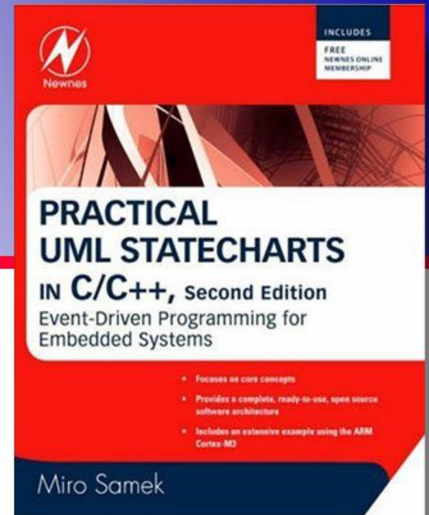
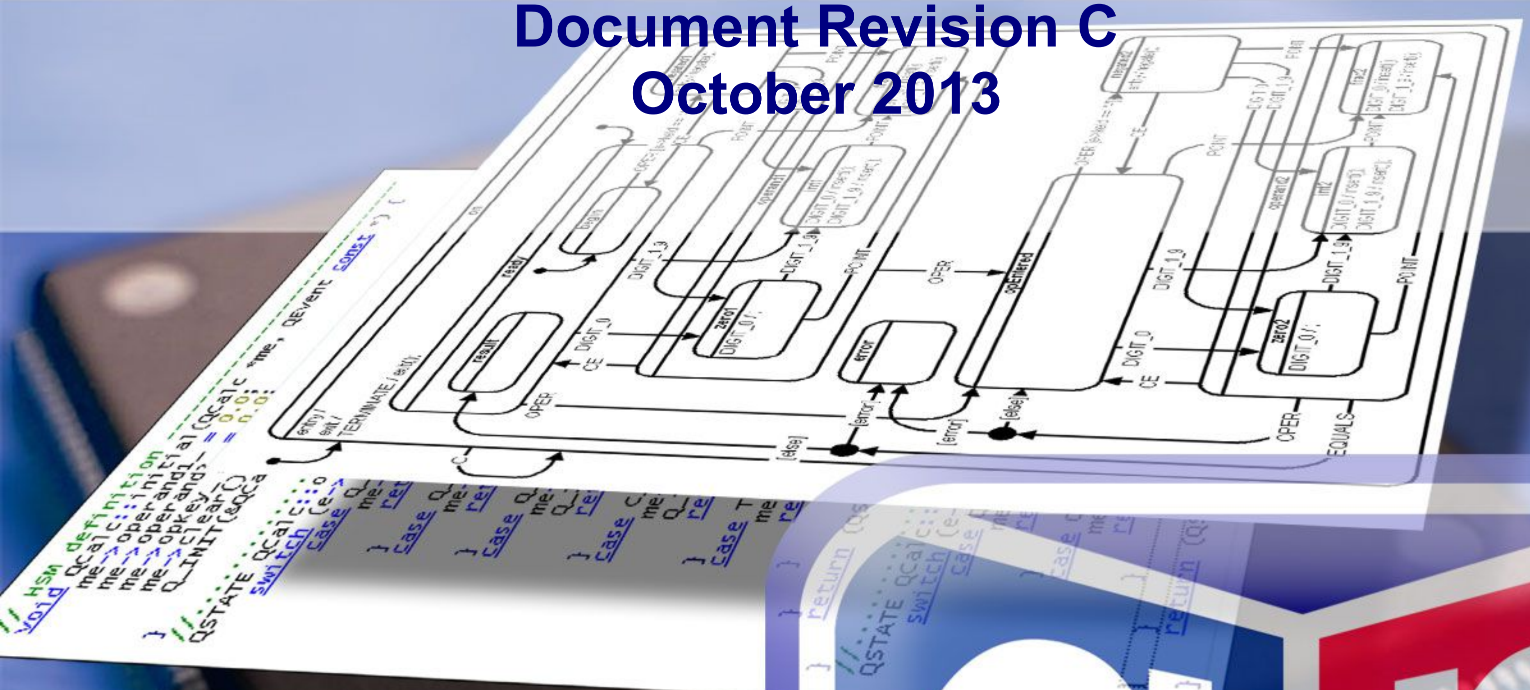
Quantum™ LeaPs
innovating embedded systems



