not to display frames. However, you may visit &lt;a href="ftp://ftp.ni.com/pub/devzone/tut/mca_navigation1.html"&gt;the related document.&lt;/a&gt;]