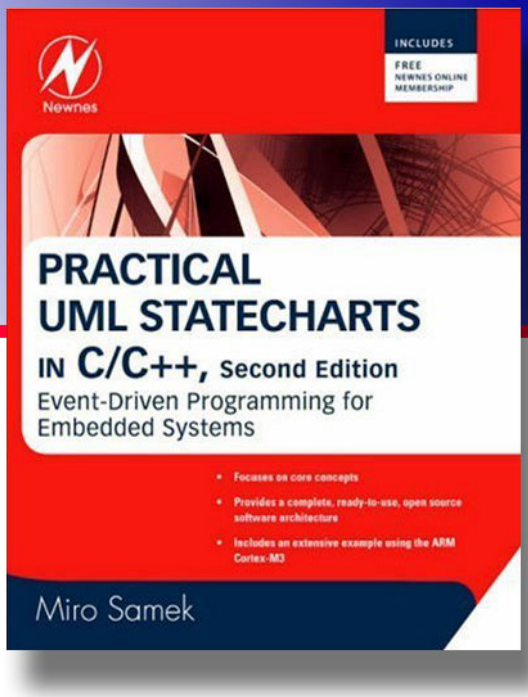
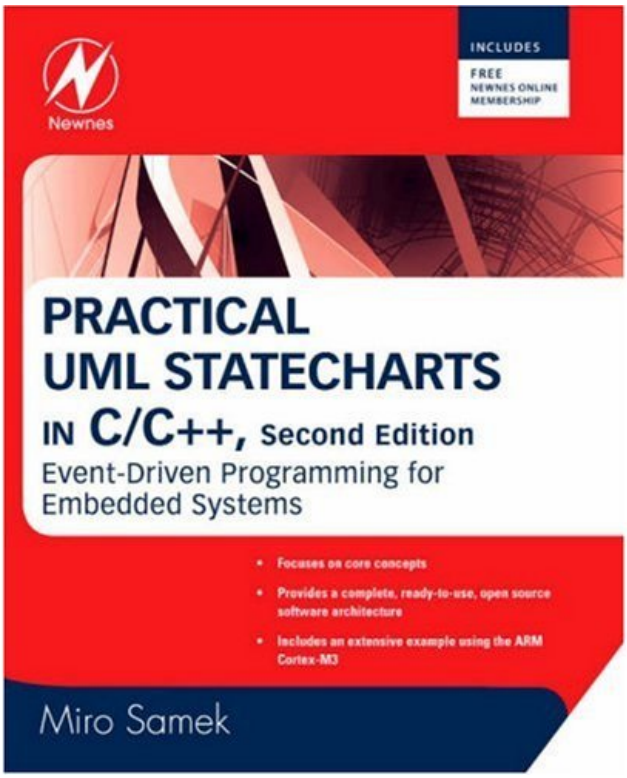
Quantum™ LeaPs
innovating embedded systems



Design Pattern Reminder

Document Revision B
September 2008

www.quantum-leaps.com
www.state-machine.com



The following excerpt comes from the book *Practical UML Statecharts in C/C++, 2nd Ed: Event-Driven Programming for Embedded Systems* by Miro Samek, Newnes 2008.

ISBN-10: 0750687061
ISBN-13: 978-0750687065

Reminder

Intent

Make the statechart topology more flexible by inventing an event and posting it to self.

Problem

Often in state modeling, loosely related functions of a system are strongly coupled by a common event. Consider, for example, periodic data acquisition, in which a sensor producing the data needs to be polled at a predetermined rate. Assume that a periodic TIMEOUT event is dispatched to the system at the desired rate to provide the stimulus for polling the sensor. Because the system has only one external event (the TIMEOUT event), it seems that this event needs to trigger both the polling of the sensor and the processing of the data. A straightforward but suboptimal solution is to organize the state machine into two distinct orthogonal regions (for polling and processing)[1]. However, orthogonal regions increase the cost of dispatching events (see the "Orthogonal Component" pattern) and require complex synchronization between the regions because polling and processing are not quite independent.

