Application Design Patterns: State Machines

Publish Date: Sep 08, 2011

Overview

The State Machine is one of the fundamental architectures LabVIEW developers frequently use to build applications quickly. State Machine architecture can be used to implement complex decision-making algorithms represented by state diagrams or flow charts. More precisely, a state machine implements any algorithm described by a “Moore machine,” which performs a specific action for each state in the diagram.

State Machines are used in applications where distinguishable states exist. Each state can lead to one or multiple states, and can also 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 many different actions can be performed. The actions performed can depend on previous and current inputs as well as states. A “shutdown” state can then be used to perform clean up actions.

State Machines are most commonly used when programming user interfaces. When creating a user interface, different user actions send the user interface into different processing segments. Each of these segments will act as states in the State Machine. These segments can either lead to another segment for further processing or wait for another user event. In this example, the State Machine constantly monitors the user for the next action to take. There is another design pattern that can be used to implement a user interface, the Queued Message Handler. A Queued Message Handler is a more sophisticated version of the State Machine and offers additional flexibility, but it also adds additional complexity. Queued Message Handlers will be covered at another time.

Process testing is another common application of State Machines. In this example, each segment of the process is represented by a state. Depending on the result of each state’s test, a different state may be called. This can happen continually, performing in-depth analysis of the process being tested.

Table of Contents

1. Why Use a State Machine?

Besides its powerful ability to implement decision making algorithms, state machines are also functional forms of application planning. As the complexity of applications grow, so does the need for adequate design. State diagrams and flowcharts are useful and sometimes essential for the design process. Not only are State Machines advantageous in application planning, they are also easy to create. Creating an effective State Machine requires the designer to make a table of possible states. With this table the designer can plan how each state is related to another. The design process involved in creating an operative State Machine will also improve the overall design of the application.

2. Build a State Machine

We want to generate an application that fires a cannon continuously without allowing it to get dangerously hot. To begin, we will represent this logic with a state diagram (Fig.1). Notice that in this diagram, the states (oval nodes) describe the actions that are performed when the control process is in that state, whereas the transitions (arrows) simply describe when and how the process can move from one state to another.

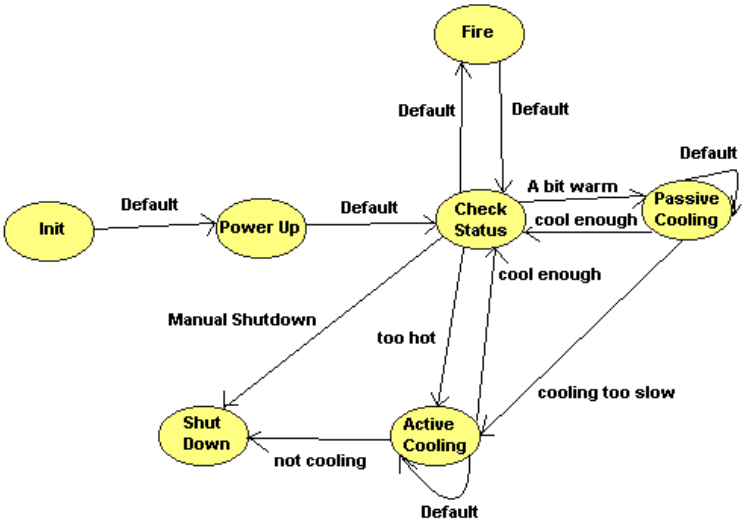

Figure 1: State Diagram

Each state in a State Machine does something unique and calls other states. State communication depends on some condition or sequence. To translate the state diagram into a LabVIEW diagram, you need the following infrastructure:

- While loop – continually executes the various states
- Case structure – each case contains code to be executed for each state

- Shift register – contains state transition information
- Transition code – determines the next state in the sequence (see below for examples)

The flow of the state diagram (Fig.1) is implemented by the loop, while the individual states are replaced by cases in the case structure. A shift register on the while loop keeps track of the current state, which is fed into the case structure input.

