Module 5

Design and Optimization of Embedded and Real-time Systems

1 UML and Object-oriented Design

1.1 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 hierarchical state machines (HSM) can be added to support both high-level features and low-level components.

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 get the UML 2.5 Specification from its official website:
https://www.omg.org/spec/UML/2.5.1/PDF

UML by itself does not provide any means of implementation. There are many tools, both commercial and open-source, that translate UML models to executable code. QP is one of such tools and we will learn how QP goes beyond a single HSM to support object-oriented design of an entire embedded system.

Among different kinds of UML diagrams, the following three are commonly used for embedded system design. We have already looked at statecharts, so we will focus on the other two kinds below.

1. Class diagrams
2. Sequence diagrams
3. Statecharts

1.2 Class Diagrams

A class diagram shows the *static structure* of a class. 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*.

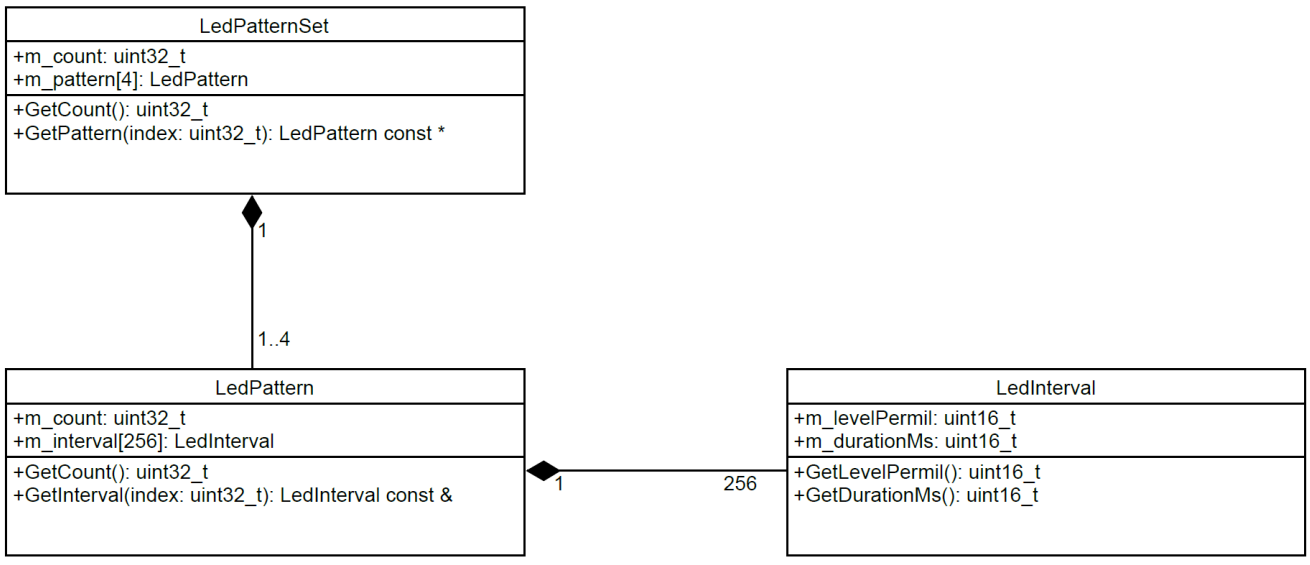
A class diagram also shows the static relationship between classes such as composition (*has-a*) or inheritance (*is-a*). Composition is denoted by a line with a solid diamond attached to the containing class. Inheritance is denoted by a line with a hollow triangle attached to the base class.

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 do, 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.

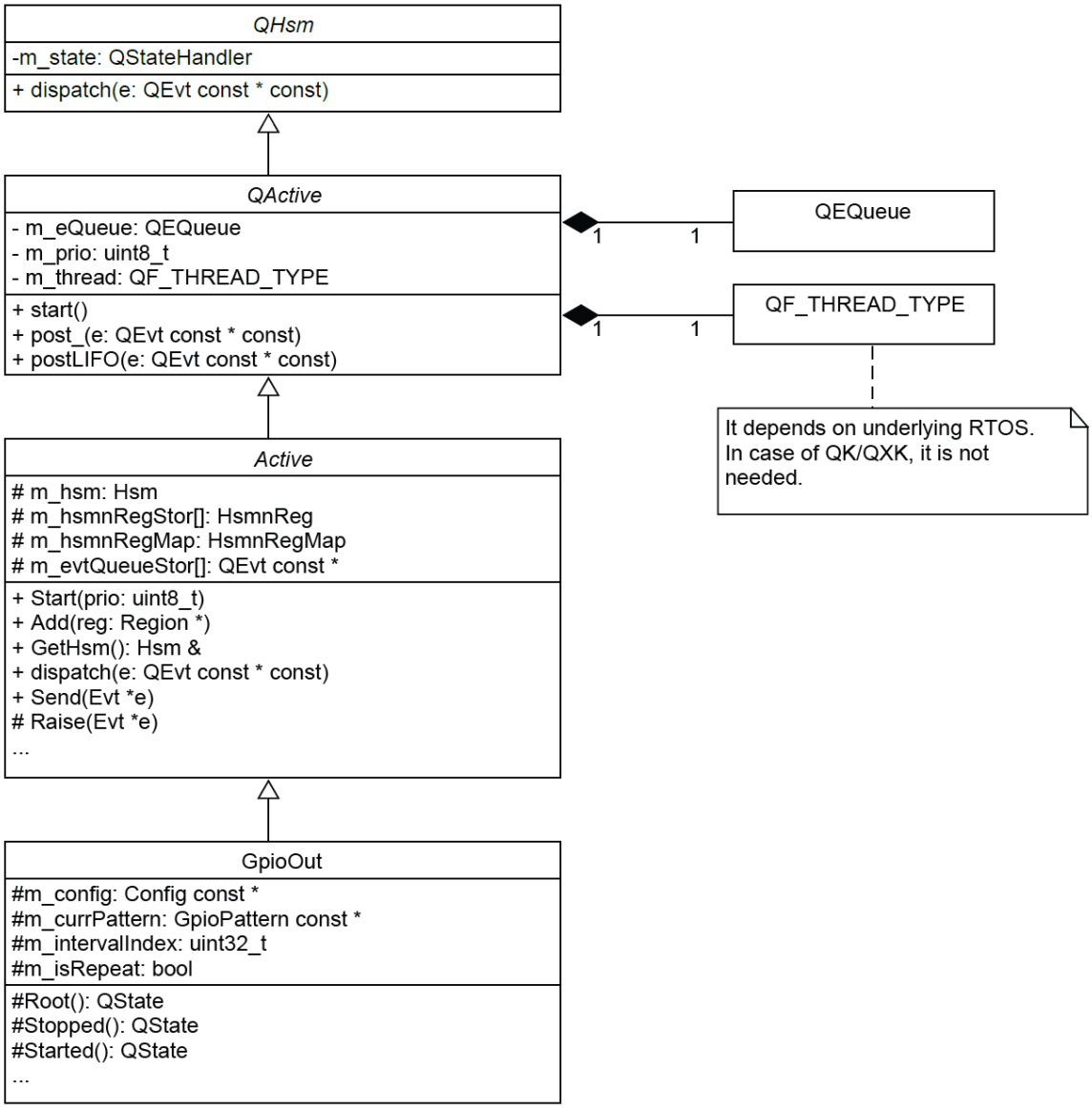
What we are actually saying is an LED pattern set *has a* collection of LED patterns, each of which *has a* sequence of intervals, which in turn *has a* brightness level and an associated duration.



