

EMBSYS 110 Module 4

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. This time we will use real-life examples to show how these techniques can be applied to practical embedded projects. We will start by building 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 or *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.

1.2 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:

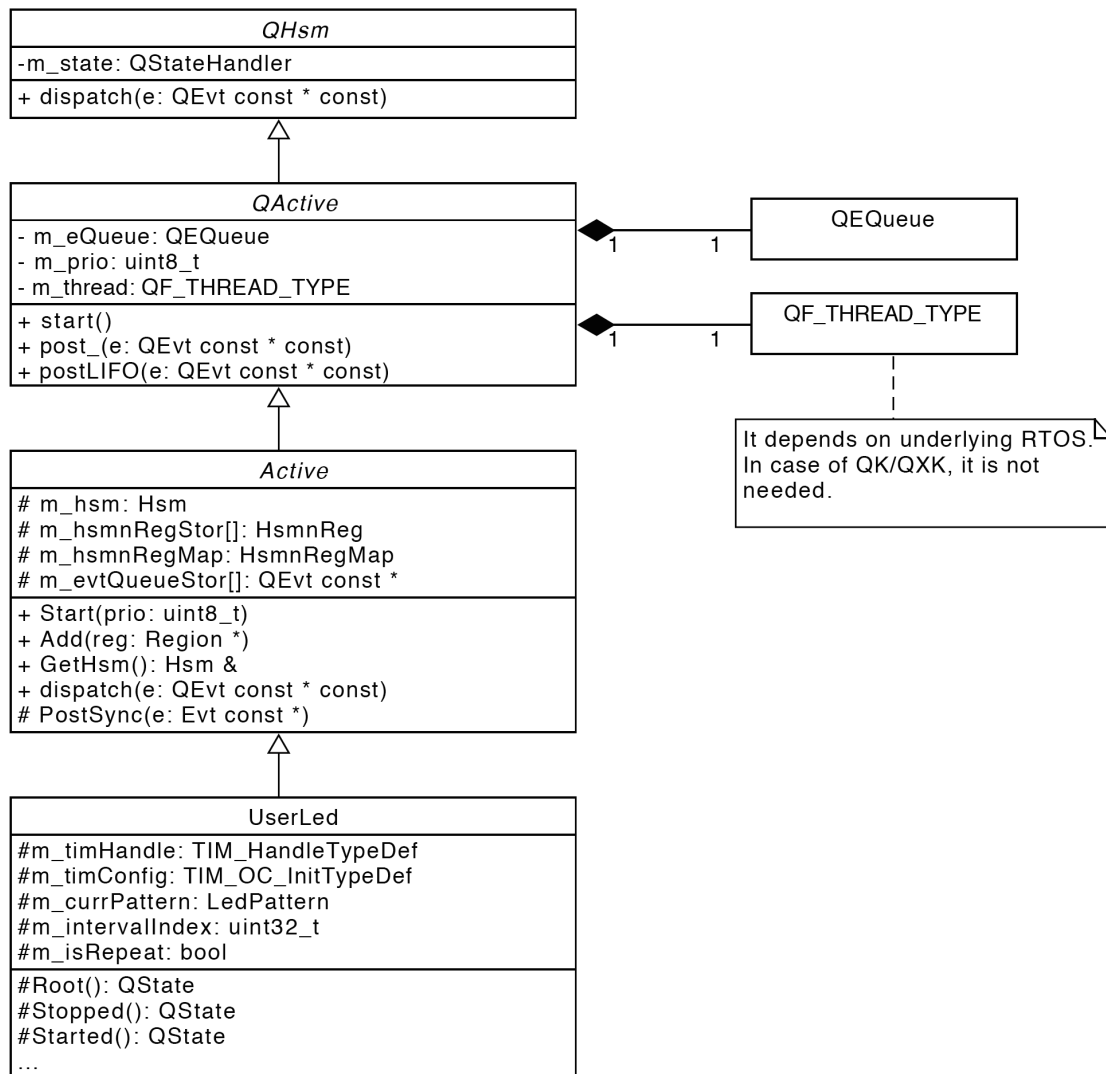
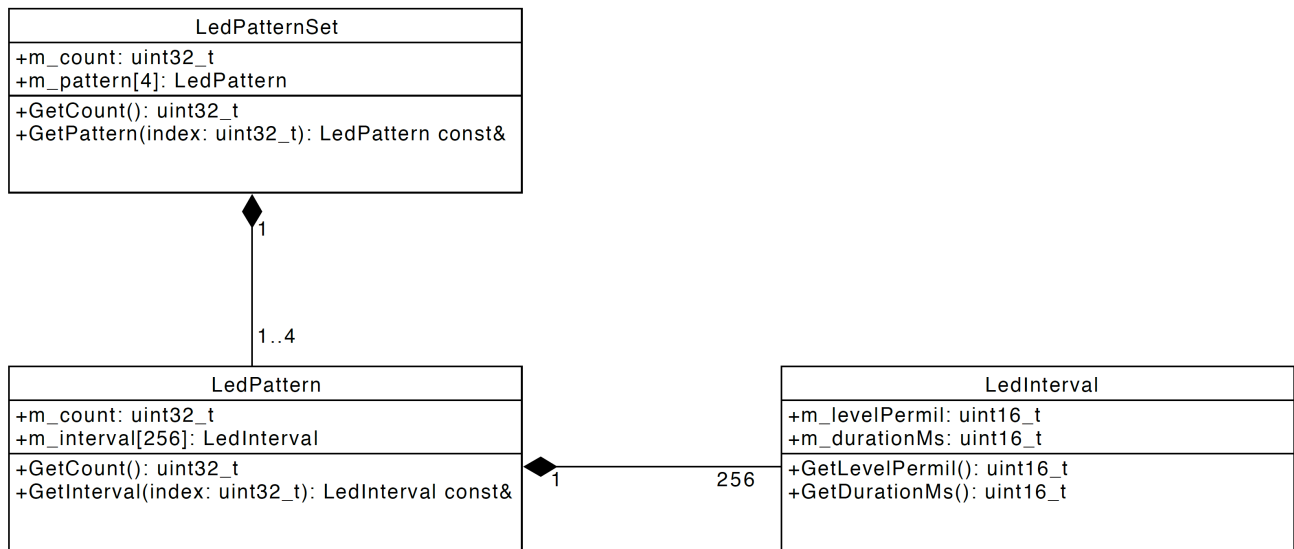
1. A set of LED patterns and the patterns contained by the set.
2. An LED pattern and the intervals contained by the pattern.

