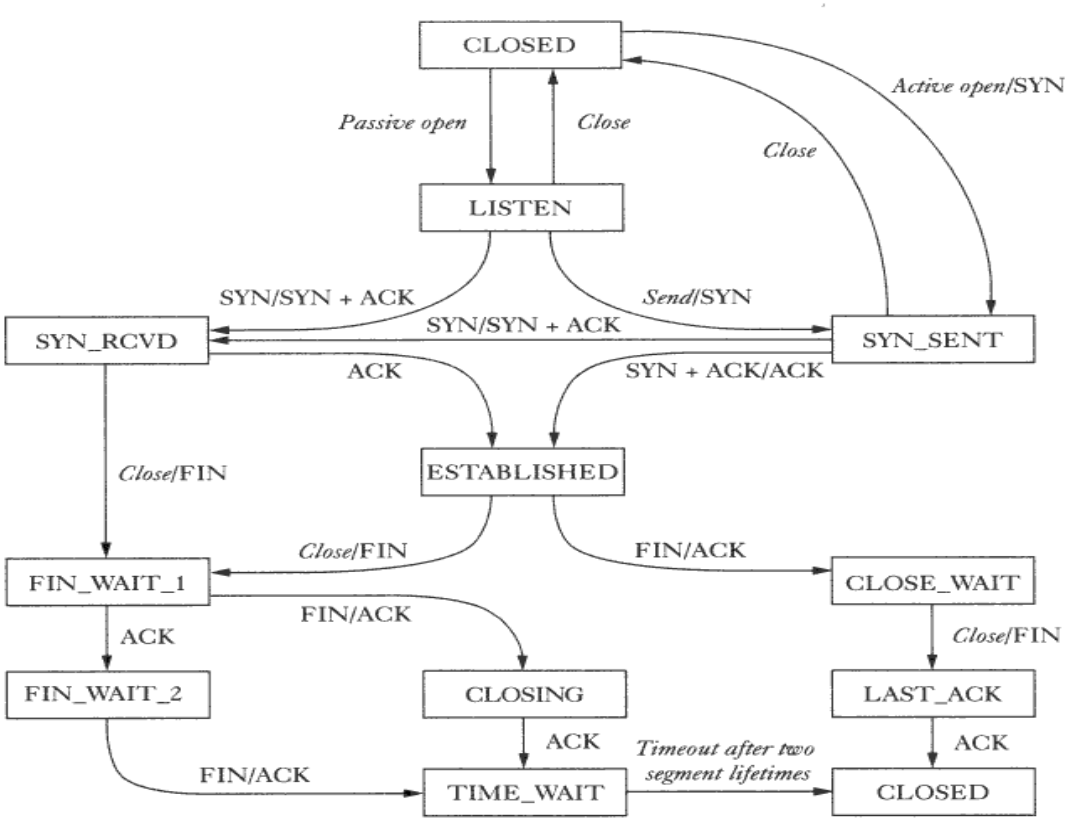
A Back to Basics Finite State Machine for Mission Critical Applications

Marcel-Titus Marginean mtm@mezonix.com

Finite State machine is a very common design patters and a large variety of implementation exists from simple switch / case statements to full blown hierarchical state machines implementation. A typical state machine like the one used by TCP protocol and presented in the diagram bellow it is represented as a graph where the nodes are states and the oriented edges are valid transition. Each edge is labeled with a pair Event/Action specifying the event that triggers the transition and the action taken when the transition happen. If there is no action required only the event is marked.



TCP State Machine

Most of the implementations of a FSM make use of a classic State Pattern, modeling each state as a separate class derived from a State Base Class. While this design pattern it is very powerful enabling complex designs it is often avoided by many embedded / real-time engineers due to a few reasons:
- Perceived complexity of the design when in most places where a state machines is needed the problems are relative simple, for example a motor with few states, a packet builder from a byte stuffed serial data stream etc. The fact that each state is a separate class appears an an overkill in this situations and often embedded engineer fall back on the error prone practice of switch/case statements. A simpler FSM having all the processing compact into a single class is often preferred for this kind of applications.
- The codding of the transitions is often distributed across states therefore lacking a bird-eye view of the whole transition table for easy audit. Audit is essential in Mission Critical applications. A design

that allows all the information defining the state machine to be grouped together for easy audit would be preferred. It is even better if the programmer implementing a FSM is allowed to focus exclusively on the problem at hand minimizing the focus he have to pay to “internal plumbing” i.e. implementation details of the pattern.
- Some common design examples floating around the Internet employs a Java-ish approach to state transition deleting and dynamically reallocating states objects and this is undesirable in Mission Critical and Real-Time Embedded applications where engineers prefer to allocate all required memory at system start-up and delete it upon termination. A design which have no other memory allocation / de-allocation outside of constructor / destructor is preferred.

The BbFSM (Back to Basics FSM) attempts to address these issues while also providing a few additional tools to help catch a few common mistakes, like codding multiple transitions triggered by the same event from a given state or passing invalid values in defining a Finite State Machine.