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

1. GpioOut – An application class responsible for generating patterns on an GPIO output pin.
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, GpioOut *is a* high-level active object, which *is a* basic active object, which in turn *is a* hierarchical state-machine.



1.3 Sequence Diagrams

1.3.1 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).

When one object calls a public member function of another object, the calling object cannot do anything else while waiting for the function to return. We call this kind of interaction *synchronous* or *blocking*, since the calling object is blocked until the function returns.

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. We call this kind of interaction *asynchronous* or *non-blocking*.

Synchronous and asynchronous design are two fundamentally different design philosophy. They are suitable for different types of applications. For data-centric applications based on sequential logic and algorithms, synchronous design is a good choice. 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 to do asynchronous design.

1.3.2 Limitations

Each sequence diagram usually shows only one particular scenario – a happy path or an exception path. 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.

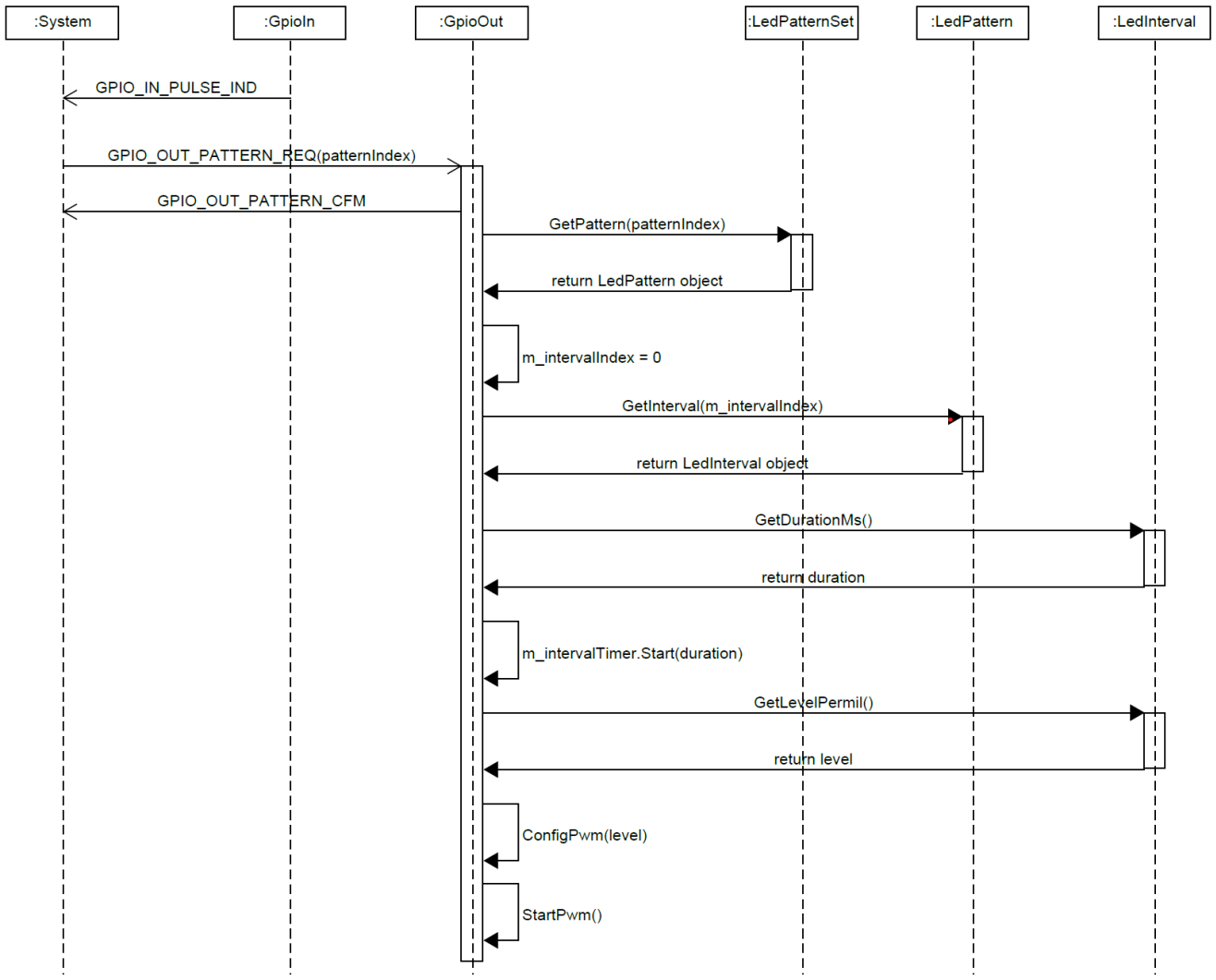
It does not show what happens if the order of received event is different or a failure occurs at a different place. In other words it does not show a *complete* picture of the system being modeled, rather like showing a small sample of cross-sectional views of a 3D physical model. You may need an astronomical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

