

EMBSYS 110 Module 4

Design and Optimization of Embedded and Real-time Systems

1 Finite-State-Machine (FSM)

Finite state-machine is the foundational concept in the theory of computation. It can be viewed as a mathematical model of computation. Outside academia it is very popular in certain fields of embedded systems such as telecommunications or anything that involves protocols. Apart from that it is surprisingly not commonly used by many software engineers, even in UI development where the concept of state is dominant.

Someone would say state-machine adds complexity, as a traditional state diagram can look quite complicated with numerous states and transitions interwoven like a nest. It's both true and false. It is true that a traditional state diagram due to its infamous state explosion problem fails to express any realistic design in a neat way. Later we will see how statecharts solve this problem with some innovative and intuitive concepts. It is false though that it increases the complexity of your design. Any complexity is inherent in a design. All it does is to reveal the complexity so we can visualize it, discuss it and reason about it. Quite the contrary, when you find your state diagram over-complicated, you will likely find a way to simplify it and hence yield a simpler design.

1.1 FSM Notation

What is a finite state-machine? Any textbook on the theory of computation will likely give you a *formal* definition of FSM in the first chapter. Typically it is expressed as a tuple of a finite set of states (S), a finite set of events (E) and a transition function ($S \times E \rightarrow S$).

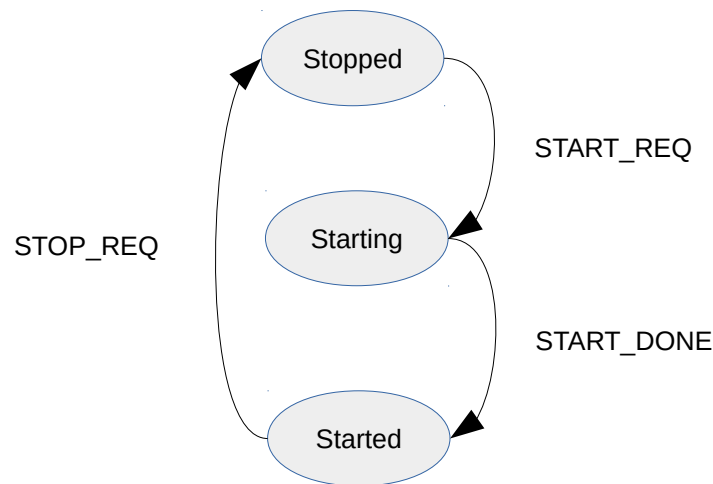
In plain programming terms, you will have an enumeration of *State* and an enumeration of *Event*. *State* lists all the possible situations or conditions that your software component is concerned of when reacting to an event. *Event* lists all the different types of signals that your software component may receive, upon which some actions *may* be taken along with a *possible* state change (called transition).

This is a simple example:

```
enum State {  
    STOPPED,  
    STARTING,  
    STARTED  
};
```

```
enum Event {  
    START_REQ,  
    START_DONE,  
    STOP_REQ  
};
```

The corresponding state diagram is shown below. There is no standard convention of a traditional state diagram. Typically we use a circle or eclipse to represent a *state*, a label for an *event* and a directed line for a transition.



Once a state diagram is drawn, software developers implement it in executable code like C/C++. There are several common ways to do that and we will show them in the next section.

1.2 Philosophy

Before moving on let's pause for a moment to reflect on what it is so special about *state*. Even without hierarchical states and all sorts of cool features added in statecharts, having a simple enumeration of states (*enum State*) alone brings a huge improvement. *State* exists in our software without us thinking about it. Ultimately what software does is to take some actions upon inputs it receives. The actions it takes depend on not only the inputs themselves but also the current conditions of the software. What do we use to represent such conditions in software?

At the bottom layer it is bits of zeros and ones of course (until we move on to quantum computers). At a higher level, we abstract related bits into bytes or words, and eventually into variables, structures or objects. Let's focus on variables, since objects can be thought of as the member variables it contains. To put all these together, we can say your software determines what actions to take depending on some combinations of the current values of variables.

We aren't actually saying anything new here, since we all know from our software experiences that programming involves (i) defining variables and (ii) use conditional constructs like `if-else`,

switch-case, while and for together with logical operators like &&, || to check on some combinations of the current values of variables. The issue is just a single 32-bit variable can store 2^{32} possible values or states. If you combine multiple of them together in a logical expression, the total number of possible combinations or cases to *consider* is going to be order of magnitude higher. Have you seen conditional code like this before?

```
if ((battLevel >= THRESHOLD_1) && (battLevel < THRESHOLD_2) &&
    !((temperature < TEMP_LOW) || (temperature >= TEMP_HIGH)) &&
    (usbHal.IsPluggedIn() && !dcCharger.IsPluggedIn()) &&
    (usbHal.ReadRegister(OVER_CURRENT_ERROR_COUNT) < 3) &&
    (!chargingStarted) {
    // Starts CC USB charging.
    usbHal.StartCharging(CONSTANT_CURRENT_MODE);
    chargingStarted = true;
    ....
}
```

When looking at a particular conditional statement individually we might still understand what will happen under certain specific conditions. In the example above we could understand under what conditions would USB charging be started. However if we *just* do that, we are ignoring all the other conditions. We can't tell if something else is going to happen when the condition above is evaluated to be false, and if so what those actions will be. Either the design is incomplete if it hasn't considered all possible cases, or those other cases are scattered or buried somewhere else in the source code which could be very hard to recover to form a complete picture.

Fundamentally it is a problem of abstraction. Out of the possible 2^{32} possible *cases* (values) of a single integer variable, only a handful of those values or (ranges of value) are meaningful to us. Defining *states* allows us to think at a higher level of abstraction, i.e. to reason about a relatively small discrete state-space rather than a much larger continuous variable-space.

A state diagram makes it very clear and explicit when state is changed and what it is changed to. If we use a combination of variables or flags to represent state implicitly we need to be concerned of when each one of those variables or flags is set, cleared and tested. As they tend to be scattered in the source code, it can be a daunting task especially when the author is somebody else.

1.3 Traditional Implementation

See *Chapter 3 Standard State Machine Implementations* of the PSiCC book for details. This is a summary of those approaches.