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.



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 rectangular 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.

1.3.2 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.

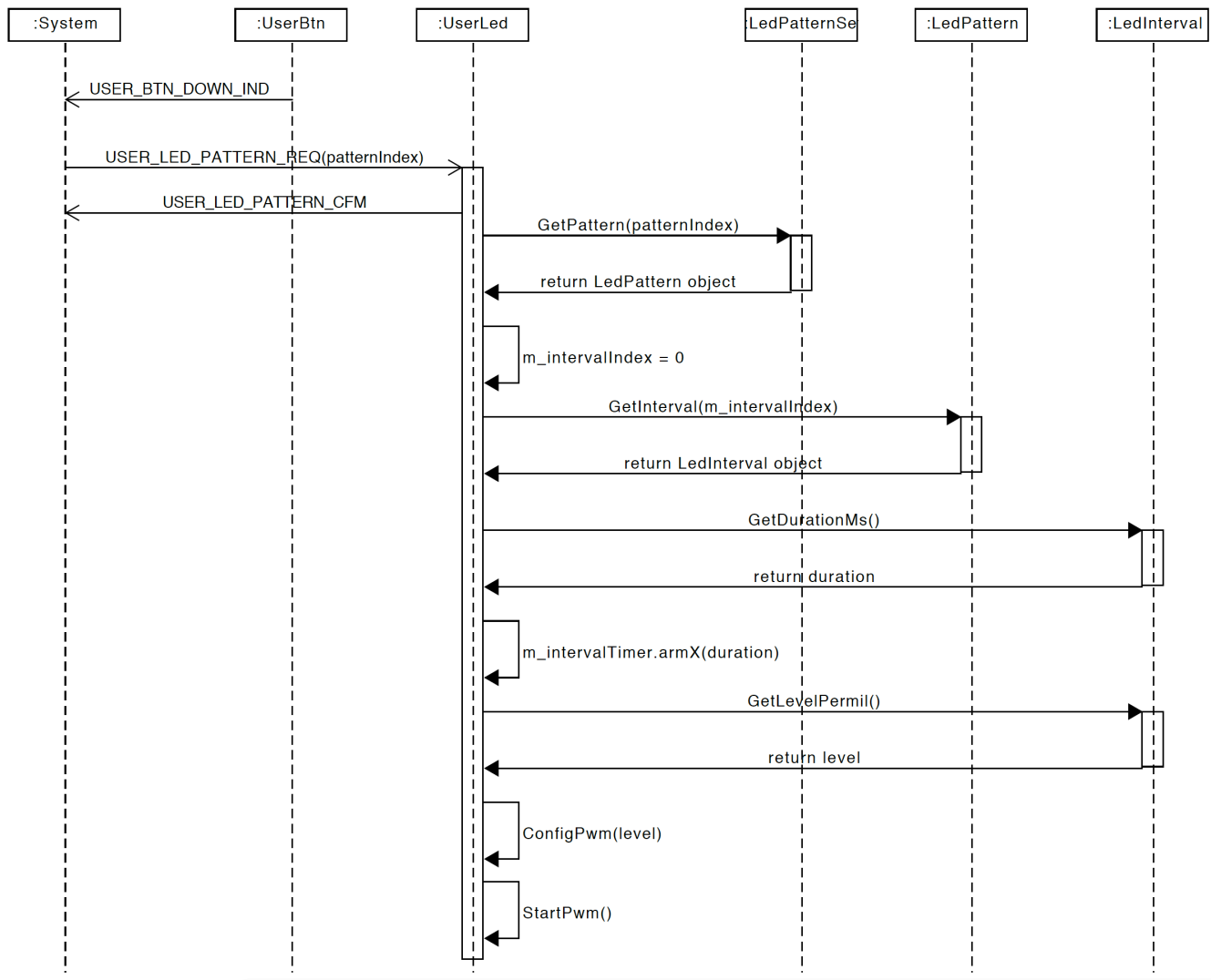
It does not show what happens if the order of received event is different or a failure occurs at a different place. In other words it does not show a *complete* picture of the system being modeled, rather like showing a small sample of cross-sectional views of a 3D physical model. You may need an astronomical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

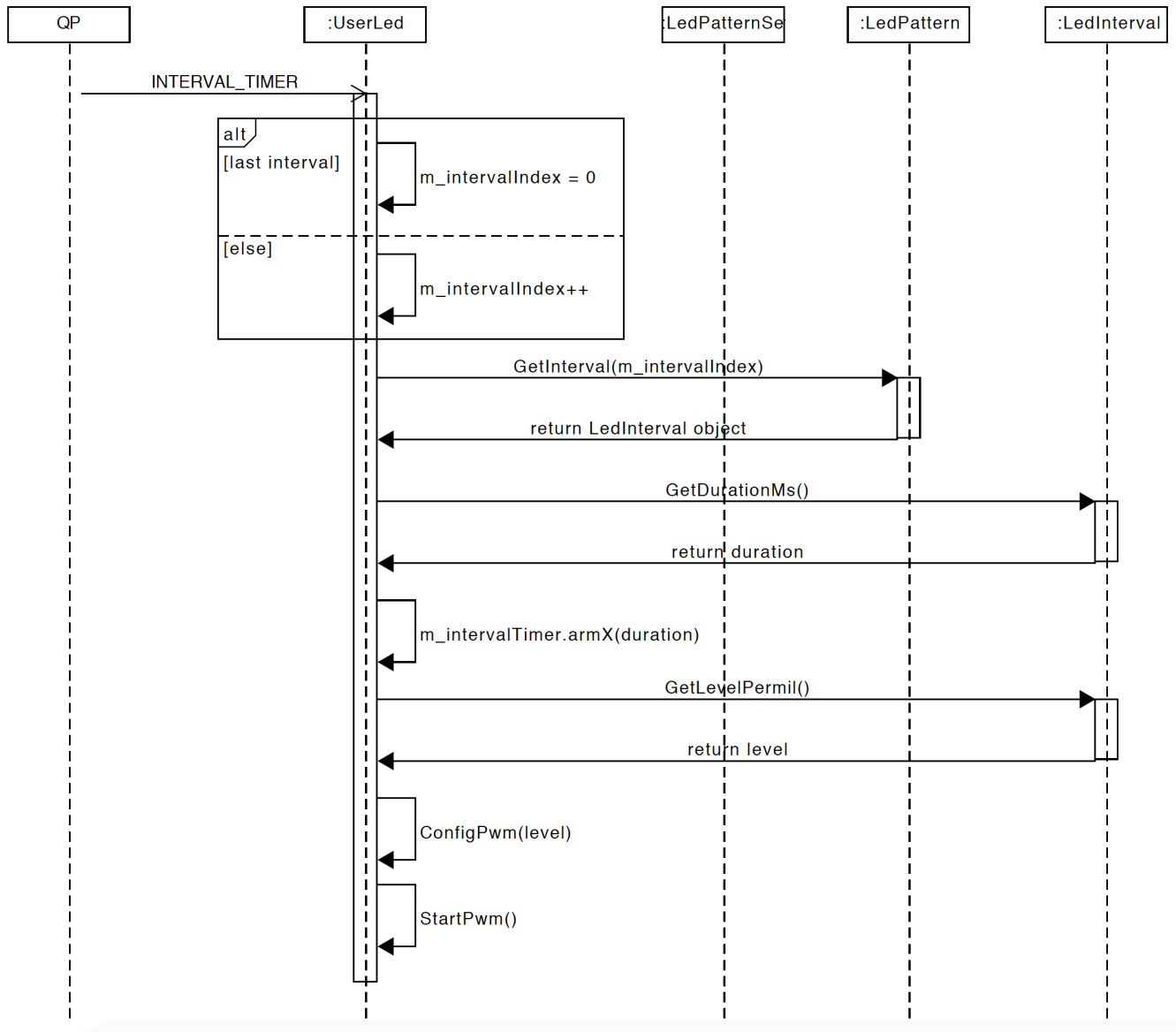
One may be tempted to start coding based on one or two sequence diagrams about normal cases. When