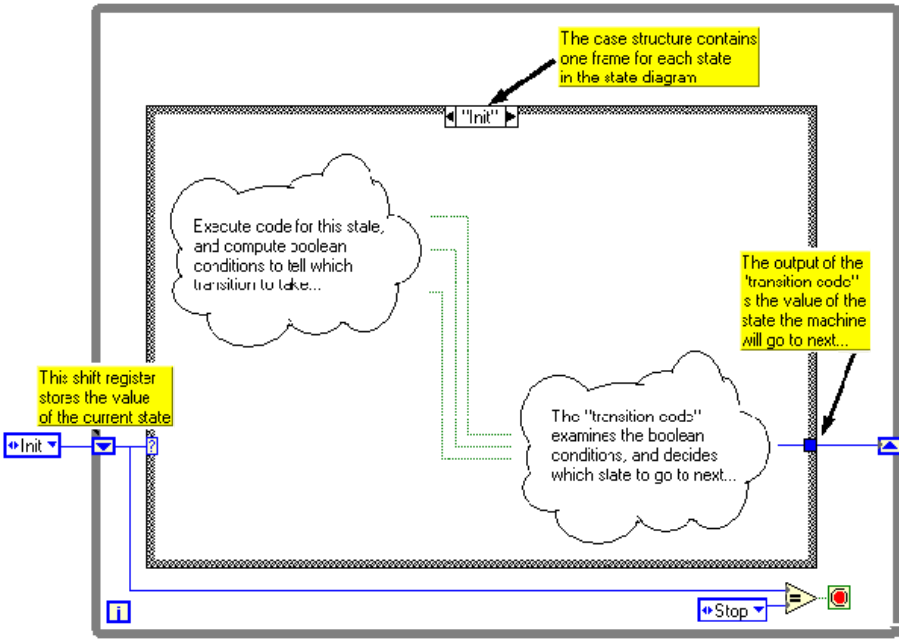

Figure 2: State Machine

There are different methods to determine which state to transition to next. Four common methods are discussed below (Note – in the following method examples, “Init” could transition to any state):

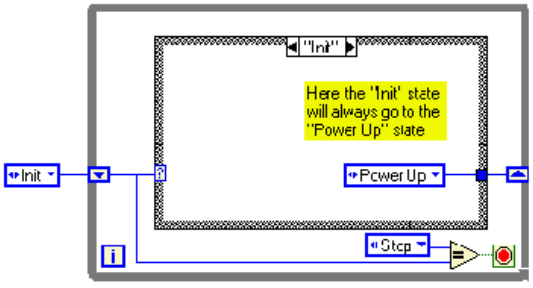

Figure 3a

Figure 3a demonstrates an “Init” state that has only one possible transition.

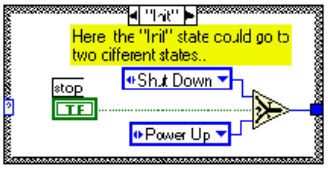

Figure 3b

Figure 3b demonstrates an “Init” state with two possible transitions, “Shut Down” or “Power Up”.

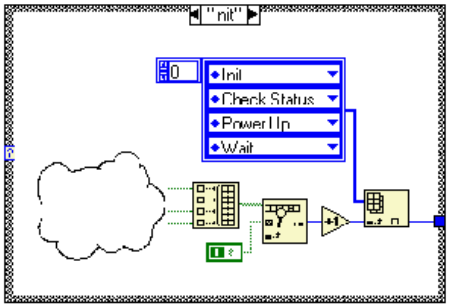

Figure 3c

Figure 3c demonstrates an “Init” state using a boolean array along with an array of enum constants. There is a boolean in the boolean array corresponding to each transition “Init” can make. The array of enum constants represents each transition name. The index of the first “True” boolean in the boolean array corresponds to the index of the new state in the array of enums.

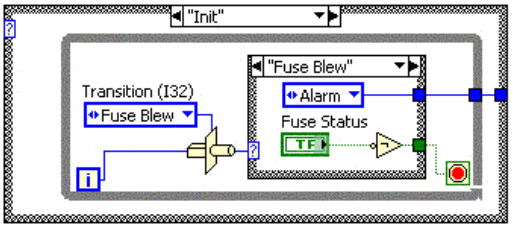

Figure 3d

Figure 3d demonstrates an “Init” state using an inner loop and case structure to transition to the next state. The inner case structure contains one diagram for each transition that leaves the current state. Each of the cases in the inner case structure has two outputs – a boolean value, which specifies whether or not the transition should be taken, and an enumerated constant, which specifies the state to which the transition goes. By using the loop index as an input to the case structure, this code effectively runs through each transition case one by one, until it finds a diagram with a “True” boolean output. After the “True” boolean output is found, the case outputs the new state to which the transition goes. Though this method may appear slightly more complicated than the previous methods, it does offer the ability to add names to transitions by “casting” the output of the loop index to an enumerated type. This benefit allows you to add “automatic documentation” to your transition code.