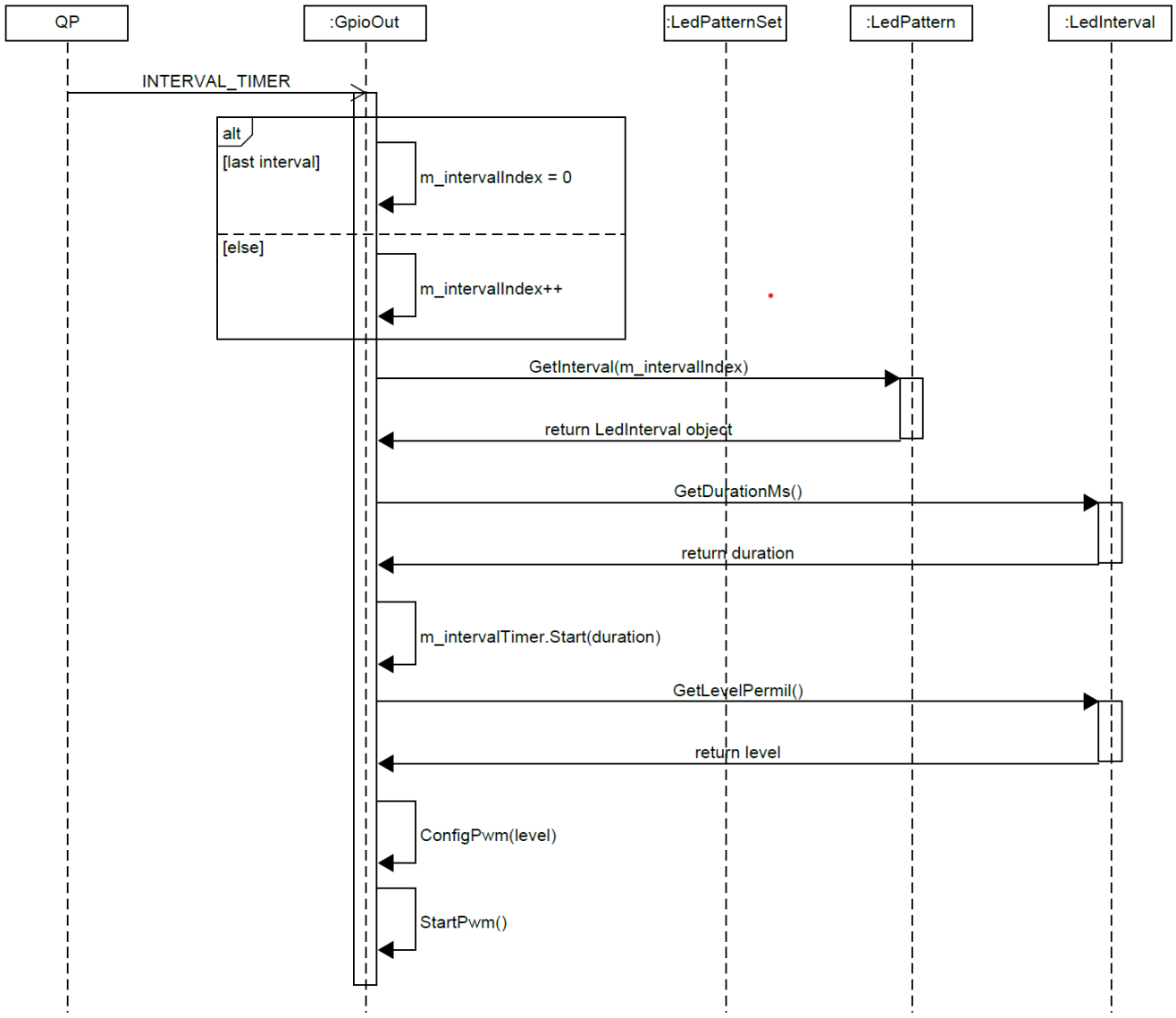
It is not uncommon for developers to start coding based on one or two sequence diagrams about normal cases (*happy paths*). When the time comes to worry about failure cases (usually under schedule pressure), they will then refactor the code to add variables and conditional code to handle exception

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 design of a system. To that end, we need statecharts to give us a *complete*, *precise* and *concise* description of how a system behaves.

1.3.3 Example

Below is an example showing how the GpioOut 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.





2 Application Framework with QP

2.1 QHsm Event Processor

Previously we have demonstrated how to implement a simple HSM named *Demo* using QP. This is a recap of the basic ideas:

1. Derive your application HSM class (e.g. Demo) from the QHsm base class provided by QP. (Note there can be immediately base classes.)
2. Implement each state as a member function of the HSM class.
3. Handle received events in each state function with a switch-case construct according to the behaviors defined in the corresponding statechart.

Since this is such a fundamental coding pattern, it's worth showing it again:

```
QState Demo::S21(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: {
            EVENT(e);
            return Q_TRAN(&Demo::S211);
        }
        case DEMO_A_REQ: {
            EVENT(e);
            return Q_TRAN(&Demo::S21);
        }
        case DEMO_B_REQ: {
            EVENT(e);
            return Q_TRAN(&Demo::S211);
        }
        case DEMO_G_REQ: {
            EVENT(e);
            return Q_TRAN(&Demo::S11);
        }
    }
    return Q_SUPER(&Demo::S2);
}
```

QHsm is called an *event processor*. Its job is to process events ***dispatched*** to it (via its public method *dispatch()*) according to the rules encoded in its state functions. Notice the keyword "***dispatched"*** is in passive voice, which means some other objects must call its dispatch() function to pass events to it. In other words a QHsm object does not actively grab events to process on its own. It is passive.

QHsm is useful by itself, since it handles all the complex transition rules in an arbitrarily complex state hierarchy. However it will be even more useful if we can extend QHsm to become *active*. We call such extension an **a*ctive object*** represented by *QActive* in QP.

Note – For a complete and formal description of the statechart semantics, see this:

https://www.researchgate.net/publication/2533509_On_the_Formal_Semantics_of_VisualSTATE_Statecharts

2.2 Active Object

By *active*, we mean an active object will actively wait for events to arrive and dispatch them to an HSM on its own. What does it need to make this happen?

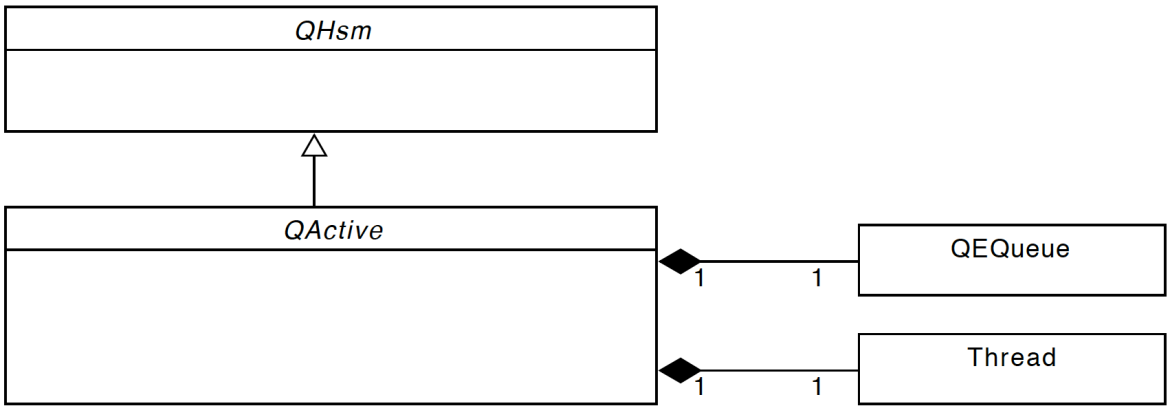
Just think about how you would normally do when using an RTOS. Most likely you would create a thread and have it block on an event queue for events for arrive. As long as the queue is non-empty, the thread will get the oldest event out of the queue to process. After each event it will loop back to check the queue again, and if the queue is empty it will block on it (i.e. the GetEvent() call will not return until an event arrives). This loop is called an event loop, and it typically looks like this:

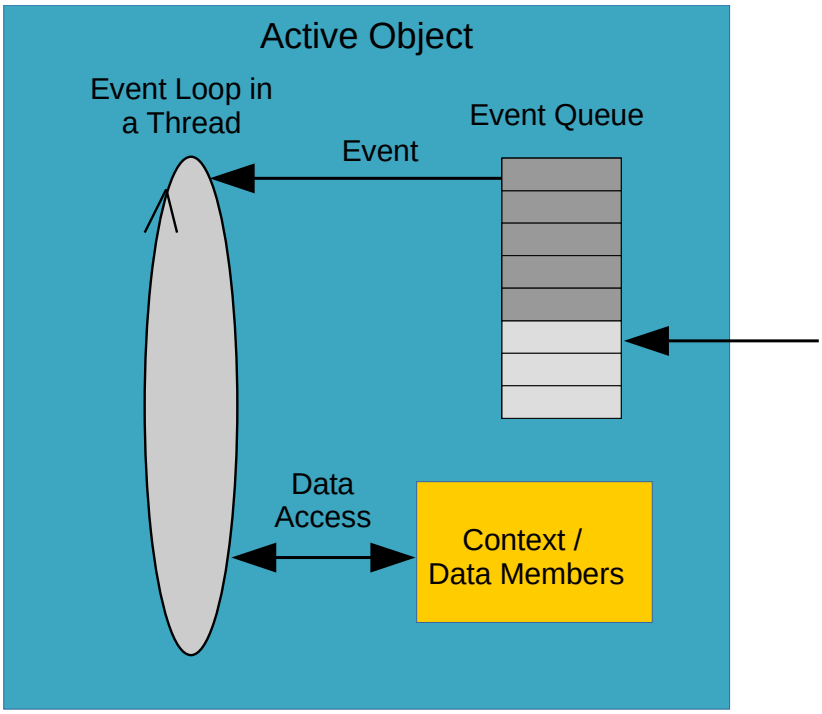
```
void ThreadMain() {
    while (running) {                          // Event loop.
        QEvt const *e = eventQueue.GetEvent();  // Blocking call.
        switch (e->sig) {
            stateMachine.dispatch(e)           // Handles event in state-machine.
        }
    }
}
```

This pattern will probably show up many times in a project. Rather than crafting it from scratch every time it would be better for us to apply object-oriented design to:

1. Encapsulate related concepts together in a class, which means adding a *thread* and an *event queue* to a *hierarchical state-machine*. We call it QActive.
2. Enable code reuse by inheriting application specific classes from common reusable base classes (QHsm and QActive).

In a nutshell, an active object *is a* hierarchical state-machine that *has a* thread and *has an* event queue as shown below.

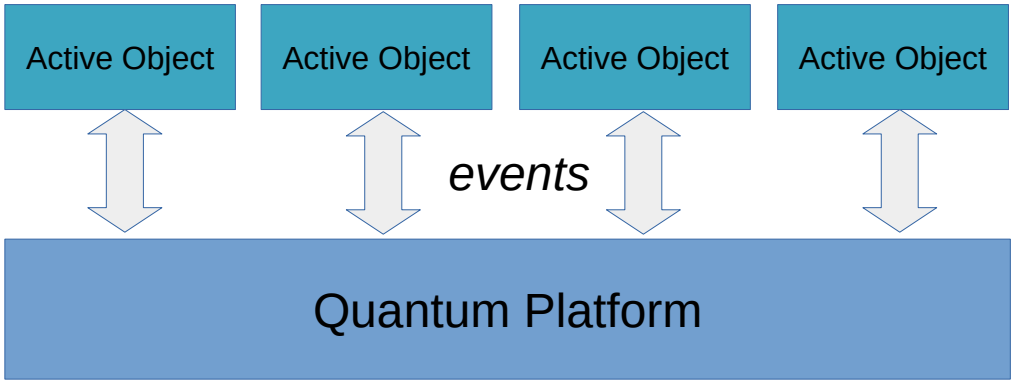




2.3 Publish-Subscribe

With active objects being fundamental building blocks, we can view a software system as a collection (or *partition*) of multiple active objects. Each active object is assigned with distinct and well-defined responsibilities and they communicate with each other asynchronously via events. Altogether they implement the complete system behaviors. Through this architectural pattern, we achieve a high degree of *decoupling* or *separation of concerns*.

There are different ways active objects can post events to each other. One pattern is called *publish-subscribe* which is provided by QP.



QP serves as an event broker in this pattern. Active objects exchange events through QP. As a result, they do not call public methods (API) of other active objects directly, and therefore do not need to maintain references to other active objects. Maintaining references to other objects involve object lifetime and ownership management, which can be complicated and can easily cause dangling pointers or memory leak issues.

An active object publishes an event via this QP API:

```
QF::publish_(QEvt const * const e)
```

QP will post this event to the event queues of any active objects that have subscribed to this event type via this API:

```
void QActive::subscribe(enum_t const sig)
```

The advantages of publish-subscribe include a high degree of flexibility and decoupling since there are no linkages among active objects (i.e. there are no direct function calls from one object to another). It is very easy to drop in another component as long as it subscribes to the same events.

Another advantage of this type of event-driven architecture is that there is minimal or no sharing of data between threads, thanks to encapsulating a thread within the active object class which provides a context for the thread. That context, in form of data members of the class, includes everything the

thread needs to access directly. When threads do not share data, there is little need to use mutex and the whole class of issues related to mutexes and data contention disappear. See the diagram above on *Active Object* for an illustration of local context within each active object.

Nevertheless, the broadcast nature of publish-subscribe can be problematic to ensure reliability. A publisher does not know if there are any subscribers of a published event, and if so who they are. It is therefore difficult to implement return-event paths for acknowledgment and event handshaking. We might be caught by race-condition issues that are very hard to track down.

3 Enhanced Framework

QP provides a higher level of abstraction (events and states) over tradition RTOS primitives such as threads, queues, mutexes, etc. It allows us to focus on actual behaviors rather than low level interactions such as when we need to lock a mutex, whether a function call will block or which inter-task communication primitives are used in different cases.

QP is intended to be a generic framework. It is flexible but can be minimal. When you use it in your projects you often need to provide external hooks and apply certain patterns repeatedly. For example,

1. When creating an active object you need to allocate memory for its event queue externally.
2. To support orthogonal regions (i.e. multiple HSMs in one active object) you need to delegate events from the active object to each HSMs explicitly.
3. The built-in publish-subscribe mechanism may not fit your projects if you need strong synchronization among objects. You will need to apply some well-defined semantics to your event interface and customize the event delivery mechanism.
4. Though QP provides a built-in debugging tool called QSPY, you may still want to use a custom one for your own needs.