the need arises for us to worry about failure cases, we can then refactor the code or if under schedule pressure simply hack it. 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.

1.3.3 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.





2 Application Framework with QP

2.1 Basic Framework

2.1.1 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 rules defined in a 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 member function *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 will it be more useful if we can extend QHsm to become active? It is a very attractive idea, and without a better name we call such extended object an **active object** represented by *QActive* in QP.

2.1.2 Active Object

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

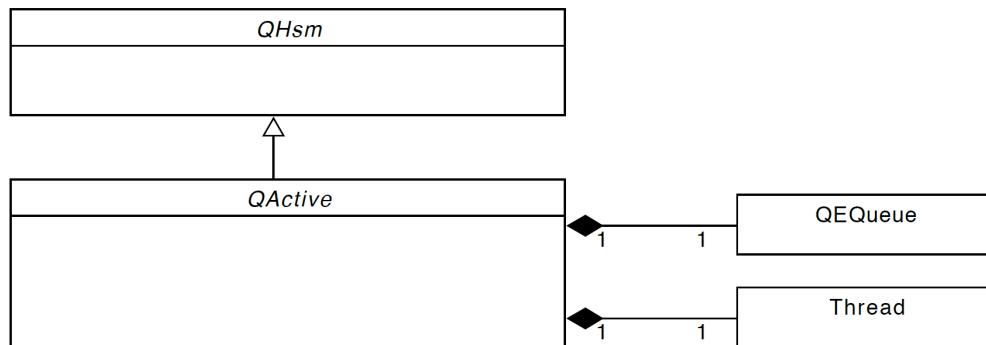
Just think about how you would normally do when using an RTOS. Mostly likely you would create a thread and have it block on an event queue for events to arrive. As long as the queue is non-empty, the thread will get the oldest event out of the queue to process. After each event it will loop back to check 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) {
        QEvt const *e = eventQueue.GetEvent(); // Event loop.
        switch (e->sig) { // Blocking call.
            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 you recall we have talked about this relationship briefly when we studied class diagrams (about *UserLed*). Now we should understand what it is all about.