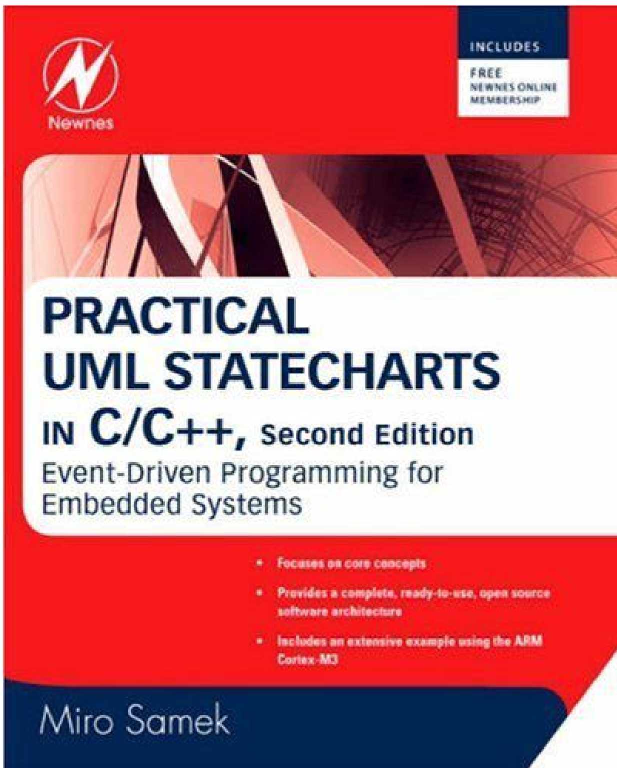
Design Pattern
Deferred Event

Document Revision C
October 2013



Copyright © Quantum Leaps, LLC

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

Copyright © Miro Samek, All Rights Reserved.
Copyright © Quantum Leaps, All Rights Reserved.

Deferred Event

Intent

Simplify state machines by modifying the sequencing of events.

Problem

One of the biggest challenges in designing reactive systems is that such systems must be prepared to handle every event at any time. However, sometimes an event arrives at a particularly inconvenient moment when the system is in the midst of some complex event sequence. In many cases, the nature of the event is such that it can be postponed (within limits) until the system is finished with the current sequence, at which time the event can be recalled and conveniently processed.

Consider, for example, the case of a server application that processes transactions (e.g., from ATM[1] terminals). Once a transaction starts, it typically goes through a sequence of processing, which commences with receiving the data from a remote terminal followed by the authorization of the transaction. Unfortunately, new transaction requests to the server arrive at random times, so it is possible to get a request while the server is still busy processing the previous transaction. One option is to ignore the request, but this might not be acceptable. Another option is to start processing the new transaction immediately, which can complicate things immensely because multiple outstanding transactions would need to be handled simultaneously

Solution

The solution is to *defer* the new request and handle it at a more convenient time, which effectively leads to altering the sequence of events presented to the state machine.

UML statecharts support such a mechanism directly (see Section 2.3.11 in Chapter 2) by allowing every state to specify a list of deferred events. As long as an event is on the combined deferred list of the currently

[1] ATM stands for automated teller machine, a.k.a. “cash machine”.