1. Double-switch

Use the first level of switch-case to check states and the second level to check for events. A simplified example is shown below:

```

void Dispatch(Event event) {
    switch (currentState) {
        case STOPPED: StoppedStateHandler(event); break;
        case STARTING: StartingStateHandler(event); break;
        case STARTED: StartedStateHandler(event); break;
    }
}

void StoppedStateHandler(Event event) {
    switch(event) {
        case START_REQ:
            DoStart();
            currentState = STARTING;
            break;
        case START_DONE:
            LOG("Unexpected event START_DONE in state STOPPED.");
            break;
        case STOP_REQ:
            LOG("Ignored event STOP_REQ in state STOPPED.");
            break;
    }
}

```

2. State table

Rather than having state and event checks in explicit switch-case statements, we encode the *business logic* in a data table so the code itself becomes simpler and generic. The underlying idea is similar to the double-switch approach. There are a few variations of how much information is encoded in the table. Some approaches encode just the handler function for each combination of the current state and event while others include a guard condition and the next state. The example below illustrates the minimal approach:

```

typedef void (*Handler)(Event event);

void StoppedStateStartReq(Event event) {
    DoStart();
    currentState = STARTING;
}
void UnexpectedEvent(Event event) {
    LOG("Unexpected event %d in state %d", event, currentState);
}
void IgnoredEvent(Event event) {
    LOG("Ignored event %d in state %d", event, currentState);
}
...

```

```

const Handler table[STATE_COUNT][EVENT_COUNT] = {
    { StoppedStateStartReq, UnexpectedEvent,      IgnoredEvent      },
    { IgnoredEvent,        StartingStateStartDone, UnexpectedEvent  },
    { IgnoredEvent,        UnexpectedEvent,       StartedStateStopReq }
};

void Dispatch(Event event) {
    table[currentState][event](event);
}

```

3. State pattern.

This approach comes from the *State Pattern* in the classic **GoF** book on object-oriented design patterns. Each state is represented by a subclass derived from a common base class for all states. Each event is handled by a virtual function declared in the base class, allowing each individual state to customize its handling of the event by overriding the virtual function in its own subclass.

A pointer to the base class points to the subclass object representing the current state. An event is dispatched to it by invoking the corresponding virtual function via that base class pointer. This is an elegant approach. However due to the lack of support for state hierarchy, it is overshadowed by more modern approaches including QP.

2 Hierarchical State-Machine (HSM)

2.1 Statechart Notation

2.1.1 An Anecdote

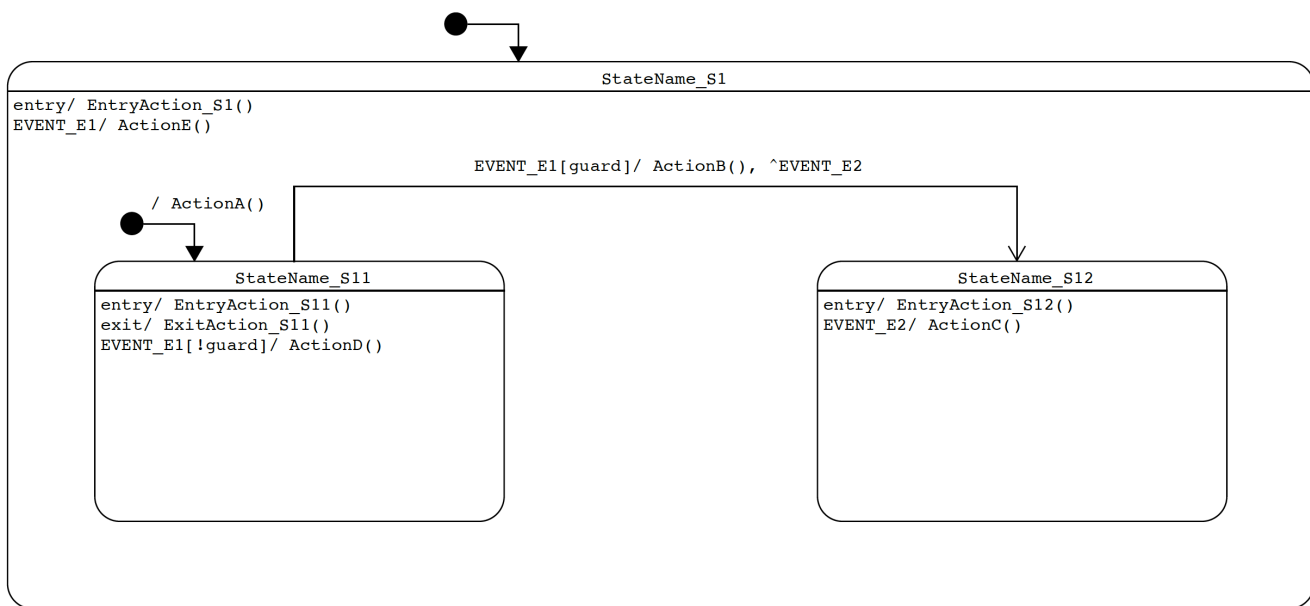
Please read PSiCC Chapter 2 for a crash course in statechart notation. For a detailed guide read OMG UML Version 2.5 Chapter 14 StateMachines.

Let's revisit the paper *Statecharts in the Making* by D. Harel. This is another quote from it:

"I recall an anecdote from somewhere in late 1983, in which in the midst of one of the sessions at the IAI the blackboard contained a rather complicated statechart that specified the intricate behavior of some portion of the Lavi avionics system... There was a knock on the door and in came one of the air force pilots from the headquarters of the project. He was a member of the "customer" requirements team, so he knew all about the intended aircraft (and eventually he would probably be able to fly one pretty easily too...), was smart and intelligent, but he had never seen a state machine or a state diagram before, not to mention a statechart. He stared for a moment at this picture on the blackboard, with its complicated mess of blobs, blobs inside other blobs, colored arrows splitting and merging, etc., and asked "What's that?" One of the members of the team said "Oh, that's the behavior of the so-and-so part of the system, and, by the way, these rounded rectangles are states, and the arrows are transitions between states". And that was all that was said. The pilot stood there studying the blackboard for a minute or two, and then said, "I think you have a mistake down here, this arrow should go over here and not over there"; and he was right. For me, this little event was significant, as it really seemed to indicate that perhaps what was happening was "right", that maybe this was a good and useful way of doing things. If an outsider could come in, just like that, and be able to grasp something that was pretty complicated but without being exposed to the technical details of the language or the approach, then maybe we are on the right track. Very encouraging."