Make it easy to inspect

A state machine is fully described by four pieces of information:

List of states: CLOSED, LISTEN, SYN_RCVD, SYN_SENT, ESTABLISHED, FIN_WAIT_1, FIN_WAIT_2, CLOSING, TIME_WAIT, CLOSE_WAIT, LAST_ACK

List of events: PasiveOpen, ActiveOpen, Close, SYN, SYN_ACK, Send, ACK, FIN, FIN_ACK, Timeout

The initial State upon creation: CLOSED

The Transition table:

From State	Trigger Event	To State	Action
CLOSED	PasiveOpen	LISTEN	None
CLOSED	ActiveOpen	SYN_SENT	Send Syn
LISTEN	SYN	SYN_RCVD	Send Syn Ack
LISTEN	Send	SYN_SENT	Send Syn
SYN_RCVD	ACK	ESTABLISHED	None
SYN_RCVD	Close	FIN_WAIT_1	Send Fin
SYN_SENT	Close	CLOSED	None
SYN_SENT	SYN_ACK	ESTABLISHED	Send Ack
ESTABLISHED	Close	FIN_WAIT_1	Send Fin
ESTABLISHED	FIN	CLOSE_WAIT,	Send Ack
CLOSE_WAIT	Close	LAST_ACK	Send Fin
LAST_ACK	ACK	CLOSED	None
FIN_WAIT_1	ACK	FIN_WAIT_2	None
FIN_WAIT_1	FIN	CLOSING	Send Ack
FIN_WAIT_1	FIN_ACK	TIME_WAIT	Send Ack

From State	Trigger Event	To State	Action
FIN_WAIT_2	FIN	TIME_WAIT	Send Ack
CLOSING	ACK	TIME_WAIT,	None
TIME_WAIT	Timeout	CLOSED	None

BbFSM design goal is to be able to group those pieces of information together into an obvious piece of code to allow for easy code inspection during code reviews. The usage of BbFSM enable us to achieve this goal easily while declaring the state machine class:

```
class TCP_StateMachine: public BbFSM{
public:

 enum States {CLOSED, LISTEN, SYN_RCVD, SYN_SENT, ESTABLISHED, FIN_WAIT_1,
              FIN_WAIT_2, CLOSING, TIME_WAIT, CLOSE_WAIT, LAST_ACK};
 enum Events {PasiveOpen, ActiveOpen, Close, SYN, SYN_ACK, Send,
              ACK, FIN, FIN_ACK, Timeout};

  TCP_StateMachine():BbFSM(CLOSED){  // initial state is closed.

//      From State, Triggering Event, To State, Action (i.e. method to call)

    TRANSITION(CLOSED,PasiveOpen,   LISTEN,   None);
    TRANSITION(CLOSED,ActiveOpen,   SYN_SENT, &TCP_StateMachine::SendSyn);
    TRANSITION(LISTEN,SYN,          SYN_RCVD, &TCP_StateMachine::SendSynAck);
    TRANSITION(LISTEN,Send,         SYN_SENT,     &TCP_StateMachine::SendSyn);
    TRANSITION(SYN_RCVD,ACK,        ESTABLISHED,  None);
    TRANSITION(SYN_RCVD,Close,      FIN_WAIT_1,   &TCP_StateMachine::SendFin);
    TRANSITION(SYN_SENT,Close,      CLOSED,       None);
    TRANSITION(SYN_SENT,SYN_ACK,    ESTABLISHED,  &TCP_StateMachine::SendAck);
    TRANSITION(ESTABLISHED,Close,   FIN_WAIT_1,   &TCP_StateMachine::SendFin);
    TRANSITION(ESTABLISHED, FIN,    CLOSE_WAIT,   &TCP_StateMachine::SendAck);
    TRANSITION(CLOSE_WAIT,Close, LAST_ACK,      &TCP_StateMachine::SendFin);
    TRANSITION(LAST_ACK, ACK,     CLOSED,       None);
    TRANSITION(FIN_WAIT_1,ACK,    FIN_WAIT_2,   None);
    TRANSITION(FIN_WAIT_1,FIN,    CLOSING,      &TCP_StateMachine::SendAck);
    TRANSITION(FIN_WAIT_1,FIN_ACK, TIME_WAIT,   &TCP_StateMachine::SendAck);
    TRANSITION(FIN_WAIT_2, FIN,    TIME_WAIT,   &TCP_StateMachine::SendAck);
    TRANSITION(CLOSING, ACK,      TIME_WAIT,    None);
    TRANSITION(TIME_WAIT, Timeout,      CLOSED, None);
  }

protected: // Declaration of actions / methods

  void SendSyn(States from, Events evt, States to, void *, size_t);
  void SendSynAck(States from, Events evt, States to, void *, size_t);
  void SendAck(States from, Events evt, States to, void *, size_t);
  void SendFin(States from, Events evt, States to, void *, size_t);

  virtual void ERROR_EventReceivedInTerminalState(int from, int evt,
                                                  void *data, size_t dataSz);
};
```