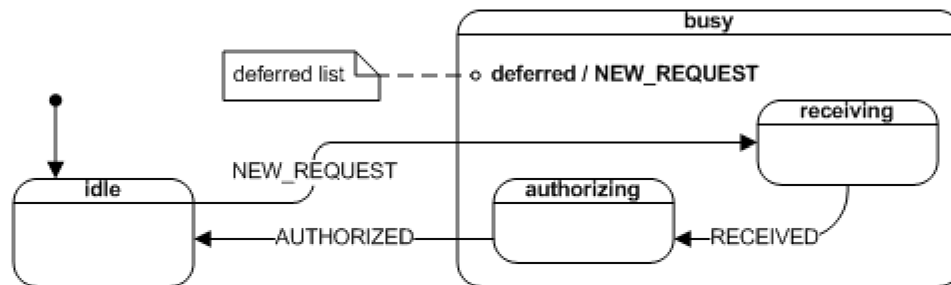
Copyright © Quantum Leaps, LLC. All Rights Reserved.

active state configuration, it is not presented to the state machine but instead is queued for later processing. Upon a state transition, events that are no longer deferred are automatically *recalled* and dispatched to the state machine.

Figure 5.5 illustrates a solution based on this mechanism. The transaction server state machine starts in the "idle" state. The NEW_REQUEST event causes a transition to a substate of the "busy" state. The "busy" state defers the NEW_REQUEST event (note the special "`deferred`" keyword in the internal transition compartment of the "busy" state). Any NEW_REQUEST arriving when the server is still in one of the "busy" substates gets automatically deferred. Upon the transition AUTHORIZED back to the "idle" state, the NEW_REQUEST is automatically recalled. The request is then processed in the "idle" state, just as any other event.

Note: Hierarchical state nesting immensely complicates event deferral because the deferred lists in all superstates of the current state contribute to the mechanism.

Figure 5.5 Event deferral using the built-in UML mechanism.



The lightweight QEP event processor does not support the powerful, but heavyweight, event deferral mechanism of the UML specification. However, you can achieve identical functionality by deferring and recalling events explicitly. In fact, the QF real-time framework supports event deferral by providing `defer()` and `recall()` operations.

Figure 5.6 The Deferred Event state pattern.

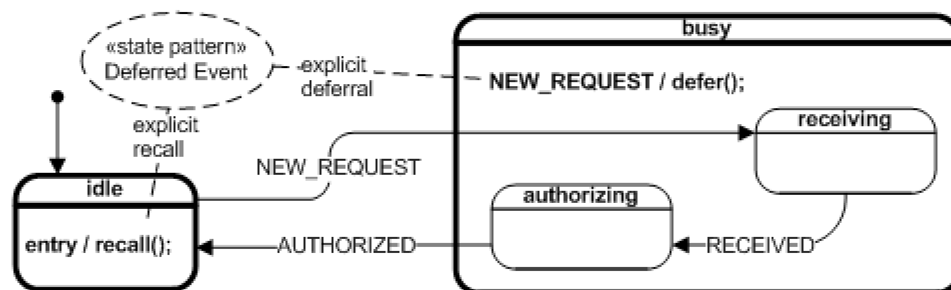


Figure 5.6 shows how to integrate the explicit `defer()` and `recall()` operations into a HSM to achieve the desired effect. The internal transition NEW_REQUEST in the "busy" state traps any NEW_REQEST received in any of the substates. This internal transition calls the `defer()` operation to postpone the event. The "idle" state explicitly recalls any deferred events by calling `recall()` in the entry action. The `recall()` operation posts the first of the deferred events (if available) to self. The state machine then processes the recalled event just as any other event.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

Note: Even though the deferred event is in principle available directly from the `recall()` operation, it is *not* processed in the entry action to "idle". Rather, the `recall()` operation posts the event to self (to the event queue of this state machine). The state machine then handles the NEW_REQUEST event as any other event, that is, in the transition from "idle" to "receiving".

Sample Code

The sample code for the Deferred Event state pattern is found in the directory `qpc\examples\win32\mingw\defer\`. You can execute the application by double-clicking on the file `DEFER.EXE` file in the `dbg\` subdirectory. Figure 5.7 shows the output generated by the `DEFER.EXE` application. The application prints every state entry (to "idle", "receiving", and "authorizing"). Additionally, you get notification of every NEW_REQUEST event and whether it has been deferred or processed directly. You generate new requests by pressing the 'n' key. Please note that request #7 is NOT deferred because the deferred event queue gets full. Please see the explanation section following Listing 5.3 for an overview of options to handle this situation.