To provide an even higher level of abstraction on top of QP, I have developed my own application framework (simply called *Fw)* located under the folder *Src/framework* in the source tree.

3.1 Common Attributes

In QP there is no builtin identifiers for active objects or HSMs, except the C++ pointers to them. If you want to have a simple ID number or name to be associated with each active object or HSM, you would have to add a *m_id* or *m_name* field to each concrete class derived from QActive or QHsm. Over time you will have a collection of data members that are common to all your concrete classes. It is a good chance to apply encapsulation and use a class to contain all these common attributes as well as common helper methods.

We call this class *Hsm*, meaning that it contains attributes and methods that are common to all

hierarchical state-machines. It is defined in *Src/framework/include/fw_hsm.h*. Below is an extract of the class definition:

```
class Hsm {
...
protected:
    enum {
        DEFER_QUEUE_COUNT = 16,
        REMINDER_QUEUE_COUNT = 4
    };
    Hsmn m_hsmn;
    char const * m_name;
    QP::QHsm *m_qhsm;
    char const *m_state;
    DeferEQueue m_deferEQueue;
    QP::QEQueue m_reminderQueue;
    ...
}
```

The member *m_hsmn* means *HSM Number* which is a unique identification of each HSM instance in the system. It is enumerated statically in *Inc/app_hsmn.h* with the help of macros:

```
#define APP_HSM \
    ...
    ADD_HSM(GPIO_IN, 5) \
    ADD_HSM(DEMO, 1) \
    ADD_HSM(GPIO_OUT, 1) \
    ...

#define ALIAS_HSM \
    ...
    ADD_ALIAS(USER_BTN,         GPIO_IN) \
    ADD_ALIAS(ACCEL_GYRO_INT,   GPIO_IN+1) \
    ADD_ALIAS(MAG_DRDY,         GPIO_IN+2) \
    ADD_ALIAS(HUMID_TEMP_DRDY,  GPIO_IN+3) \
    ADD_ALIAS(PRESS_INT,        GPIO_IN+4) \
    ADD_ALIAS(USER_LED,         GPIO_OUT) \
    ...
```

In the list above `DEMO` is the HSMN (HSM Number) of the Demo active object we used to demonstrate the statechart example in the PSiCC book. The constructor of Demo passes the HSMN `DEMO` along with the corresponding name string `"DEMO"` to the constructor of the intermediate base class *Active*:

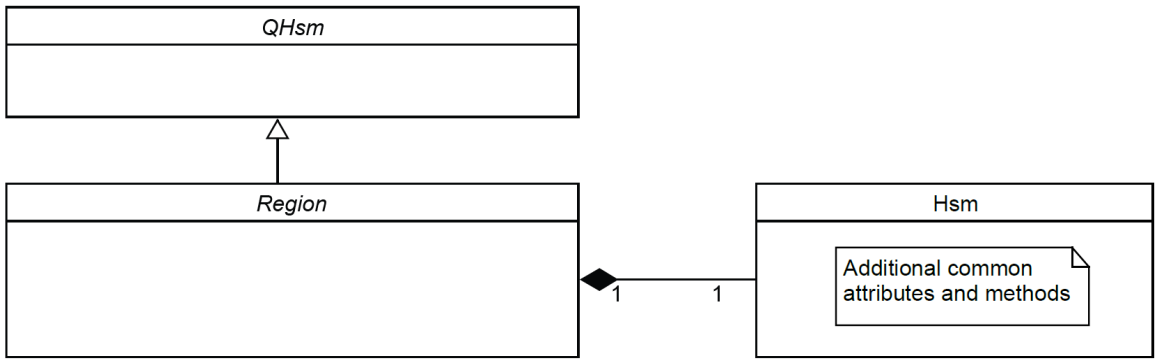
```
Demo::Demo() :
    Active((QStateHandler)&Demo::InitialPseudoState, DEMO, "DEMO"),
    m_stateTimer(GetHsmn(), STATE_TIMER), m_inEvt(QEvt::STATIC_EVT) {
    SET_EVT_NAME(DEMO);
}
```

We will see in the next section what *Active* is and how it uses the HSMN passed to its constructor.

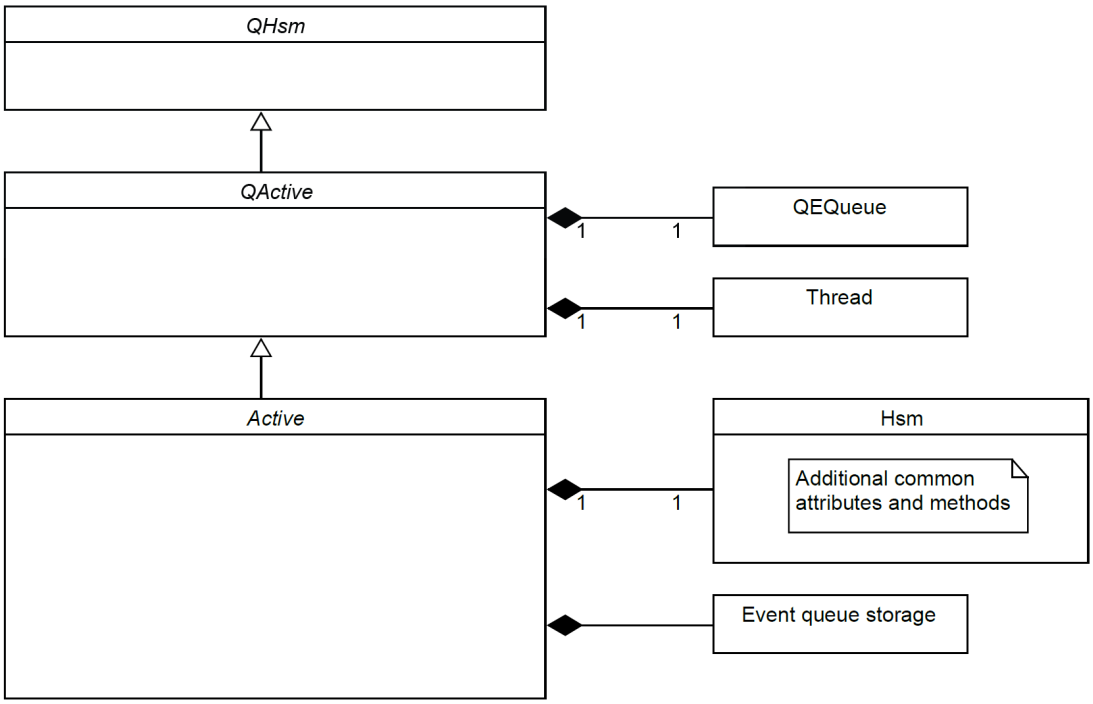
3.2 Region and Active

We introduced the class Hsm in the previous section, which is used to encapsulate common attributes and methods for all HSMs. We have also seen how QP uses the class QHsm to represent an HSM and how it derives QActive from QHsm to represent an active object.