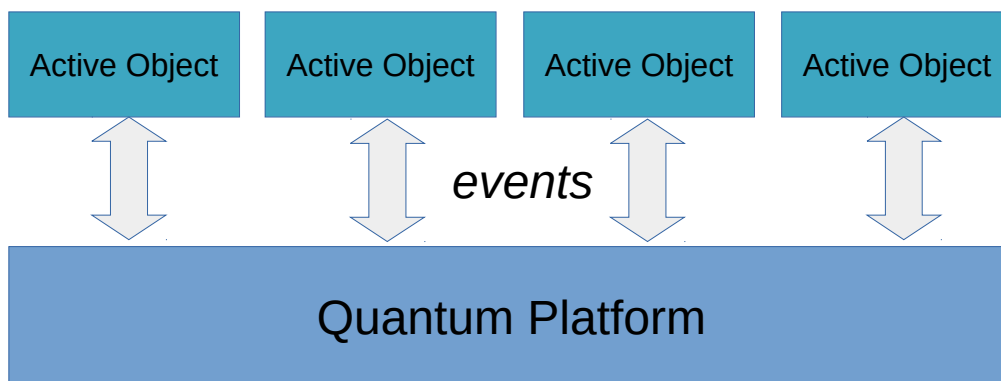
2.1.3 Publish-subscribe

Now we have learned an active object being a fundamental system component. It is natural to extend this idea to contemplate a software system as a collection (or a *partition as in set-theory*) of multiple active objects. Each active object is responsible for a certain area of concern, and hence yielding desirable separation of concerns or decoupling. They communicate with each other in one way or the other, and together they define the overall system behaviors.

It surely sounds simple, doesn't it? However it leads to two important questions:

1. How should we partition the system into active objects?
2. How should active objects communicate with each other?

It turns out these are difficult questions. They are difficult because they are very abstract. They are difficult because there are more than one ways of doing it, and it's hard to measure which way is optimal. Let's look at some common approaches suitable for real-time embedded systems. The first one to look at is the publish-subscribe model provided by QP as illustrated below.



In the publish-subscribe model, active objects communicate with each other via events. This is one kind of event-driven systems. QP serves as an event broker in this model. 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 (a.k.a lifetime management) can be complicated since it can easily cause dangling pointers or memory leak issues.

An active object publishes an event via this QP API:

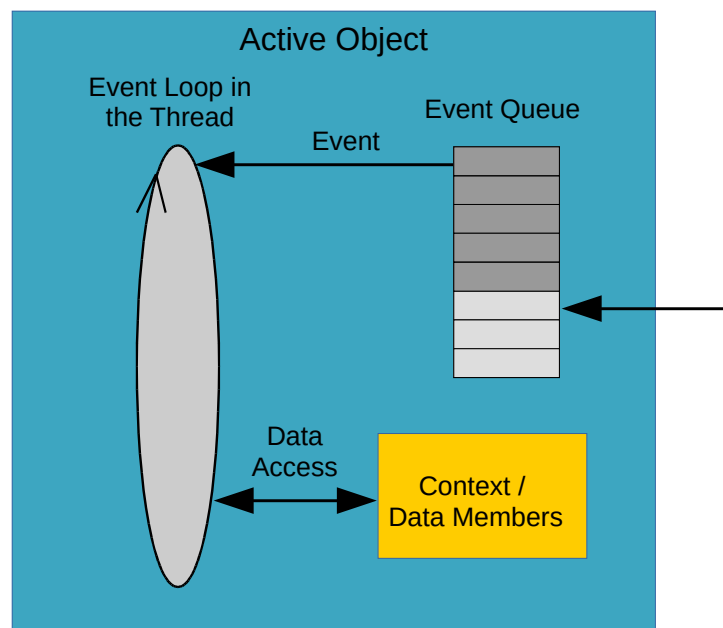
```
QF::publish_(QEvt const * const e)
```

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 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 disappeared. The diagram below illustrates the concept of a localized context per active object.



Nevertheless, the broadcast nature of publish-subscribe can be problematic if we need to ensure reliability, since a publisher does not know who the subscribers are. It does not know if there is *none*, *one* or *multiple* recipients of a published event, and therefore it can be difficult to implement acknowledgement or return-event-path for guaranteed delivery or request-reply handshaking. If not careful one might be caught by race-condition issues that are very hard to track down.

2.2 Advanced Framework

QP provides a higher level of abstraction over tradition RTOS primitives such as threads, queues, mutexes, etc. It allows us to focus on the more important aspects, i.e. the actual behaviors, rather than the low level interactions such as when we need to lock a mutex, whether a function call will block, what happens if one thread updates a data set while it is being shared by other threads, etc.