Figure 5.7 Annotated output generated by `DEFER.EXE`.

```
Command Prompt - defer\dbg\DEFER.EXE
Deferred Event state pattern
QEP version: 3.4.02
QF  version: 3.4.02
Press n to generate a new request
Press ESC to quit...
idle-ENTRY;
No deferred requests
Processing request #1
receiving-ENTRY;
authorizing-ENTRY;
Request #2 deferred;
Request #3 deferred;
Request #4 deferred;
idle-ENTRY;
Request #2 recalled
Processing request #2
receiving-ENTRY;
Request #5 deferred;
authorizing-ENTRY;
Request #6 deferred;
Request #7 IGNORED;
idle-ENTRY;
Request #3 recalled
Processing request #3
receiving-ENTRY;
authorizing-ENTRY;
idle-ENTRY;
Request #4 recalled
Processing request #4
receiving-ENTRY;
authorizing-ENTRY;
idle-ENTRY;
Request #5 recalled
Processing request #5
receiving-ENTRY;
authorizing-ENTRY;
idle-ENTRY;
Request #6 recalled
Processing request #6
receiving-ENTRY;
authorizing-ENTRY;
idle-ENTRY;
No deferred requests
Processing request #8
receiving-ENTRY;
authorizing-ENTRY;
idle-ENTRY;
No deferred requests
```

Top-most initial transition
All requests processed
New request #1
New request #2
New request #3
New request #4
New request #5
New request #6
New request #7 IGNORED
Recalling and processing deferred requests
All requests processed
New request #8
All requests processed

Listing 5.3 The Deferred Event sample code (file `defer.c`).

Copyright © Quantum Leaps, LLC. All Rights Reserved.

```
(1) #include "qp_port.h"
    #include "bsp.h"

    /*..........................................................................*/
    enum TServerSignals {
        NEW_REQUEST_SIG = Q_USER_SIG,                  /* the new request signal */
        RECEIVED_SIG,                            /* the request has been received */
        AUTHORIZED_SIG,                        /* the request has been authorized */
        TERMINATE_SIG                                /* terminate the application */
    };
    /*..........................................................................*/
(2) typedef struct RequestEvtTag {
        QEvent super;                                     /* derive from QEvent */
        uint8_t ref_num;                        /* reference number of the request */
    } RequestEvt;

    /*..........................................................................*/
(3) typedef struct TServerTag {               /* Transaction Server active object */
(4)     QActive super;                                   /* derive from QActive */

(5)     QEQueue requestQueue;    /* native QF queue for deferred request events */
(6)     QEvent const *requestQSto[3];  /* storage for the deferred queue buffer */

(7)     QTimeEvt receivedEvt;                      /* private time event generator */
(8)     QTimeEvt authorizedEvt;                    /* private time event generator */
    } TServer;

    void TServer_ctor(TServer *me);                            /* the default ctor */
                                              /* hierarchical state machine ... */
    QState TServer_initial    (TServer *me, QEvent const *e);
    QState TServer_idle       (TServer *me, QEvent const *e);
    QState TServer_busy       (TServer *me, QEvent const *e);
    QState TServer_receiving  (TServer *me, QEvent const *e);
    QState TServer_authorizing(TServer *me, QEvent const *e);
    QState TServer_final      (TServer *me, QEvent const *e);

    /*..........................................................................*/
    void TServer_ctor(TServer *me) {                           /* the default ctor */
        QActive_ctor(&me->super, (QStateHandler)&TServer_initial);
(9)     QEQueue_init(&me->requestQueue,
                     me->requestQSto, Q_DIM(me->requestQSto));
        QTimeEvt_ctor(&me->receivedEvt,   RECEIVED_SIG);
        QTimeEvt_ctor(&me->authorizedEvt, AUTHORIZED_SIG);
    }
    /* HSM definition ----------------------------------------------------------*/
    QState TServer_initial(TServer *me, QEvent const *e) {
         (void)e;           /* avoid the compiler warning about unused parameter */
        return Q_TRAN(&TServer_idle);
    }
    /*..........................................................................*/
    QState TServer_final(TServer *me, QEvent const *e) {
         (void)me;          /* avoid the compiler warning about unused parameter */
        switch (e->sig) {
            case Q_ENTRY_SIG: {
                printf("final-ENTRY;\nBye!Bye!\n");
                BSP_exit();                          /* terminate the application */
                return Q_HANDLED();
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