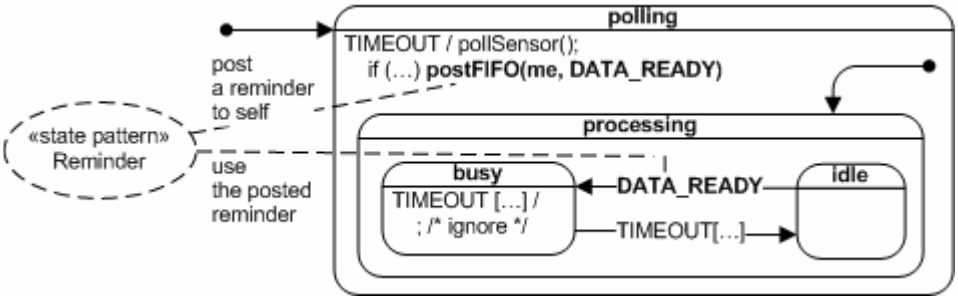
Solution

A simpler and more efficient solution is to invent a stimulus (DATA_READY) and to propagate it to self as a reminder that the data is ready for processing (Figure 5.2). This new stimulus provides a way to decouple polling from processing without using orthogonal regions. Moreover, you can use state nesting to arrange these two functions in a hierarchical relation[2], which gives you even more control over the behavior.

[1] This example illustrates an alternative design for the Polling state pattern described in [Douglass 99].
[2] Using state hierarchy in this fashion is typically more efficient than using orthogonal regions.

In the most basic arrangement, the "processing" state can be a substate of "polling" and can simply inherit the "polling" behavior so that polling occurs in the background to processing. However, the processing state might also choose to override polling. For instance, to prevent flooding the CPU with sensor data, processing might inhibit polling occasionally. The statechart in Figure 5.2 illustrates this option. The "busy" substate of processing overrides the TIMEOUT event and thus prevents this event from being handled in the higher level polling superstate.

Figure 5.3 The Reminder state pattern.



Further flexibility of this solution entails fine control over the generation of the invented DATA_READY event, which does not have to be posted at every occurrence of the original TIMEOUT event. For example, to improve performance, the polling state could buffer the raw sensor data and generate the DATA_READY event only when the buffer fills up, Figure 5.2 illustrates this option with the `if(…)` condition, which precedes the `postFIFO(me, DATA_READY)` action in the "polling" state.

Sample Code

The sample code for the Reminder state pattern is found in the directory `qpc\examples\80x86\dos\tcpp101\l\reminder\`. You can execute the application by double-clicking on the file `REMINDER.EXE` file in the `dbg\` subdirectory. Figure 5.4 shows the output generated by the `REMINDER.EXE` application. The application prints every state entry (to "busy" or "idle"), as well as the number of times the TIMEOUT event has been handled in "polling" and "processing", respectively. Listing 5.2 shows the example implementation of the Reminder pattern from Figure 5.3.

Figure 5.4 Output generated by `REMINDER.EXE`.

The Reminder state pattern posts the reminder event to self. This operation involves event queuing and is not supported by the raw QEP event processor. Therefore the sample code uses the QEP event processor as well as the QF real-time framework, which are both components of the QP event-driven framework. The QF component provides event queuing as well as the time events, both of which are used in the sample code.