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 a lot of external hooks and apply usage patterns. 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 debugging tool called QSPY, you may still want to add your own that is customized for your need.

To provide a slightly higher level of abstraction on top of QP, we introduce our own application framework layer, simply called *fw* in the name space of *FW*. You will find its source code under the folder *framework*.

2.2.1 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 perfect chance to apply encapsulation and use a class to contain all these common attributes.

We call this class *Hsm*, meaning that it contains attributes that are common to all hierarchical state-machines. It is defined in *framework/include/fw_hsm.h*. Below is an extract of the class definition:

```

class Hsm {
protected:
    enum {
        OUT_HSMN_SEQ_MAP_COUNT = 8,
        DEFER_QUEUE_COUNT = 16,
        REMINDER_QUEUE_COUNT = 4
    };
    Hsmn m_hsmn;
    char const * m_name;
    QP::QHsm *m_qhsm;
    char const *m_state;
    EvtName m_timerEvtName;
    EvtCount m_timerEvtCount;
    EvtName m_internalEvtName;
    EvtCount m_internalEvtCount;
    EvtName m_interfaceEvtName;
    EvtCount m_interfaceEvtCount;
    ...
}

```

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

```

#define APP_HSM \
    ADD_HSM(SYSTEM, 1) \
    ADD_HSM(CONSOLE, 2) \
    ...
    ADD_HSM(DEMO, 1) \
    ...
#define ALIAS_HSM \
    ADD_ALIAS(CONSOLE_UART2,    CONSOLE+0) \
    ADD_ALIAS(CONSOLE_UART1,    CONSOLE+1) \
    ...

```

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",
        timerEvtName, ARRAY_COUNT(timerEvtName),
        internalEvtName, ARRAY_COUNT(internalEvtName),
        interfaceEvtName, ARRAY_COUNT(interfaceEvtName)),
    m_stateTimer(GetHsm().GetHsmn(), STATE_TIMER) {}