```
            }
        }
        return Q_SUPER(&QHsm_top);
    }
    /*..........................................................................*/
    QState TServer_idle(TServer *me, QEvent const *e) {
        switch (e->sig) {
            case Q_ENTRY_SIG: {
                RequestEvt const *rq;
                printf("idle-ENTRY;\n");

                /* recall the request from the private requestQueue */
(10)            rq = (RequestEvt const *)QActive_recall((QActive *)me,
                                              &me->requestQueue);
                if (rq != (RequestEvt *)0) {          /* recall posted an event? */
(11)                printf("Request #%d recalled\n", (int)rq->refNum);
                }
                else {
(12)                printf("No deferred requests\n");
                }
                return Q_HANDLED();
            }
            case NEW_REQUEST_SIG: {
                printf("Processing request #%d\n",
                       (int)((RequestEvt const *)e)->refNum);
                return Q_TRAN(&TServer_receiving);
            }
            case TERMINATE_SIG: {
                return Q_TRAN(&TServer_final);
            }
        }
        return Q_SUPER(&QHsm_top);
    }
    /*..........................................................................*/
    QState TServer_busy(TServer *me, QEvent const *e) {
        switch (e->sig) {
            case NEW_REQUEST_SIG: {
(13)            if (QEQueue_getNFree(&me->requestQueue) > 0) {     /* can defer? */
                                                         /* defer the request */
(14)                QActive_defer((QActive *)me, &me->requestQueue, e);

                    printf("Request #%d deferred;\n",
                           (int)((RequestEvt const *)e)->ref_num);
                }
                else {
                    /* notify the request sender that the request was ignored.. */
(15)                printf("Request #%d IGNORED;\n",
                           (int)((RequestEvt const *)e)->ref_num);
                }
                return Q_HANDLED();
            }
            case TERMINATE_SIG: {
                return Q_TRAN(&TServer_final);
            }
        }
        return Q_SUPER(&QHsm_top);
    }
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

```
/*..........................................................................*/
QState TServer_receiving(TServer *me, QEvent const *e) {
    switch (e->sig) {
        case Q_ENTRY_SIG: {
            printf("receiving-ENTRY;\n");
                                                /* one-shot timeout in 1 second */
            QTimeEvt_fireIn(&me->receivedEvt, (QActive *)me,
                            BSP_TICKS_PER_SEC);
            return Q_HANDLED();
        }
        case Q_EXIT_SIG: {
            QTimeEvt_disarm(&me->receivedEvt);
            return Q_HANDLED();
        }
        case RECEIVED_SIG: {
            return Q_TRAN(&TServer_authorizing);
        }
    }
    return Q_SUPER(&TServer_busy);
}
/*..........................................................................*/
QState TServer_authorizing(TServer *me, QEvent const *e) {
    switch (e->sig) {
        case Q_ENTRY_SIG: {
            printf("authorizing-ENTRY;\n");
                                               /* one-shot timeout in 2 seconds */
            QTimeEvt_fireIn(&me->authorizedEvt, (QActive *)me,
                            2*BSP_TICKS_PER_SEC);
            return Q_HANDLED();
        }
        case Q_EXIT_SIG: {
            QTimeEvt_disarm(&me->authorizedEvt);
            return Q_HANDLED();
        }
        case AUTHORIZED_SIG: {
            return Q_TRAN(&TServer_idle);
        }
    }
    return Q_SUPER(&TServer_busy);
}
```

(1) The Deferred Event state pattern relies heavily on event queuing, which 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 specific direct support for deferring and recalling events.