2.1.2 Crash Course

This is my even more concise version of a crash course on statechart notation. We will see if the statechart notation is really as intuitive as D. Harel claimed. Let's take a look at a simple and hypothetical statechart below. It doesn't do anything meaningful but does illustrate most of the important notation and concepts. In the subsequent sections, we will gradually introduce more advanced concepts.



1. State

- A *state* is represented by a round-cornered rectangle with its name denoted in the upper compartment.
- A state may contain other states. The containing state is called a *superstate* or *composite state*. The contained state is called a *substate* or *nested state*. There can be multiple levels of nesting. This is like superclass and subclass relationship in OOP.
- Due to the hierarchy of state composition, this kind of state-machine is called *hierarchical state machine*, or simply *HSM*.
- When the system is in **StateName_S1**, it must also be in either **StateName_S11** or **StateName_S12** but not both.

StateName_S1 is also called an *OR-state*. Whenever the system is in an OR-state, the system must also be in exactly one of its substates. Later we will see another kind of composite state called an AND-state.

- The default initial state is indicated by a *dark solid circle/dot*, called the initial pseudo-state. The transition out of it points to the default state to enter at each nesting level.

In our example the system enters **StateName_S1** followed by entering **StateName_S11**.

- In each state there can be optional *entry action* and *exit action*, labeled as
 entry/ action_list
 exit/ action_list

See the next bullet point for explanation of `action_list`. Essentially an entry action is the activities to be performed every time when a state is entered, no matter how (i.e. via whichever transition) the state is entered. Similarly an exit action is the activities to be performed every time when a state is exited. (Think about how powerful this concept is!)

2. Transition

- a) A *transition* is a directed line from a source state to a destination state showing a state change triggered by an occurrence of an event labeled in the following format:
`EVENT_NAME[guard_condition]/ action_list`
- b) `EVENT_NAME` identifies the type of the triggering event, such as `EVENT_E1` or more concretely `START_REQ` and `START_DONE`.
- c) `guard_condition` is a logic expression to determine if a transition is enabled. A transition is only *enabled* if the triggering event arrives *and* the associated guard condition is evaluated to true. Examples include:

`[retryCount < 3]`

`[temperature >= 0 && temperature < 90]`

Guard conditions allow the use of *extended state variables*. They are typically member variables of state-machine objects. While they provide flexibility in controlling the flow of state-machines and help reduce the number of states, they are nonetheless the very things that state-machines are trying to replace. We need to find a sweet spot.

- d) `action_list` is a comma-separated list of activities to perform when the transition is triggered or fired. A transition is triggered when it is *enabled* and *selected*. We mentioned above that the `guard_condition` determines if a transition is enabled. Later we will discuss how a transition gets selected when there is a conflict among multiple enabled transitions (conflict resolution).

An `action_list` can be empty, meaning that there are no activities to accompany the transition. Otherwise it may contain a combination of (i) actual functions to be called, (2) events to be generated (with `^` or `^^1`) or (3) free-formed descriptions. Examples of `action_list` include:

`/ gpioHal.SetPin(LED1), motorDriver.Forward(speed)`

`/ turn on LED1, forward motor at speed`

`/ retryCount++, ^RETRY`

In the above example, `RETRY` is called a *reminder* event. It is posted to itself to trigger

¹ `^^` is my own notation for posting an event in a LIFO manner, i.e. synchronously.

some followup activities to be performed. There are different favors regarding how an event is posted (e.g. FIFO vs LIFO), which are related to the underlying queuing model.

- e) Initial transition is a directed line from an initial pseudo-state (dark solid dot) to the default state to enter. By the nature of being *default*, there shouldn't be `EVENT_NAME` or `[guard_condition]` in the label. (There is an exception. If there is a choice point or junction, there can be a `[guard_condition]` in each branch out of the choice point or junction.)

3. Internal Transition

- a) An internal transition is a transition without state change. It does sound contradictory. It remains in the same state without even exiting and re-entering the state. (There is significance here.) Since there is no state change there is no need for a directed line. Only a label is needed, which is placed inside the state rectangle starting at the top-left corner.
- b) Internal transitions can be viewed as the activities to be performed while remaining in a certain state. Often these activities represent the main features of a system when it is in a stable state, such as transfer data, handling periodic timeout events, etc. Together with the transition lines, they allow the complete system behaviors to be represented, which is not possible with traditional state diagrams.

2.2 Statechart Rules

2.2.1 State Hierarchy and Transition Triggering

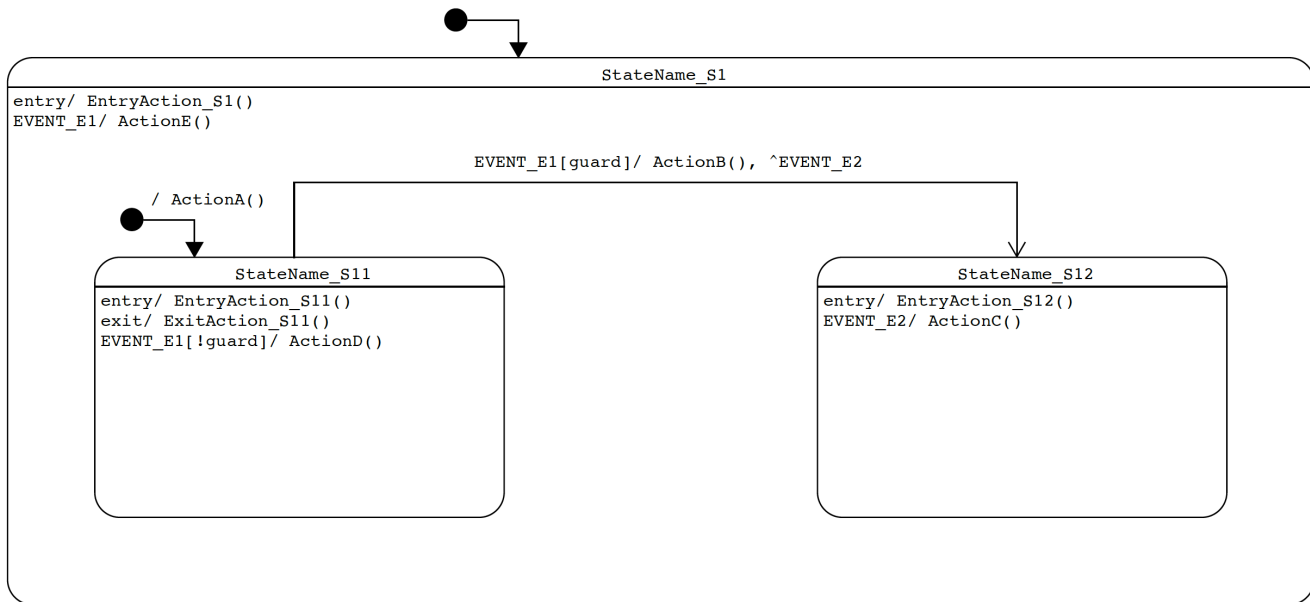
In a statechart, states are organized in a tree structure. Starting from the topmost root state, each state can contain substates. The relationship between a substate and its containing states (superstates) is like that between a subclass and its superclasses, which is an "is-a" relationship. For example if the system is in `StateName_S11` we say it is also in `StateName_S1`.