Putting them all together, we derive a new class *Region* from QHsm, which allows us to extend the features of QHsm. One of the extensions is to *compose* an Hsm object to hold additional attributes and methods. The class name *Region* comes from the statechart concept of *orthogonal regions* which can be viewed as concurrent (parallel) hierarchical state-machines running in the same thread-context of an active object. In other words a Region *is an* QHsm composing an Hsm object as illustrated below:

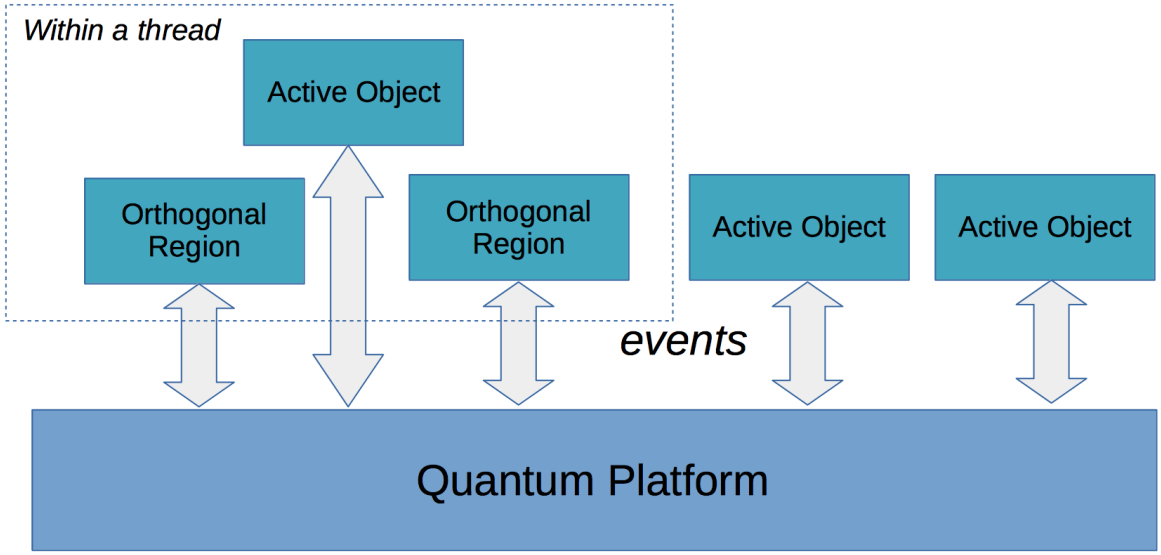


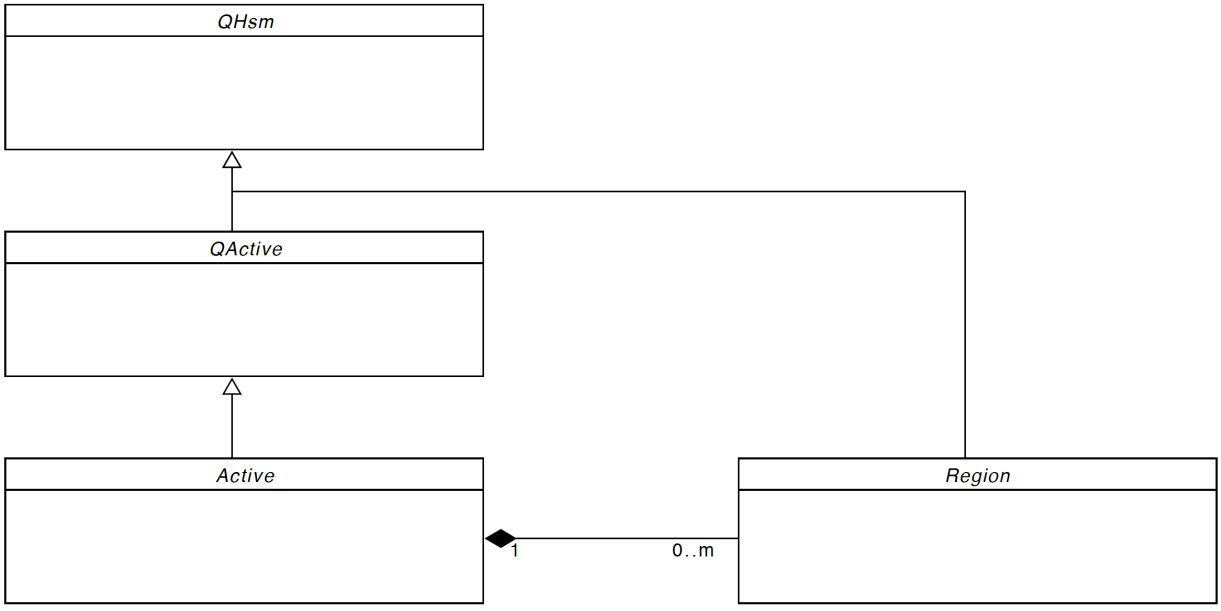
Similarly we extend QActive by deriving a new class *Active* from it. Active composes an Hsm object along with memory storage for the event queue making it easier to create an active object.



Now we have the class Region representing our extended hierarchical state-machine (or orthogonal region) and the class Active representing our extended active object. How about we combine them together? We can have an active object composing one or more Region objects to support concurrent state-machines running in the same thread, similar to orthogonal regions in a statecharts. (Note – An active object is by itself an HSM, and therefore you don't need to add a Region to it to implement a single HSM.)

The concept of orthogonal regions is illustrated in the block diagram and class diagram below:





For reference, the class definitions of Region and Active are listed below:

```
class Region : public QP::QHsm {
public:
    Region(QP::QStateHandler const initial, Hsmn hsmn, char const *name) :
        QP::QHsm(initial),
        m_hsm(hsmn, name, this),
        m_container(NULL) {}
    void Init(Active *container);
    void Init(XThread *container);
    Hsm &GetHsm() { return m_hsm; }
    virtual void dispatch(QP::QEvt const * const e);
    void Send(Evt *e);
    ...
protected:
    void PostFront(Evt const *e);
    void Raise(Evt *e);
    ...
    Hsm m_hsm;
    QP::QActive *m_container;
};


class Active : public QP::QActive {
public:
    Active(QP::QStateHandler const initial, Hsmn hsmn, char const *name) :
        QP::QActive(initial),
        m_hsm(hsmn, name, this),
        m_hsmnRegMap(m_hsmnRegStor, ARRAY_COUNT(m_hsmnRegStor), ...){
    }
    void Start(uint8_t prio);
    void Add(Region *reg);
    Hsm &GetHsm() { return m_hsm; }
    virtual void dispatch(QP::QEvt const * const e);
    void Send(Evt *e);
    ...
protected:
    void PostFront(Evt const *e);
    void Raise(Evt *e);
    ...
    enum {
        MAX_REGION_COUNT = 8,
        EVT_QUEUE_COUNT = 64
    };
    Hsm m_hsm;
    HsmnReg m_hsmnRegStor[MAX_REGION_COUNT];
    HsmnRegMap m_hsmnRegMap;
    QP::QEvt const *m_evtQueueStor[EVT_QUEUE_COUNT];
};
```

3.3 Event Interface

In an event driven system the definition of events is crucial and deserves deeper considerations. This includes the enumeration of event types and the corresponding event classes carrying event parameters. QP does not dictate how events are defined. In simple examples one global enumeration would suffice. However for larger projects this single list would be hard to maintain since it would easily contain more than a hundred events for the entire system.