(2) The `RequestEvt` event has a parameter `ref_num` (reference number) that uniquely identifies the request.

(3-4) The transaction server (`TServer`) 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 `TServer` also indirectly derives from `QHsm`.

(5) The QF real-time framework provides a "raw" thread-safe event queue class `QEQueue` that is needed to implement event deferral. Here the `TServer` state machine declares the private `requestQueue` event queue to store the deferred request events. The `QEQueue` facility is discussed in Section 7.8.3 of Chapter 7.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

(6) The `QEQueue` requires storage for the ring buffer, which the user must provide, because only the application designer knows how to size this buffer. Please note that event queues in QF store just pointers to `QEvent`, not the whole event objects.

(7-8) The delays of receiving the whole transaction request (RECEIVED) and receiving the authorization notification (AUTHORIZED) are modeled in this example with the time events provided in QF.

(9) The private `requestQueue` event queue is initialized and given its buffer storage.

(10) Per the HSM design, the entry action to the "idle" state recalls the request events. The function `QActive_recall()` returns the pointer to the recalled event, or NULL if no event is currently deferred.

Note: Even though you can "peek" inside the recalled event, you should not process it at this point. By the time `QActive_recall()` function returns, the event is already posted to the active object's event queue using the *last-in-first-out* (LIFO) policy, which guarantees that the recalled event will be the very next to process. (If other events were allowed to overtake the recalled event, the state machine might transition to a state where the recalled event would no longer be convenient.) The state machine will then handle the event like any other request coming at the convenient time. This is the central point of the Deferred Event design pattern.

(11-12) The recalled event is inspected only to notify the user, but not to handle it.

(13) Before the "busy" superstate defers the request, it checks if the private event queue can accept a new deferred event.

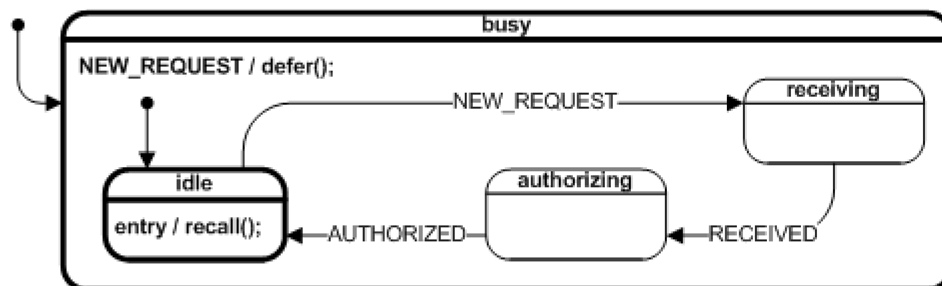
(14) If so, the event is deferred by calling the `QActive_defer()` QF function.

(15) Otherwise, the request is ignored and the user is notified about this fact.

Note: Losing events like this is often unacceptable. In fact, the default policy of QF is to fail an internal assertion whenever an event could be lost. In particular, the `QActve_defer()` function would fire an internal assertion if the event queue could not accept the deferred event. You can try this option by commenting out the `if`-statement in Listing 5.3(13).

Figure 5.8 shows a variation of the Deferred Event state pattern, in which the state machine has the "canonical" structure recommended by the Ultimate Hook pattern. The "busy" state becomes the superstate of all states, including "idle". The "idle" substate overrides the NEW_REQUEST event. All other substates of "busy" rely on the default event handling inside the "busy" superstate, which defers the NEW_REQUEST event. You can very easily try this option by re-parenting the "idle" state. You simply change "`return (QState)&QHsm_top`" to "`return (QState)&TServer_busy`" in the `TServer_idle()` state handler function.

Figure 5.8 A variation of the Deferred Event state pattern.



Copyright © Quantum Leaps, LLC. All Rights Reserved.