When an event arrives, we first determine the set of enabled transitions in the current leaf state (most nested) and all of its superstates. (Recall that a transition is enabled when its guard condition is evaluated to be true.)

There is a conflict if there is more than one enabled transitions in the set. The system will attempt to resolve the conflict by selecting one transition out of the set in the order from the most nested state outward to the root state. The conflict cannot be unresolved if there are more than one transitions enabled in the same state, and we will call the statechart malformed.

In other words, priority is given to a more nested state, in a way similar to how overriding functions are resolved from the most derived class to the base class in OOP.

In the example statechart above, reproduced here to save some scrolling, we observe that:



1. When EVENT_E1 arrives in StateName_S11,
 - a) if guard is true, ActionB() will be called and the system transits to StateName_S12.
 - b) if guard is false, ActionD() will be called and the system stays in the same state.
 - c) ActionE() in the superstate will never be called.
2. When EVENT_E2 arrives in StateName_S11, there are no enabled transitions and the event will be discarded.
3. When EVENT_E2 arrives in StateName_S12, ActionC() will be called.
4. When EVENT_E1 arrives in StateName_S12, ActionE() in the superstate will be called.

Note – There is a subtle difference between a guard condition and an *if* statement in the action list. In the former case, if a guard condition is false the transition is not enabled, and therefore will not be considered for selection. In the latter case, no matter whether the if-condition is true or false the transition, when selected, has already fired, and therefore overrides any other transitions in its parent states.

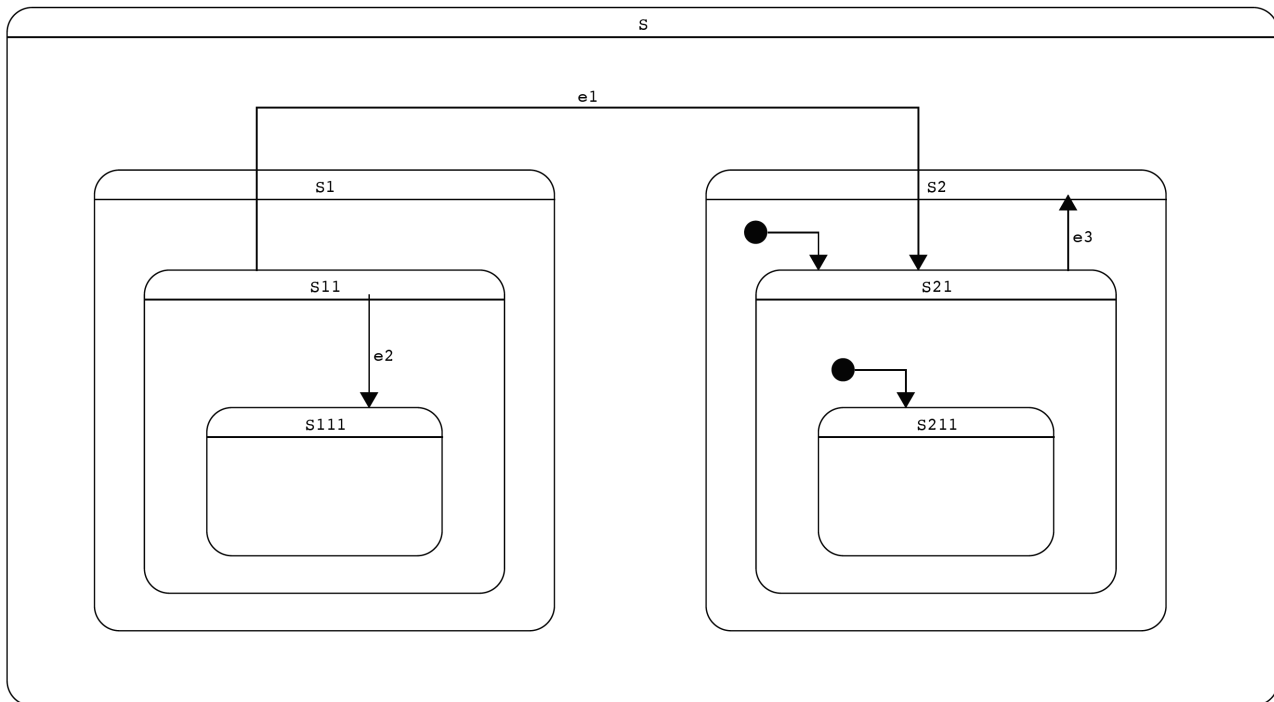
2.2.2 State Exits and Entries in a Transition

Let's define some terminology for this discussion:

1. Current state – The leaf (most nested) state the system is in at the moment.
2. Transition source – The originating state of a transition line.
3. Transition target – The terminating state of a transition line.

4. Final state – The new leaf state after transition.

A *transition source* can be the same as the *current state*, or a parent of the current state. A *final state* can be the same as a *transition target*, or a child of the transition target.



For example, when *e1* arrives in *S111*, the transition source (*S11*) is the parent of the current state (*S111*). The transition target (*S21*) is the parent of the final state (*S211*) due to initial transition.