Built-in safety features

The required parameter of the BbFSM constructor forces the user to provide the initial state, therefore making sure the state machine is always initialized properly. After the constructor is finished, no other memory allocation/deallocation happen until the destructor is being called. As a result making the state machine member into a class allocated at startup and deleted at program termination is compliant with recommended Embedded RT practices.
The base class BsFSM define the ***ERROR_EventReceivedInTerminalState*** method as pure, therefore forcing the programmer to implement it. This method is called whenever the state machine receive events after reaching a terminal state, i.e. a state from which no transition out exists. While in theory there is absolutely nothing wrong with a terminal state in state machine design, in embedded systems where a piece of software is expected to run for months or years running into a terminal state it is rather odd. The reason why the framework forces the programmer to create a body for this function is to make sure that he is aware of this states existence. If the programmer knows that reaching a terminal state is normal the body of the function can be empty and nothing happen, but if this is a state machine that is not supposed to reach a terminal state this method can attempt to implement some disaster mitigation code or at minimum log the unexpected result in the hope that the problem will be seen during testing.
A similar method **WARNING_UnhandledEventByCurentState** is called whenever an event is received for which no transition exists from current state. However this method is implemented as an empty body in base class and the programmer can re implement it only if is needed. Something like this is expected to happen without being an error (for example the impatient user presses start button twice or more while the device is initializing). Therefore the framework does not coerce it's implementation if not desired. In the example above this warning method is not overridden.

The **TRANSITION** macro it is used by the framework to add entries into the transition table (practically a map of maps). The reason for this being a macro which in turn call the **transition**(...) methods of the base class is to be able to enforce a set of safety features. The macro implementation beside calling the **transition** it also enforce compile time checks to not have two transitions from the same state triggered by the same event which is a mistake in FSM design. It also prevents using a numeric value instead of a state or event name as well as the mistake of reversing a state with the event by triggering a compilation error in all these cases. In order for this macro to work the **States** and **Event** enumerations must have these names. That is, the States enumeration must be names **States** and Events enumeration must be named **Events**. This safety feature as matter of fact already caught an error in the implementation of this example, originally I was looking at the transition TCP diagram from the url: http://www.texample.net/media/tikz/examples/PNG/tcp-state-machine.png
which contains a error. In that diagram the transition from FIN_WAIT_1 to CLOSING is represented as an ACK instead of an FIN / ACK. When I implemented the transition as

```
TRANSITION(FIN_WAIT_1,  ACK,        CLOSING,   None);
```

I got a compiler error because the table entry conflicted with the one above. This made the mistake evident and proved that the safety feature worked as intended. It was just that the error caught this time has not been a programmer error but an error in the specification document.

Implementation details

The class BbFSM define the state machine and each implementation inherits from this one as presented in the example above. The states transition table is stored into a map of maps while the current state is stored in **currentState** variable used to look-up into the first level map. The first level map is indexed with the **From** state and failure to lookup **currentState** in this outer level map triggers the **ERROR_EventReceivedInTerminalState**(...) handler to be called. The value type of this map is the second level map, indexed by the **Event** id. Failure of lookup at this level trigger the **WARNING_UnhandledEventByCurentState**(...) method to be called. If found, the value type in this second level map it is a pair where the first element is the **To** state id and the second element is a pointer to an instance of a **Caller** object which holds a pointer to the method registered for this transition and is able to call it when the transition happen.
The **transition(...)** methods are used to register the methods for a given **From**-**Event** transition. If **None** member variable is passed, a **NULL** pointer instead of the caller pointer is registered. The state transition will take place but no callback method is being notified. The macro **TRANSITION** is using the names passed as parameters to declare three constants:
const States State_<FromState>Event<EventName> with value equal with <ToState>
const States State<FromState>Event<EventName> with value equal with <FromState>
const Events State<FromState>Event_<EventName> with value equal with the event value
This way any entry that have the same From_State and Event as another one will generate a compilation errors because it redefines an already existing constant. Swapping the event with a state will also trigger a compilation error because it will attempt to assign an event value to a state constant or a state to an event. We pass to the transition() method the values assigned to these constants instead of the original macro parameters in order to quiet the compiler warnings.
To trigger a transition an event is being posted with **putEvent(...)** method, which can also take two optional parameters in the form of a **void pointer** and an **size_t** value. The framework does not interpret these two extra parameters in any way, they are just passed to the appropriate handler. The putEvent method from the base class takes an integer instead of **Events** as most likely it will be called from a deserialized value from the network or some hardware. It is however always possible to create a postEvent in the derived class which takes an event and call in turn the putEvent from the main class, if the events will always be posted from hardcoded statements instead of a variable set by unrelated code. With implementation of the framework having well under 100 lines of code it can be easily understood modified and audited.