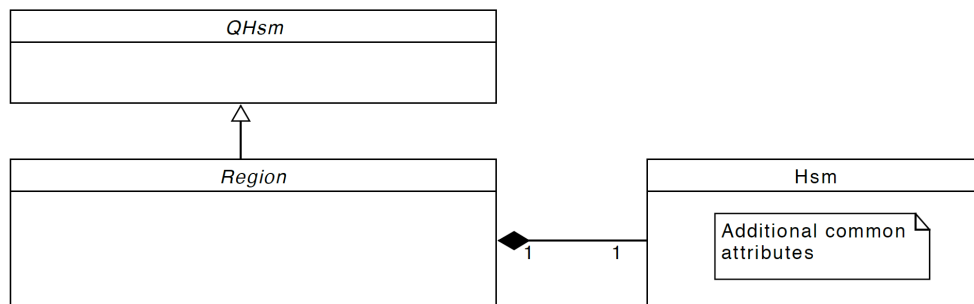
```

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

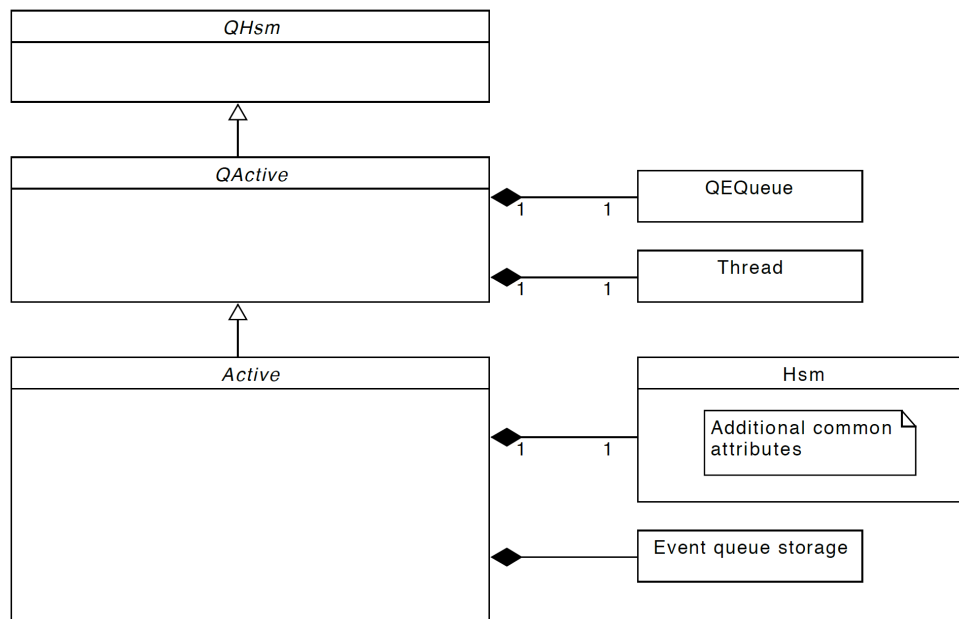
2.2.2 Active and Region

We introduced the class `Hsm` in the previous section, which is used to hold common attributes 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 all the additional attributes. The class name *Region* comes from the statechart concept of *orthogonal regions* which can be viewed as concurrent 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 in the class diagram below:

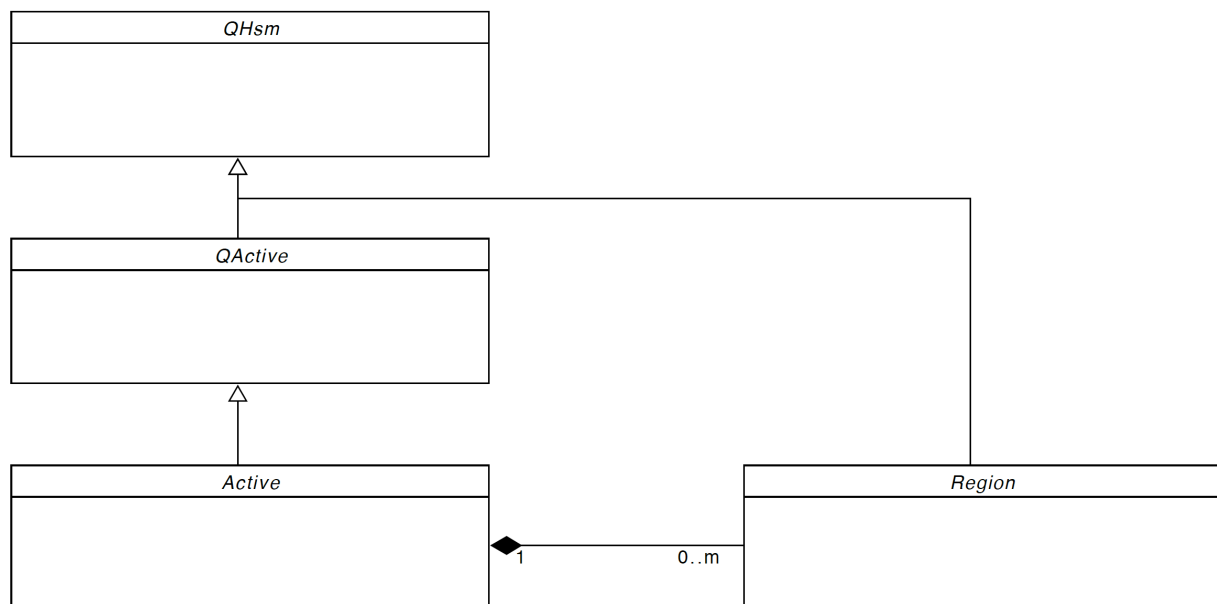
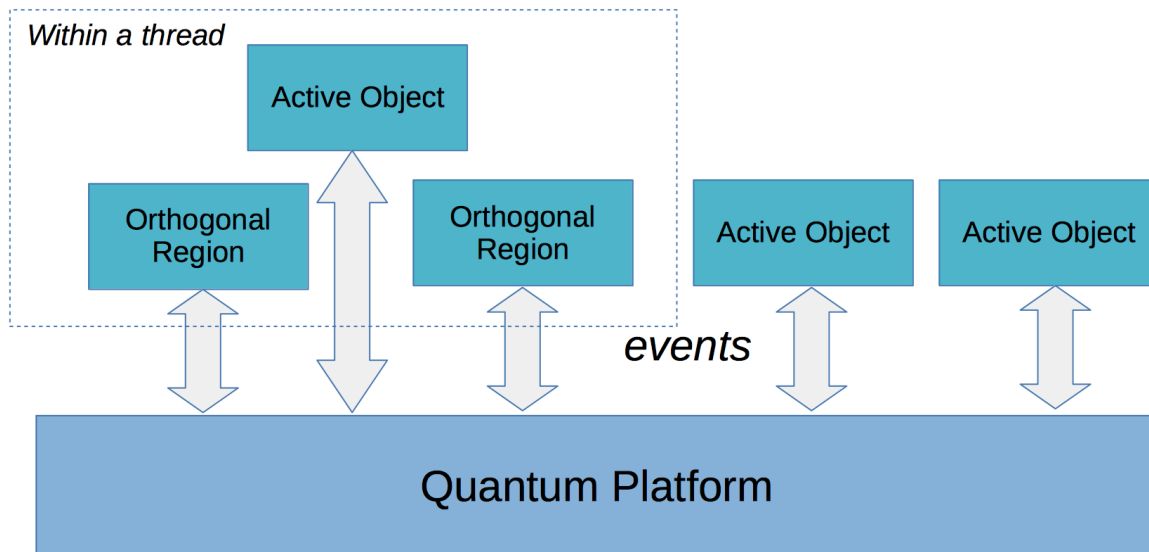


Similarly we extend `QActive` by deriving a new class *Active* from it. *Active* composes an `Hsm` object along with memory storage for the event queue making it easier to create an active object.



Now we have class Region representing our extended hierarchical state-machine (or orthogonal region) and 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);  
  
protected:  
    void PostSync(Evt const *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);  
  
protected:  
    void PostSync(Evt const *e);  
    enum {  
        MAX_REGION_COUNT = 8,  
        EVT_QUEUE_COUNT = 16  
    };  
    Hsm m_hsm;  
    HsmnReg m_hsmnRegStor[MAX_REGION_COUNT];  
    HsmnRegMap m_hsmnRegMap;  
    QP::QEvt const *m_evtQueueStor[EVT_QUEUE_COUNT];  
};
```