During a state transition states will be exited and entered, and the corresponding exit and entry actions will be automatically performed. The sequence of actions can be divided into the following three main steps:

1. Exit actions from the current state up to *but not including* the transition source. When both are the same there will be no actions in this step.
2. Exit actions from the transition source up to but not including the *lowest common ancestor* (LCA), followed by entry actions from there into the transition target.

According to this rule, even when the transition source is the same as the transition target, that source/target state is still exited and re-entered.

However there are two special cases for this rule.

- a) If the transition source is a parent of the transition target (*e2*), the transition source is not

exited and re-entered.

b) If the transition source is a substate of the transition target (e3), the transition target is not exited and re-entered.

3. Entry actions from *but not including* the transition target into the final state.

If the transition target is the same as the final state there will be no actions in this step; otherwise the system will follow the initial transitions repeatedly into the final state.

3 Introduction to Quantum Platform (QP)

Now that we know what statecharts are and how more complex they are compared to traditional state diagrams, we are sure that conventional methods for FSM (e.g double-switch, state table and state pattern) are not sufficient to implement statecharts. Let's dive in to see how QP solve this problem elegantly.

3.1 Basic Ideas

Like the state table method, QP represents each state with a function and maintains a function pointer to point to the current state function. It goes beyond state table by supporting hierarchical states.

Like the state pattern method, QP represents each *state machine* with a class. However it does *not* implement each state in its own class. Instead QP implements each state as a member function of the state machine class which automatically provides a context for all the states.

Like the double-switch method, it uses a switch-case construct to handle different events in a given state.

In other words, QP combines the good parts of existing methods and patterns to form a new one which is superior to each of the original ones.

3.2 State Member Function

The center piece of QP is the use of a member function to implement each state of an HSM. QP provides a base class QHsm as the basis for all state-machine classes. Developers derive their own application state-machine classes from QHsm and implement their states as member functions of those derived classes.

We will see a complete example in the next section. First we present the basic form of a state member function:

```
QState MyHsm::MyState(MyHsm * const me, QEvt const * const e) {  
    switch (e->sig) {  
        case Q_ENTRY_SIG: {                                // Entry actions.
```

```

        ...
        return Q_HANDLED();
    }
    case Q_EXIT_SIG: {                                // Exit actions.
        ...
        return Q_HANDLED();
    }
    case Q_INIT_SIG: {                                // Initial transition.
        ...
        return Q_TRAN(&MyHsm::MyDefaultSubState);
    }
    case MY_EVENT_A: {
        ...
        return Q_TRAN(&MyHsm::MyTargetState); // State transition.
    }
    case MY_EVENT_B: {
        ...
        return Q_HANDLED();                        // Internal transition.
    }
    case MY_EVENT_C: {
        if (me->MyGuard()) {
            ...
            return Q_HANDLED();                    // Guard condition passes
        }
        break;                                       // Ignored in this state.
    }
}                                                    // Event not handled.
return Q_SUPER(&MyHsm::MySuperState);            // Returns super state.
}

```

The basic form above contains almost all important concepts of a statechart.

1. MyState is a member function of the state-machine class MyHsm. Since it is declared as a *static* member function, the "*this*" pointer to the object is passed explicitly into the function via the parameter *me*. This is not necessary but is done this way in QP to support out-dated compilers that do not support pointer-to-member-functions properly. Yes, you can have multiple instances of a state-machine class.
2. The event to be handled is passed in as the second parameter *e*. It is passed by reference as a pointer to the base event class *QEvt*. From its member *sig* the switch-case statement differentiates which event has arrived.

Once it has figured out the event type, it can downcast *e* to the actual subclass derived from *QEvt*. It is done this way to avoid the overhead of alternatives using virtual functions or RTTI.

3. QP provides built-in event types, namely *Q_ENTRY_SIG*, *Q_EXIT_SIG* and *Q_INIT_SIG* as

annotated above. We can add as many application specific events as we need, such as MY_EVENT_A, MY_EVENT_B and MY_EVENT_C shown above.

4. A guard condition is implemented with an *if* statement. Note that if a guard condition is evaluated to be false, the transition is *disabled* and the event must be propagated to the super states until a transition is selected (or the event is discarded if none is found).

What is the difference between the following two transitions?

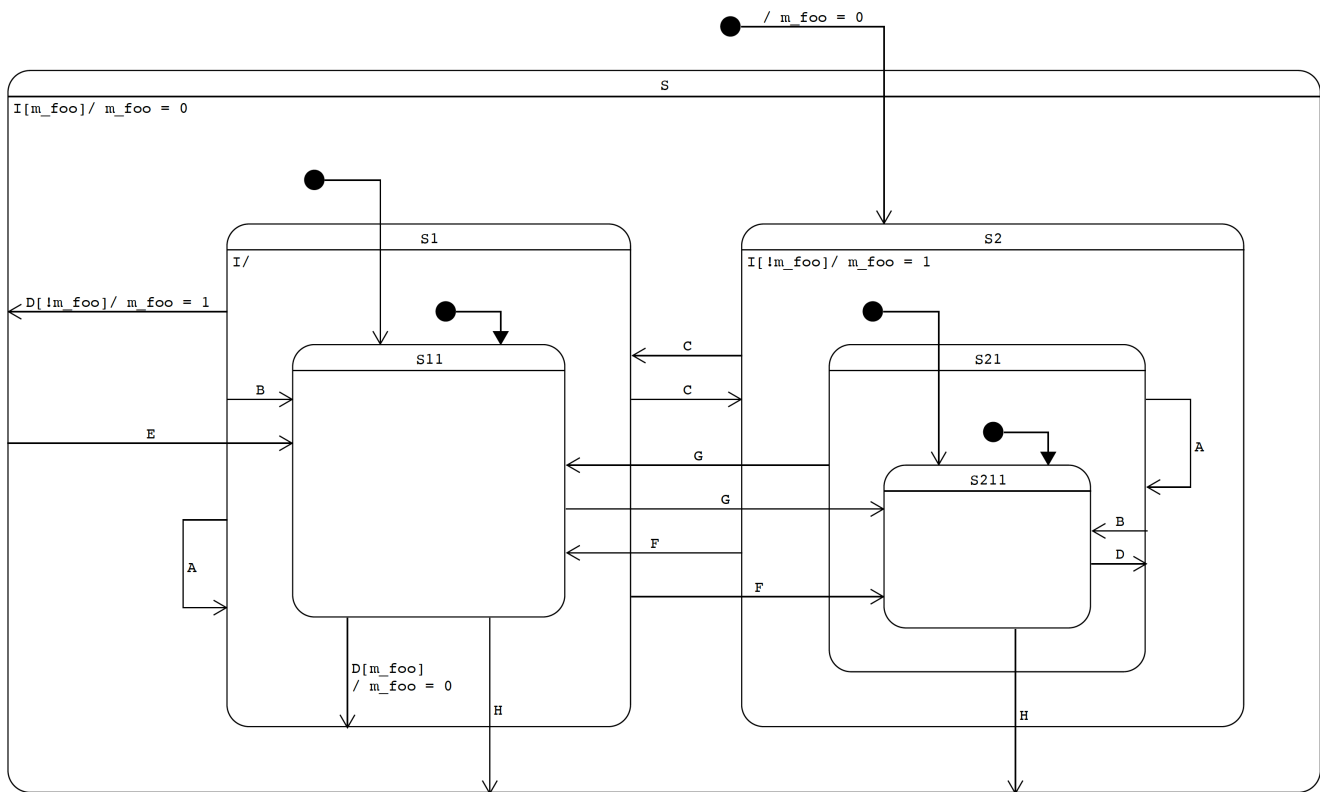
a) MY_EVENT_C[MyGuard()] / ...