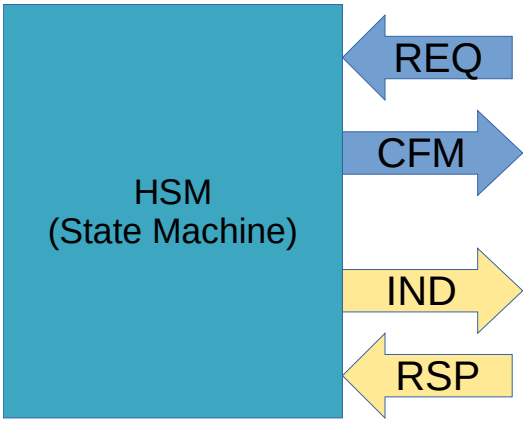
In many event driven systems the definition of events does not follow strong semantics. Most events use causal names like *SOMETHING_HAPPENED*, or *DO_SOMETHING*, etc. Due to the asynchronous nature of event driven systems, it can easily lead to race-conditions.

In our enhanced framework, we *partition* the event space among different HSM classes. Each HSM (either a Region or an Active object) *owns* three kinds of events, namely

1. Interface events
2. Internal events
3. Timer events

3.3.1 Interface events

Interface events are used for communications among HSMs. Their naming convention follows strong semantic rules to give them clear and precise meanings. Each event must fall into one of the four categories, namely *request*, *confirmation*, *indication* or *response*. A request forms a pair with a confirmation, while an indication forms a pair with a response. Together they define the external interface of an HSM through which it communicates with other HSMs.



Interface events are defined in the *interface file* of each HSM, such as DemoInterface.h. We use macros to partition the event space among all the HSMs to avoid conflicts, and to generate an event enumeration together with the corresponding event name strings automatically. Derived event classes are defined to carry additional event parameters. Simple events without additional parameters may just

use the *Evt* or *ErrorEvt* base class directly, though it won't hurt to create derived event classes for them as well. The following is an extract of DemoInterface.h:

```
// A list of interface events for Demo HSM. It is used to generate an enumeration
// along with the matching event name strings.
#define DEMO_INTERFACE_EVT \
    ADD_EVT(DEMO_START_REQ) \
    ADD_EVT(DEMO_START_CFM) \
    ADD_EVT(DEMO_STOP_REQ) \
    ADD_EVT(DEMO_STOP_CFM) \
    ADD_EVT(DEMO_A_REQ) \
    ADD_EVT(DEMO_B_REQ) \
    ADD_EVT(DEMO_C_REQ) \
    ADD_EVT(DEMO_D_REQ) \
    ADD_EVT(DEMO_E_REQ) \
    ADD_EVT(DEMO_F_REQ) \
    ADD_EVT(DEMO_G_REQ) \
    ADD_EVT(DEMO_H_REQ) \
    ADD_EVT(DEMO_I_REQ)

#undef ADD_EVT
#define ADD_EVT(e_) e_,

// Enumeration of events for Demo HSM.
enum {
    DEMO_INTERFACE_EVT_START = INTERFACE_EVT_START(DEMO),
    DEMO_INTERFACE_EVT
};

// A derived event class for event DEMO_START_REQ.
class DemoStartReq : public Evt {
public:
    enum {
        TIMEOUT_MS = 200
    };
    DemoStartReq() :
        Evt(DEMO_START_REQ) {}
};
...
```

Event names are generated in the source file, such as Demo.cpp:

```
#undef ADD_EVT
#define ADD_EVT(e_) #e_,

static char const * const interfaceEvtName[] = {
    "DEMO_INTERFACE_EVT_START",
    DEMO_INTERFACE_EVT
```

```
};
```

The constant string array `interfaceEvtName[]` is *automatically* populated by the macro DEMO_INTERFACE_EVT to store the event names matching the order of the event enumeration. It is then injected into the logging system (via the macro *SET_EVT_NAME* called from the constructor of Demo) to generate human-friendly log messages.

```
Demo::Demo() :
    Active((QStateHandler)&Demo::InitialPseudoState, DEMO, "DEMO"),
    m_stateTimer(GetHsmn(), STATE_TIMER), m_inEvt(QEvt::STATIC_EVT) {
    SET_EVT_NAME(DEMO);
}
```

3.3.2 Internal events

Internal events are used within an HSM. They are posted by an HSM to itself as *reminders*. They are used to break up a long event processing step (called *macro-step*) into multiple shorter steps (called *micro-steps*). While interface events must be posted to the *main event queue* of an active object, internal events can be posted to the *internal event queue* owned by each HSM (by calling *raise()*). Since an internal event queue has a higher priority than the main event queue, it is guaranteed that an HSM will process all internal events immediately, before any events in the main event queue. Internal events are defined in the protected scope of an HSM class, such as the following in Demo.h:

```
class Demo : public Active {
    ...
    #define DEMO_INTERNAL_EVT \
        ADD_EVT(DONE) \
        ADD_EVT(FAILED)

    #undef ADD_EVT
    #define ADD_EVT(e_) e_,

    enum {
        DEMO_INTERNAL_EVT_START = INTERNAL_EVT_START(DEMO),
        DEMO_INTERNAL_EVT
    };
    ...

    class Failed : public ErrorEvt {
    public:
        Failed(Error error, Hsmn origin, Reason reason) :
            ErrorEvt(FAILED, error, origin, reason) {}
    };
};
```

3.3.3 Timer events

Like internal events, timer events are used within an HSM. It is posted by QP to an HSM upon

expiration of a timer started by the same HSM. Timer events enable an HSM to wait or delay an action without blocking. They are defined in the protected scope of an HSM class, such as the following code in Demo.h:

```
class Demo : public Active {
    ...
    #define DEMO_TIMER_EVT \
        ADD_EVT(STATE_TIMER)

    #undef ADD_EVT
    #define ADD_EVT(e_) e_,

    enum {
        DEMO_TIMER_EVT_START = TIMER_EVT_START(DEMO),
        DEMO_TIMER_EVT
    };
    ...
};
```