Listing 5.2 The Reminder sample code (file `reminder.c`).

```
(1) #include "qp_port.h"                                        /* QP interface */
    #include "bsp.h"                                  /* board support package */

    enum SensorSignals {
        TIMEOUT_SIG = Q_USER_SIG,                  /* the periodic timeout signal */
(2)     DATA_READY_SIG,                           /* the invented reminder signal */
        TERMINATE_SIG                               /* terminate the application */
    };
    /*.....................................................................*/
    typedef struct SensorTag {                        /* the Sensor active object */
(3)     QActive super;                                     /* derive from QActive */

(4)     QTimeEvt timeEvt;                          /* private time event generator */
        uint16_t pollCtr;
        uint16_t procCtr;
    } Sensor;

    void Sensor_ctor(Sensor *me);
                                                /* hierarchical state machine ... */
    QState Sensor_initial   (Sensor *me, QEvent const *e);
    QState Sensor_polling   (Sensor *me, QEvent const *e);
    QState Sensor_processing(Sensor *me, QEvent const *e);
    QState Sensor_idle      (Sensor *me, QEvent const *e);
    QState Sensor_busy      (Sensor *me, QEvent const *e);
    QState Sensor_final     (Sensor *me, QEvent const *e);

    /*.....................................................................*/
    void Sensor_ctor(Sensor *me) {
        QActive_ctor_(&me->super, (QStateHandler)&Sensor_initial);
(5)     QTimeEvt_ctor(&me->timeEvt, TIMEOUT_SIG);            /* time event ctor */
    }
    /* HSM definition -----------------------------------------------------*/
    QState Sensor_initial(Sensor *me, QEvent const *e) {
        me->pollCtr = 0;
        me->procCtr = 0;
(6)     return Q_TRAN(&Sensor_polling);
    }
    /*.....................................................................*/
    QState Sensor_final(Sensor *me, QEvent const *e) {
        switch (e->sig) {
            case Q_ENTRY_SIG: {
                printf("final-ENTRY;\nBye!Bye!\n");
                BSP_exit();                          /* terminate the application */
                return Q_HANDLED();
            }
        }
        return Q_SUPER(&QHsm_top);
    }
    /*.....................................................................*/
```

```
    QState Sensor_polling(Sensor *me, QEvent const *e) {
        switch (e->sig) {
            case Q_ENTRY_SIG: {
                                              /* periodic timeout every 1/2 second */
(7)             QTimeEvt_postEvery(&me->timeEvt, (QActive *)me,
                                   BSP_TICKS_PER_SEC/2);
                return Q_HANDLED();
            }
            case Q_EXIT_SIG: {
                QTimeEvt_disarm(&me->timeEvt);
                return Q_HANDLED();
            }
            case Q_INIT_SIG: {
                return Q_TRAN(&Sensor_processing);
            }
(8)         case TIMEOUT_SIG: {
                static const QEvent reminderEvt = { DATA_READY_SIG, 0 };
                ++me->pollCtr;
                printf("polling %3d\n", me->pollCtr);
                if ((me->pollCtr & 0x3) == 0) {                   /* modulo 4 */
(9)                 QActive_postFIFO((QActive *)me, &reminderEvt);
                }
                return Q_HANDLED();
            }
            case TERMINATE_SIG: {
                return Q_TRAN(&Sensor_final);
            }
        }
        return (Q_SUPER(&QHsm_top);
    }
    /*..........................................................................*/
    QState Sensor_processing(Sensor *me, QEvent const *e) {
        switch (e->sig) {
            case Q_INIT_SIG: {
                return Q_TRAN(&Sensor_idle);
            }
        }
        return Q_SUPER(&Sensor_polling);
    }
    /*..........................................................................*/
    QState Sensor_idle(Sensor *me, QEvent const *e) {
        switch (e->sig) {
            case Q_ENTRY_SIG: {
                printf("idle-ENTRY;\n");
                return Q_HANDLED();
            }
            case DATA_READY_SIG: {
(10)            return Q_TRAN(&Sensor_busy);
            }
        }
        return Q_SUPER(&Sensor_processing);
    }
    /*..........................................................................*/
    QState Sensor_busy(Sensor *me, QEvent const *e) {
        switch (e->sig) {
            case Q_ENTRY_SIG: {
                printf("busy-ENTRY;\n");
```

```
                return Q_HANDLED();
            }
(11)        case TIMEOUT_SIG: {
                ++me->procCtr;
                printf("processing %3d\n", me->procCtr);
                if ((me->procCtr & 0x1) == 0) {                    /* modulo 2 */
                    Q_TRAN(&Sensor_idle);
                }
                return Q_HANDLED();
            }
        }
        return Q_SUPER(&Sensor_processing);
    }
```

(1) The Reminder state pattern posts the reminder event to self. This operation involves event queuing and is not supported by the raw QEP event processor. The sample code uses the whole QP, which includes the QEP event processor and the QF real-time framework. QF provides event queuing as well as the time events, both of which are used in the sample code.