b) MY_EVENT_C/ if (MyGuard()) ...

5. The return value of a state member function tells QP the result of an event handling by this state. There are three options provided by QP as macros:
 - a) Q_HANDLED() - The event is handled by either an entry action, exit action or an *internal transition*.
 - b) Q_TRAN() - The event is handled by a state transition (including initial transition) to a different or the same state. The target state is specified as a pointer to member function. All exit and entry actions along the path of a transition will be performed as QP automatically dispatches Q_ENTRY_SIG, Q_EXIT_SIG and Q_INIT_SIG to all involved states in the proper order.
 - c) Q_SUPER() - The event is not handled by this state. That is there is no enabled transition to be selected in this state. The immediate super state is specified as a pointer to member function. If this state is already at the highest level, the event will be propagated to the built-in *top* state (think of it as the paper) which automatically discards the event. This is the ingenious way of how state hierarchy is represented in QP.

3.3 Example

This example is based on Figure 2.11 on Page 88 of the PSiCC book.



It is implemented in the Demo active object. We haven't formally covered active object yet, but now we just need to know an active object *is an* HSM running in its own thread. See src/Demo/Demo.cpp for the complete implementation. The state member function for S1 is extracted here for explanation.

```
QState Demo::S1(Demo * const me, QEvt const * const e) {
    switch (e->sig) {
        case Q_ENTRY_SIG: {
            EVENT(e);
            return Q_HANDLED();
        }
        case Q_EXIT_SIG: {
            EVENT(e);
            return Q_HANDLED();
        }
        case Q_INIT_SIG: {
            return Q_TRAN(&Demo::S11);
        }
        case DEMO_A_REQ: {
            EVENT(e);
            return Q_TRAN(&Demo::S1);
        }
        case DEMO_B_REQ: {
            EVENT(e);
            return Q_TRAN(&Demo::S11);
        }
    }
}
```

```

    }
    case DEMO_C_REQ: {
        EVENT(e);
        return Q_TRAN(&Demo::S2);
    }
    case DEMO_D_REQ: {
        EVENT(e);
        if (!me->m_foo) {
            me->m_foo = 1;
            return Q_TRAN(&Demo::S);
        }
        break;
    }
    case DEMO_F_REQ: {
        EVENT(e);
        return Q_TRAN(&Demo::S211);
    }
    case DEMO_I_REQ: {
        EVENT(e);
        return Q_HANDLED();
    }
}
return Q_SUPER(&Demo::S);
}

```

Do you see the direct and obvious mapping between code and statechart? This is a simple yet powerful concept to allow developers to translate design to code and vice versa. If you had the experiences of trying to understand someone else's code (or your own a few months back) with a lots of variables that are set, clear and tested in many places, you would start appreciating the elegance of this method.

Some sample log output is shown here for reference.

```

2890 CONSOLE_UART2> demo ?
[Commands]
test          Test function
a             A evt
b             B evt
c             C evt
d             D evt
e             E evt
f             F evt
g             G evt
h             H evt
i             I evt
?             List commands

```

```

128825 CONSOLE_UART2> demo c

```


133344 DEMO(29): S2 DEMO_C_REQ from CONSOLE_UART2(2) seq=0
133344 DEMO(29): S211 EXIT
133344 DEMO(29): S21 EXIT
133344 DEMO(29): S2 EXIT
133344 DEMO(29): S1 ENTRY
133344 DEMO(29): S11 ENTRY

138788 CONSOLE_UART2> demo g
140892 DEMO(29): S11 DEMO_G_REQ from CONSOLE_UART2(2) seq=0
140892 DEMO(29): S11 EXIT
140892 DEMO(29): S1 EXIT
140892 DEMO(29): S2 ENTRY
140892 DEMO(29): S21 ENTRY
140892 DEMO(29): S211 ENTRY

141571 CONSOLE_UART2> demo h
143149 DEMO(29): S211 DEMO_H_REQ from CONSOLE_UART2(2) seq=0
143149 DEMO(29): S211 EXIT
143149 DEMO(29): S21 EXIT
143149 DEMO(29): S2 EXIT
143149 DEMO(29): S1 ENTRY
143149 DEMO(29): S11 ENTRY

143562 CONSOLE_UART2> demo d
145183 DEMO(29): S11 DEMO_D_REQ from CONSOLE_UART2(2) seq=0
145183 DEMO(29): S1 DEMO_D_REQ from CONSOLE_UART2(2) seq=0
145183 DEMO(29): S11 EXIT
145183 DEMO(29): S1 EXIT
145183 DEMO(29): S1 ENTRY
145183 DEMO(29): S11 ENTRY

145600 CONSOLE_UART2> demo d
147024 DEMO(29): S11 DEMO_D_REQ from CONSOLE_UART2(2) seq=0
147024 DEMO(29): S11 EXIT
147024 DEMO(29): S11 ENTRY

4 UML and Object-oriented Design

4.1 Introduction

Earlier we have looked at object-oriented programming techniques using C++, which include encapsulation, inheritance and polymorphism. We will soon see real-life examples of how these techniques can be applied to practical embedded projects. We will learn how to build a generic application framework upon which various components can be added to support different kinds of hardware modules.