3.3.4 Event Base Classes

The class *Evt* is the base class of all application events. It contains common attributes required by all application events:

```
class Evt : public QP::QEvt {
    ...
protected:
    Hsmn m_to;        // Destination HSM to receive the event.
    Hsmn m_from;      // Source HSM sending the event.
    Sequence m_seq;   // Sequence no. for matching REQ/IND with CFM/RSP.
};
```

The class *ErrorEvt* is a special kind of *Evt* used for confirmation (CFM) and response (RSP) events. It contains additional attributes to carry the result of an operation. Note a success is treated as a special kind of “error” with an error code of *ERROR_SUCCESS*.

```
class ErrorEvt : public Evt {
public:
    ...
protected:
    Error m_error;      // Common error code.
    Hsmn m_origin;      // HSM originating the error.
    Reason m_reason;    // CFM/RSP event specific reason.
};
```

3.3.5 API to Post Events

Recall that our application HSMs are derived from the class *Active* or *Region*. *Active* is the base class

of an active object which is a hierarchical state-machine with its own thread and main event queue, whereas *Region* is the base class of a parallel state machine attached to an active object (sharing its thread and main event queue). Each state-machine, either an active object or a region, owns an internal event queue for it to post reminder events to itself.

Both Active and Region provide a rich API for an application HSM to post events to another HSM or to itself. Since they are very similar, we will take Active as an example.

Since REQ/CFM pairs are similar in nature to IND/RSP pairs, we only shows the API for REQ/CFM events only.

1. Basic API to post an event to a destination HSM.

```
// Destination and seq no. specified in e.
void Send(Evt *e);
// Destination provided by parameter “to” with seq no. auto-generated.
void Send(Evt *e,  Hsmn to);
// Destination and seq no. provided by parameters.
void Send(Evt *e, Hsmn to, Sequence seq);
```

2. API to post a request or indication event to a destination HSM. It saves the destination HSM and sequence number of an outgoing request in an *event-sequence record* (of type EvtSeqRec). This record helps perform sequence number matching and check if confirmations to all outstanding requests have been received.

```
// Sends a request to the destination and saves event seq no. to “seqRec”.
// “reset” is set for the first event sent in a group.
void SendReq(Evt *e, Hsmn to, bool reset, EvtSeqRec &seqRec);
// Uses the built-in event-sequence record for convenience.
void SendReq(Evt *e, Hsmn to, bool reset);
```

3. API to check the result of a received confirmation event and if all expected confirmations have been received. The checking is done using the *event-sequence record* filled in when requests are sent.

```
// The function returns true if the confirmation event reports success or
// the event is ignored due to sequence number mismatch.
// The function returns false if sequence numbers mismatch and the event
// reports an error.
// “allReceived” returns true if all expected confirmations have been
// received without errors.
bool CheckCfm(ErrorEvt const &e, bool &allReceived, EvtSeqRec &seqRec);
// Uses the built-in event-sequence record for convenience.
bool CheckCfm(ErrorEvt const &e, bool &allReceived);
```

4. API to send confirmation to an incoming request after it has been processed.

```
// Sends a confirmation to the request “req”. It is used when the request
// event object is still available (within its event handler).
void SendCfm(Evt *e, Evt const &req);
// Sends a confirmation to a saved request event (outside its event handler).
void SendCfm(Evt *e, Evt &savedReq);
```

4 Design Heuristics

4.1 Observations

Event-driven systems are inherently asynchronous. If we are not careful we would easily run into race-conditions, resulting in components getting out-of-sync.

Consider the case in which the System Controller (A) sends a request MOTOR_START_REQ (REQ) to the Motor Controller (B). Think about the following design examples:

1. If A simply assumes B gets the REQ event immediately and successfully starts the motor, A and B will get out-of-sync if B was in an *inconvenient* state momentarily (e.g. *Stopping* state) and ignores the REQ, of if there is a hardware problem that prevents the motor to be started at all.
2. The above problem could be avoided by using a REQ/CFM pair, such that B will send the completion status back to A, and A will wait for the CFM event before moving on. (Note – At all time, A is not blocked, and therefore is able to process other events).
3. However, what if A does not get the CFM event from B? A could wait forever, which is obviously not a good design choice. Better options include starting a timer and upon timeout having A automatically resend the REQ to B, or report the error to the upper layer controller or to the end user, which/who may decide to retry.

   What if B's CFM to the first REQ has just been delayed, and now reaches A. A will think it's the CFM to the second REQ. Again this is a race-condition, as the parameters for the two requests may not the same (e.g. different speed, or worse different direction)!
4. The above problem can be mitigated by matching sequence numbers in REQ/CFM pairs, and that is why add a sequence number in our Evt base class.

4.2 Heuristics

Based on the above observation, we have come to the following design heuristics:

1. Handle all request events in all states.
2. Use a well-defined event interface with REQ/CFM and IND/RSP pairs.
3. Match sequence numbers in REQ/CFM and IND/RSP pairs.
4. Use timers in wait states.
5. Use defer and recall pattern. (See PSiCC book)
6. Use reminder pattern. (See PSiCC book)
7. Ensure requests entering safe-states (e.g. Stopped) are always accepted (i.e. must not be

rejected).

8. Use *hierarchical control pattern* to organize components (HSMs) in a system (see Gomaa's book, Chapter 11.3.4). This is similar to the *part-whole* layered architecture proposed by Pazzi's papers.

5 System Startup Case Study

System start-up and shut-down are often tricky to get right. Being able to stop and restart a system *entirely* within software is a good test for system robustness (requiring perfect initialization and cleanup) and facilitates test automation. Try out the command "**sys stop**" followed by "**sys start**" in our project. **Question:** Have you encountered or heard of system failures when starting or stopping an embedded system?

The *stop* command brings the entire system to a non-operational (*stopped)* state. The *start* command brings the system to an operational (*started*) state again. By design, the stop request should not fail, or it would trigger an assertion as a last resort of recovery.

Review the statechart design of the *System* active object to see how some of the design heuristics mentioned above are applied.

6 Reference

1. Modeling Rover Communication Using Hierarchical State Machines with Scala. Klaus Havelund. September 2017. (http://rjoshi.org/bio/papers/tips-2017.pdf)
2. Towards the Hierarchical State Machine OrientedProteus Systems Programming Language. Klaus Havelund, etc. 2020. (http://www.havelund.com/Publications/ascend-proteus-2020.pdf)
3. Systems of Systems Modeled by a Hierarchical Part-Whole State-Based Formalism. Luca Pazzi. June 2013. (https://www.researchgate.net/publication/236156084_Systems_of_Systems_Modeled_by_a_Hierarchical_Part-Whole_State-Based_Formalism)