Function pointers, Part 3: State machines

Jacob Beningo - February 12, 2013

Function pointers can be used for a wide variety of applications including the implementation of state machines. Just like any tool, it may not always be appropriate to use function pointers for a state machine implementation.

A couple of common methods for implementing state machines is either the use of if/else statements or the use of switch/case statements. In general, it can be a good idea to use the function pointer implementation if the state machine has a lot of states or if there is a lot of code associated with each state that would cause the complexity of the state machine function to be high.

Figure 1 contains an example state machine that will be implemented within this article. It contains only four states and each state only has the ability to transition into one other state. This will allow a simple examination of how to implement the function pointer version of a state machine. However, this process can be used for any state machine with much more complex transitions and states.

The first step to implementing the state machine is to create an enumeration of each state in the machine. Listing 1 shows the example enumeration that has been defined using a typedef and given a label of StateType. It contains all four states in addition to NUMSTATES which provides the number of states in the state machine. Remember enumerations start at 0!

Figure 1 Example state machine
The next step in defining the state machine will be to define the structure for the state machine. This structure should be capable of storing the state (STATE_A thru STATE_D) in addition to being able to store a function pointer to that state.

The result of the structure is that an array can be created that holds both the state and the function that should be executed when the state machine is in that state. Now it is possible to skip the StateType element of the structure but when creating the state machine table it allows the table to be much easier to read and doesn’t affect system performance other than using a couple of extra bytes of flash. **Listing 2** shows the definition of the StateMachineType structure.

It should be noted that in this definition of StateMachineType the function pointer neither takes nor returns anything. For a fictitious machine this is fine but in the real world each state may take a certain type of variable or return one. The structure would be modified to meet the requirements of the state machine.

Once the structure has been defined, an array of type StateMachineType would be defined that contains not only each state but also the actual function that should be called for that state. This can be seen in code **Listing 3**.

It is important to keep in mind that the functions, such as Sm_StateA, should at least be prototyped before this table definition. If it is not, then the compiler will most likely give a warning or a full error that it cannot find the definition. In addition to the function prototypes, it will be necessary to define a state variable to store the current state of the machine. **Listing 4** shows both the variable definition and the function prototypes.
There are a couple of different ways the actual code for the state functions could look depending on the requirements. First, the state function could run the state code and then update the state variable (SmState) to the next state.

Alternatively, it could be that the state code runs over and over until some external event occurs that then causes it to perform the state transition and update the state variable. This is highly dependent on the application and the defined transition. For simplicities sake, this example will simply run each states code followed by a forced transition. The function definitions can be found in code `Listing 5`. 

```c
/*
 * A table that defines the valid states of the state machine and
 * the function that should be executed for each state.
 */
StateMachineType StateMachine[] =
{
    { STATE_A, Sm_StateA },
    { STATE_B, Sm_StateB },
    { STATE_C, Sm_StateC },
    { STATE_D, Sm_StateD }
};

Listing 3 State machine table

Listing 4 State variable and functions
```
Finally, with the entire framework in place for the state machine, a function needs to be defined that actually runs the state machine. In general, the author likes to use obvious naming conventions so in this case the function will be Sm_Run as can be seen in code Listing 6.

In the Sm_Run function, a simple check is performed to make sure that the current state of the state variable has not exceeded the number of states. If it has, then an exception can be thrown, otherwise the function pointer to the current state function is called, thereby running the state.

Listing 5 State machine functions
/*
 * Call this function to run the state machine
 */

void Sm_Run(void)
{
    // Check to make sure that the state that is being entered
    // is valid.
    if (SmState < NUM STATES)
    {
        // Call the function for the state
        (*StateMachine[SmState].func)();
    }
    else
    {
        // Throw an exception
    }
}

Listing 6 State machine run function

Observe that the notation used to call the state function is the same as has been used in previous posts and how simple the state machine code is. Each state is nicely broken up into individual functions that will make maintenance easier in the future due to minimization of complexity.

This code could have all been implemented into a single function with an if/else statement but the complexity of the code would be difficult especially if any of the states had a page worth of code associated with it which wouldn’t be uncommon. In addition, adding addition states doesn’t require a change to Sm_Run function! All one needs to do is add the state to the enumeration, define a function and update the state machine table.

*Jacob Beningo* is an embedded software consultant who currently works with clients in more than a dozen countries to dramatically transform their businesses by improving product quality, cost and time to market. Feel free to contact him at jacob@beningo.com, at his website [www.beningo.com](http://www.beningo.com), and sign-up for his monthly Embedded Bytes Newsletter [here](http://here).

**Related articles:**

- State machines ease programming microcontrollers
- Using finite state machines to design software
- Function pointers: An introduction
- Function pointers, Part 2: Task scheduling