As we will be talking about objects, their relationships and interactions, we need an effective way to describe them. Natural language can quickly become clumsy and imprecise. UML, Unified Modeling Language, is an attempt to offer a common graphical language to solve this problem.

You can download the UML 2.5 Specification from its official website:

<http://www.omg.org/spec/UML/2.5/About-UML/>

UML was once a big fad. Recently it has cooled down and now gets mixed reception by software professionals. UML is about modeling and design. The big trends these days are about coding, agility, continuous integration, etc. Design is not on the list. No one can dispute what ultimately matters in software development is code that executes. Pretty diagrams, though sometimes useful, are not as important.

UML by itself does not provide any means of implementation. Tools that translate UML models to executable code can be expensive yet may not fit your own workflow. QP seems to have made changes to that. We will see how QP goes beyond a single HSM to support object-oriented design for an entire embedded system.

Out of the many different types of diagrams, I find the following three the most useful for embedded design:

1. Class diagrams
2. Sequence diagrams
3. Statecharts

We have already looked at statecharts, so we will focus on the first two below.

4.2 Class Diagrams

A class diagram shows the *static structure* of classes. It reveals what data members and member functions a class contains, and the visibility of its members (public, protected or private). The public member functions of a class are collectively known as its API (application programming interface).

The type of a data member or the return type of a member function is shown after the ":" sign following

the name of a member. It can be a built-in type such as `uint32_t` or another class type.

Data members are listed in the upper compartment while member functions are listed in the lower one. Visibility is marked by a "+", "#" or "-" sign at the front of each member:

1. + public member
It can be accessed in both member and non-member functions of this class.
2. # protect member
It can only be accessed by member functions of this class, its derived classes or by its *friends*.
3. - private member
It can only be accessed by member functions of this class, or by its *friends*.

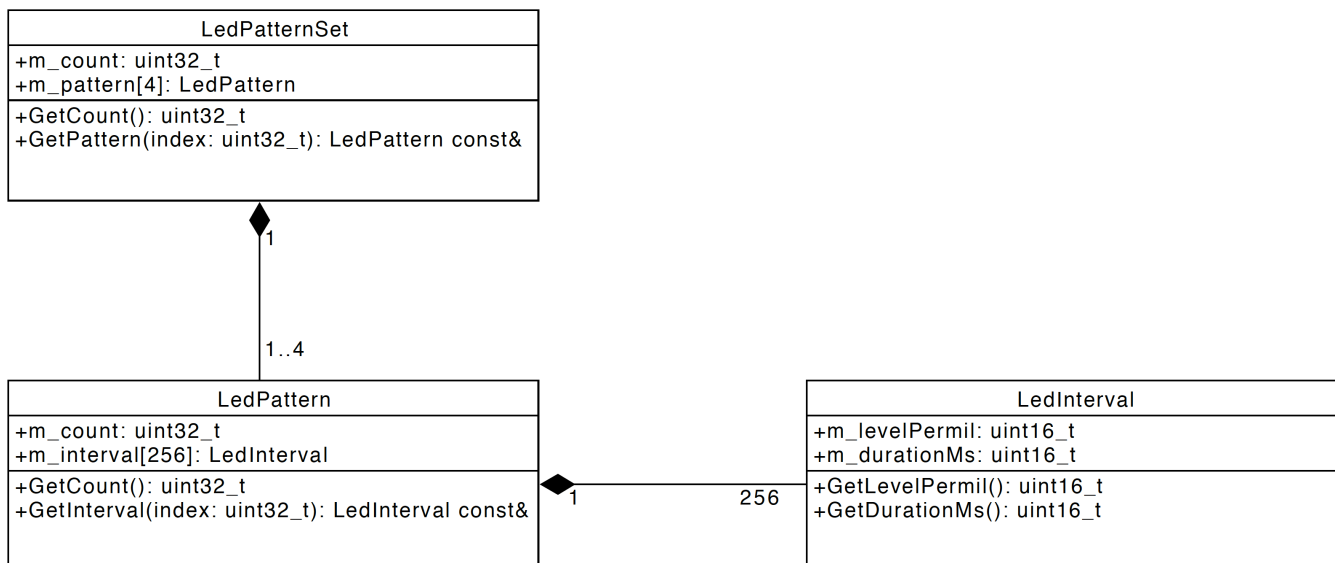
A class diagram also shows the static relationship between classes such as composition (*has-a*) or inheritance (*is-a*). Composition is denoted by a line with a solid diamond attached to the containing class. Inheritance is denoted by a line with a hollow triangle attached to the base class.

A class diagram, however, *does not* show the *dynamic* behaviors of a class, e.g. how its data members are used or what its member functions does, nor does it show the dynamic interactions among classes.

Below is an example of a class diagram showing the composition relationship between:

1. A set of LED patterns and the patterns contained by the set.
2. An LED pattern and the intervals contained by the pattern.