3. Example - Coke Machine

This application has the following requirements:

- All Coke products are sold for 50 cents.
- The machine only accepts nickels, dimes, and quarters.
- Exact change is not needed.
- Change can be returned at anytime during the process of entering coins.

Our first step will be to create a state diagram. There are quite a few ways this can be done, but remember that more states equals less efficiency. For this example, our state diagram will have eight states. The states we will be using are:

1.) INIT – initialize our Coke Machine
2.) WAIT FOR EVENT – where the machine waits for coins
3.) RETURN CHANGE – where the machine returns change
4.) COKE PRODUCT – when the machine has received 50 or more cents it will dispense the beverage
5.) QUARTER – when the customer enters a quarter
6.) DIME – when the customer enters a dime
7.) NICKLE – when the customer enters a nickel
8.) EXIT – after the change is returned and/or beverage dispensed, the machine will power down (application will terminate)

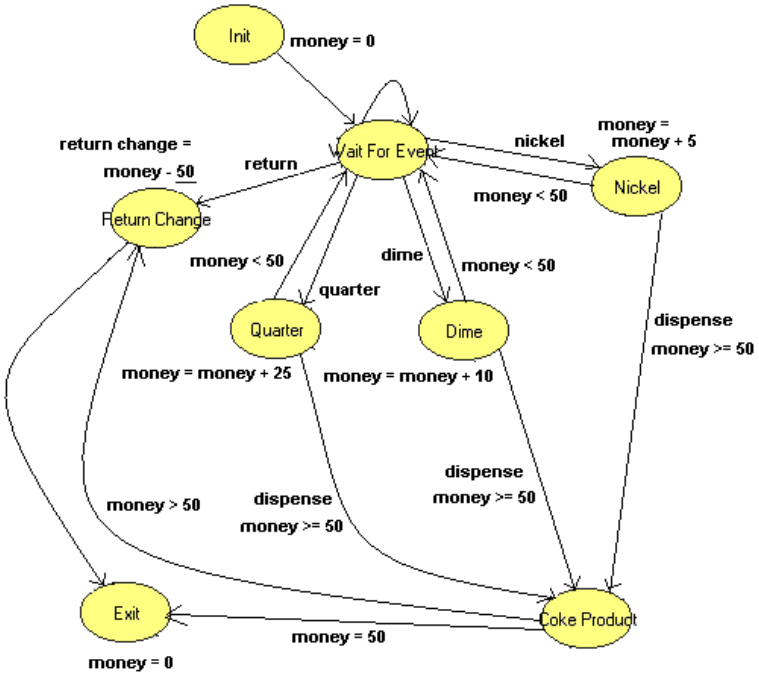

Figure 4

With the state diagram complete, we are now ready to begin our LabVIEW State Machine application. To view the final State Machine, please open the attached VI.

4. IMPORTANT NOTES

There are some caveats to be aware of when creating a State Machine, i.e. code redundancy and use of Enum.

- Code redundancy

Problem: The hardest part of creating a State Machine is to differentiate between possible states in the state diagram. For example, in the Coke Machine state diagram (Fig.4), we could have had 0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50 cent states rather than having a “wait for response” state that goes from one state to another depending on which type of coin is dropped. That would create 11 different states with the exact same case diagram. Redundant code can create a big problem in a larger application.

Solution: If different states have the same case diagram, try to combine them into one state. For example, the “wait for response” state is created in order to avoid code redundancy.

- Enum use

Problem: Enums are widely used as case selectors in State Machines. If the user attempts to add or delete a state from this enum, the remaining connected wires to the copies of this enum will break. This is one of the most common obstacles when implementing State Machines with enums.

Solution: Two possible solutions to this problem are:
1. If all the enums are copied from the changed enum, the breaks will disappear.
2. Create a new control with the enum, and select “typedef” from the submenu. By selecting typedef, all of the enum copies will be automatically updated if user adds or removes a state.