NOTE: Event queuing and event-driven timing services are available in virtually every event-driven infrastructure. For instance, Windows GUI applications can call the `PostMessage()` Win32 API to queue messages and the `WM_TIMER` message to receive timer updates.

(2) The invented reminder event signal (DATA_READY in this case) is enumerated just like all other signals in the system.

(3) The `Sensor` state machine derives from the QF class `QActive` that combines a HSM, an event queue, and a thread of execution. The `QActive` class actually derives from `QHsm`, which means that `Sensor` also indirectly derives from `QHsm`.

(4) The `Sensor` state machine declares its own private time event. Time events are managed by the QF real-time framework. Section 7.7 in Chapter 7 covers the `QTimeEvt` facility in detail.

(5) The time event must be instantiated, at which time it gets permanently associated with the given signal (TIMEOUT_SIG in this case).

(6) The top-most initial transition enters the “polling” state, which in turn enters the “idle” substate.

(7) Upon the entry to the “polling” state the time event is armed for generating periodic TIMEOUT_SIG events twice per second.

NOTE: In QF, as in every other RTOS, the time unit is the “time tick”. The board support package (BSP) defines the constant `BSP_TICKS_PER_SEC` that ties the ticking rate to the second.

(8) After being armed, the time event produces the TIMEOUT_SIG events at the programmed rate. Because neither the “idle” state nor the “processing” state handle the TIMEOUT_SIG signal, the signal is handled initially in the “polling” superstate.

(9) At a lower rate (every fourth time in this example), the “polling” state generates the reminder event (DATA_READY), which it posts to self. Event posting occurs by calling `QActive_postFIFO()` function provided in the QF real-time framework.

(10) The reminder event causes a transition from “idle” to “busy”.

(11) The “busy” state overrides the DATA_READY_SIG signal and after a few TIMEOUT events transitions back to “idle”. The cycle then repeats.

Consequences

Although conceptually very simple, the Reminder state pattern has profound consequences. It can address many more problems than illustrated in the example. You could use it as a Swiss Army knife to fix almost any problem in the state machine topology.

For example, you also can apply the Reminder idiom to eliminate troublesome completion transitions, which in the UML specification are transitions without an explicit trigger (they are triggered implicitly by completion events, a.k.a. anonymous events). The QEP event processor requires that all transitions have explicit triggers; therefore, the pattern does not support completion transitions. However, the Reminder pattern offers a workaround. You can invent an explicit trigger for every transition and post it to self. This approach actually gives you much better control over the behavior because you can explicitly specify the completion criteria.

Yet another important application of the Reminder pattern is to break up longer RTC steps into shorter ones. As explained in more detail in Chapter 6, long RTC steps exacerbate the responsiveness of a state machine and put more stress on event queues. The Reminder pattern can help you break up CPU-intensive processing (e.g., iteration) by inventing a stimulus for continuation in the same way that you stick a Post-It note to your computer monitor to remind you where you left off on some lengthy task when someone interrupts you. You can also invent event parameters to convey the context, which will allow the next step to pick up where the previous step left off (e.g., the index of the next iteration). The advantage of fragmenting lengthy processing in such a way is so that other (perhaps more urgent) events can "sneak in" allowing the state machine to handle them in a more timely way.

You have essentially two alternatives when implementing event posting: the first in, first out (FIFO) or the last in, first out (LIFO) policy, both of which are supported in the QF real-time framework (see Chapter 6). The FIFO policy is appropriate for breaking up longer RTC steps. You want to queue the reminder event after other events that have potentially accumulated while the state machine was busy, to give the other events a chance to sneak in ahead of the reminder. However, in other circumstances, you might want to process an uninterruptible sequence of posted events (such a sequence effectively forms an extended RTC step[3]). In this case, you need the LIFO policy, because a reminder posted with that policy is guaranteed to be the next event to process and no other event can overtake it.

NOTE: You should always use the LIFO policy with great caution because it changes the order of events. In particular, if multiple events are posted with the LIFO policy to an event queue and no events are removed from the queue in the meantime, the order of these events in the queue will get reversed.

[3] For example, state-based exception handling (see Section 6.7.4 in Chapter 6) typically requires immediate handling of exceptional situation, so you don't want other events to overtake the EXCEPTION event.