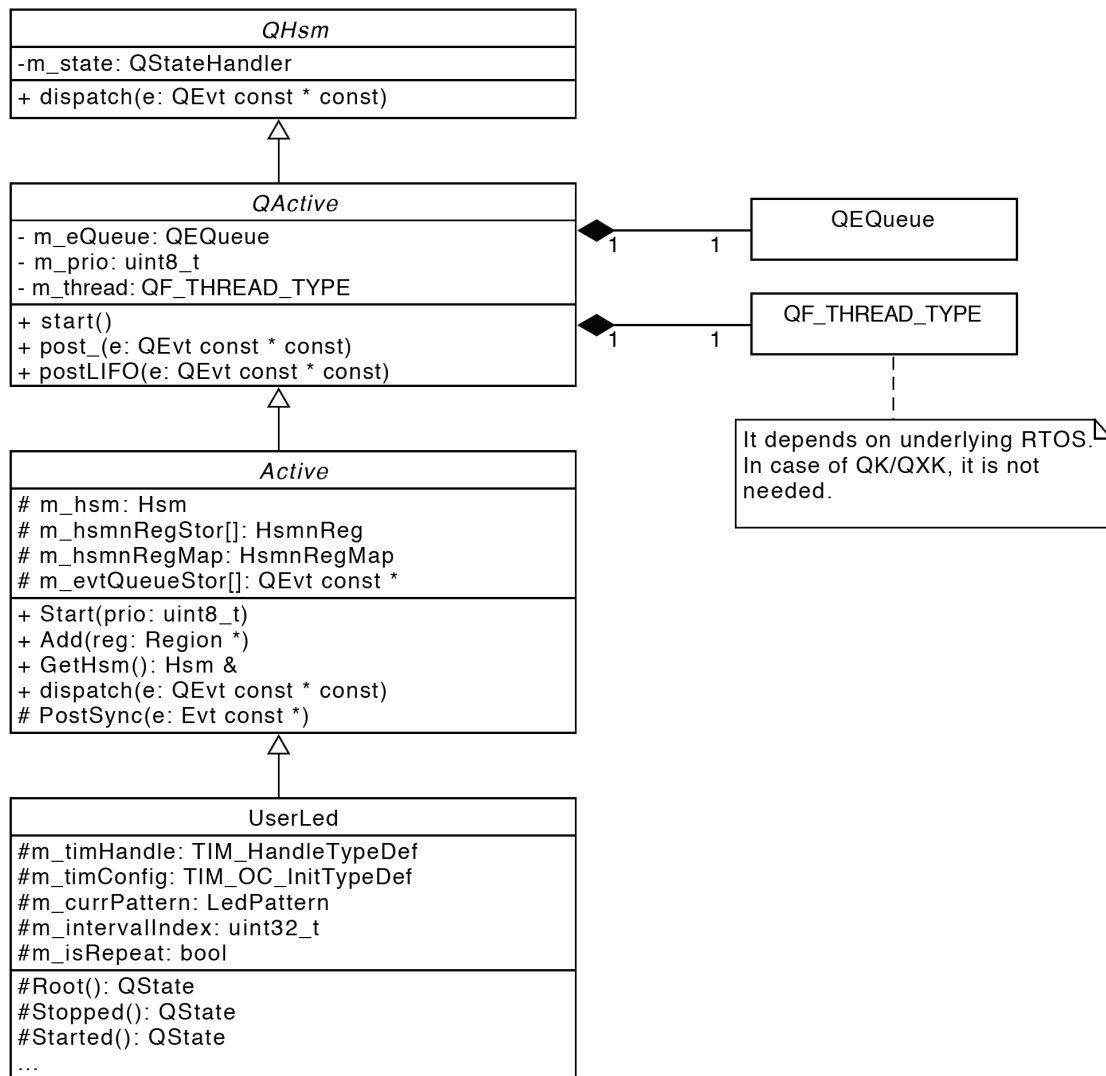
What we are actually saying is an LED pattern set *has a* collection of LED patterns, each of which *has a* sequence of intervals, which in turn *has a* brightness level and an associated duration.



The next example shows the inheritance relationship between the application active object named UserLed and its base classes. UserLed is derived from a chain of base classes in the order from Active, QActive and QHsm:

1. UserLed – An application class responsible for showing a selected LED pattern upon request.
2. Active – A reusable class in the application framework representing a high-level active object.
3. QActive – A basic active object class provided by QP.
4. QHsm – A hierarchical state-machine class provided by QP.

In other words, UserLed is a high-level active object, which is a basic active object, which in turn is a hierarchical state-machine.



4.3 Sequence Diagrams

4.3.1 Synchronous vs Asynchronous Interactions

While class diagrams do not show dynamic interactions among classes, sequence diagrams come to the rescue. A sequence diagram represents each object participating in a scenario with a lifeline which is a vertical dashed line coming down from a rectangle showing its name. It uses a horizontal line with a *solid arrowhead* to show a *synchronous* function call from one object to another (or to itself). It uses a horizontal line with a *stick arrowhead* to show an *asynchronous* message or event posted from one object to another (or to itself).

In software design the term *synchronous* typically describes something that happens atomically without anything else getting in between. For example if one object calls a public member function of another object, the calling object cannot do anything else while waiting for the function to return, and hence this kind of interaction is called *synchronous*. We can also call it as a *blocking* call, since the calling object is blocked until the function returns.

The opposite is true for *asynchronous* interaction. When one object posts an event to another object (or to itself), it simply puts the event into the event queue of the destination object. The post() call returns quickly without waiting for the event to be processed. The posting object is free to do other things while waiting for the result to come back, usually through a confirmation or response event. Another name for this type of interaction is *non-blocking*.

Synchronous and asynchronous design are two fundamentally different design philosophy. They are suitable for different types of applications. For data- or algorithm-centric applications synchronous design is attractive for its straight-forward style. For real-time embedded applications, which must remain responsive to various inputs at all times, asynchronous design is far more flexible. Statecharts and QP are our perfect partners in doing asynchronous design.

4.3.2 Limitations

Each sequence diagram usually shows only one particular scenario such as one of the happy paths or one of the failure paths. With the help of an *alternative combined fragment*, it may show conditional paths as in an if-else block. Nevertheless, the number of scenarios that can be clearly shown in one sequence diagram is still very limited.

It does not show what happens if the order of received event is different or a failure occurs at a different place. In other words it does not show a *complete* picture of the system being modeled, rather like showing a small sample of cross-sectional views of a 3D physical model. You may need an astronomical number of sequence diagrams to show all possible combinations of event order, event parameters and system conditions.

It is not uncommon for developers to start coding based on one or two sequence diagrams about normal

cases. When the time comes to worry about failure cases (usually under schedule pressure), they will then refactor the code to add variables and conditional code to handle exception cases (this is when *architectural decay* or *technical debt* starts). Sequential diagrams are very useful tools for analysis and architectural design. They help us visualize interactions, understand aspects of behaviors and partition systems into components. However they do not show the complete picture of a system. To that end, we need statecharts to give us a *complete, precise* and *concise* description of how a system behaves.

4.3.3 Example

Below is an example showing how the UserLed object initializes an LED pattern upon a button press by a user. Note the use of both asynchronous and synchronous interactions. The next diagram shows what happens when the INTERVAL_TIMEOUT event arrives. Note the use of an alternative combined fragment to show conditional paths when handling wrap-around cases.