2.2.3 Event Interface

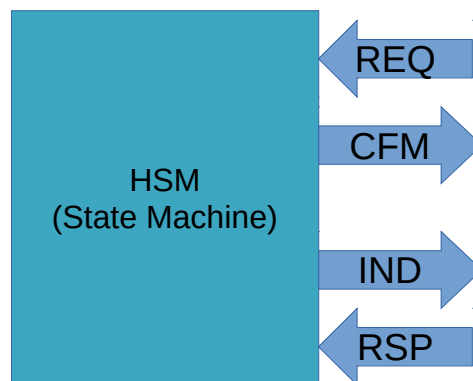
In an event driven system the definition of events is crucial and deserves deeper considerations. This includes the enumeration of event types and the corresponding event classes to carry event parameters. QP does not dictate how events are defined. In simple examples one global enumeration would suffice. However for larger projects this single list can 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 advanced framework, we require each HSM (being a Region or an Active object) to define its own events. For each HSM there are three different kinds of events:

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 an *interface file* for each HSM, such as DemoInterface.h. We use macros to generate event names automatically and to partition the event space among all HSMs in the system. If additional event parameters are needed, dedicated event classes will be defined in the interface file as well.

```
#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_STOP_IND) \  
    ADD_EVT(DEMO_STOP_RSP)
```



```

    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_,

enum {
    DEMO_INTERFACE_EVT_START = INTERFACE_EVT_START(DEMO),
    DEMO_INTERFACE_EVT
};

class DemoStartReq : public Evt {
public:
    enum {
        TIMEOUT_MS = 200
    };
    DemoStartReq(Hsmn to, Hsmn from, Sequence seq) :
        Evt(DEMO_START_REQ, to, from, seq) {}
};
...

```

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
};

```

2. Internal events

Internal events are used within an HSM. They are posted by the same HSM that processes them. They are mainly used as *reminders* to itself. While interface events must be posted in FIFO order, internal events can be posted in either FIFO or LIFO order. When an event is posted in LIFO order, it is guaranteed that it will be the next one to be processed and it can be called a synchronous event. Internal events are defined in the protected scope of an HSM class, such as the following code in Demo.h:

```

#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
};

```

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:

```

#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
};

```