Finally, I'd like to point out the true convenience of the `QActive_defer()` and `QActive_recall()` functions. The main difficulty in implementing the event deferral mechanism is actually not the explicit deferring and recalling, but rather the *memory management* for the event objects. Consider, for example, that each request event must occupy some unique memory location, yet you don't know how long the event will be used. Some request events could be recycled just after the run-to-completion (RTC) step of the `TServer` state machine, but some will be deferred and thus will be used much longer. Please recall that for memory efficiency and best performance the deferred event queue, as well as the queues of active objects in QF, store only *pointers* to events, not the whole event objects. How do you organize and manage memory for events?

This is where the QF real-time framework comes in. QF takes care of all the nitty-gritty details of managing event memory and does it very efficiently with "zero-copy" policy and in a *thread-safe* manner. As I will explain in Chapter 7, QF uses efficient event pools combined with a standard reference counting algorithm to know when to recycle events back to the pools. The functions `QActive_defer()` and `QActive_recall()` participate in the reference counting process, so that QF does not recycle deferred events prematurely.

The whole event-management mechanism is remarkably easy to use. You dynamically allocate an event, fill in the event parameters and post it. QF takes care of the rest. In particular, you never explicitly recycle the event. Listing 5.4 shows how the request events are generated in the sample code for the Deferred Event pattern.

Listing 5.4 Generation of new request events with the `Q_NEW()` macro (file `defer.c`).

```
     void BSP_onConsoleInput(uint8_t key) {
         switch (key) {
             case 'n': {                                          /* new request */
                 static uint8_t reqCtr = 0;                /* count the requests */
(1)              RequestEvt *e = Q_NEW(RequestEvt, NEW_REQUEST_SIG);
(2)              e->ref_num = (++reqCtr);              /* set the reference number */
                                          /* post directly to TServer active object */
(3)              QActive_postFIFO((QActive *)&l_tserver, (QEvent *)e);
                 break;
             }
            case 0x1B: {                                             /* ESC key */
(4)              static QEvent const terminateEvt = { TERMINATE_SIG, 0};
(5)              QActive_postFIFO((QActive *)&l_tserver, &terminateEvt);
                 break;
             }
         }
     }
```

(1) When you press the 'n' key, the QF macro `Q_NEW()` creates new `RequestEvt` event and assigns it the signal `NEW_REQUEST_SIG`. The new event is allocated from an "event pool" that the application allocates at startup.

(2) You fill in the event parameters. Here the `ref_num` parameter is set from the incremented static counter.

(3) You post the event to an active object, such as the local `l_tserver` object.

(4) Constant, never changing events, can be allocated statically. Such events should have always the `dynamic_` attribute set to zero (see Listing 4.2 and Section 4.3 in Chapter 4).

(5) You post such static event just like any other event. The QF real-time framework knows not to manage the static events.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

Consequences

Event deferral is a valuable technique for simplifying state models. Instead of constructing an unduly complex state machine to handle every event at any time, you can defer an event when it comes at an inappropriate or awkward time. The event is recalled when the state machine is better able to handle it. The Deferred Event state pattern is a lightweight alternative to the powerful but heavyweight event deferral of UML statecharts. The Deferred Event state pattern has the following consequences.

- It requires explicit deferring and recalling of the deferred events.
- The QF real-time framework provides generic `defer()` and `recall()` operations.
- If a state machine defers more than one event type, it might use the same event queue (`QEQueue`) or different event queues for each event type. The generic QF `defer()` and `recall()` operations support both options.
- Events are deferred in a high-level state, often inside an internal transition in this state.
- Events are recalled in the entry action to the state that can conveniently handle the deferred event type.
- The event should not be processed at the time it is explicitly recalled. Rather, the `recall()` operation posts it using the LIFO policy so that the state machine cannot change state before processing the event.
- Recalling an event involves posting it to self; however, unlike the Reminder pattern, deferred events in are usually external rather than invented.

Known Uses

The Real-Time Object-Oriented Modeling (ROOM) method [Selic+ 94] supports a variation of the Deferred Event pattern presented here. Just like the QF real-time framework, the ROOM virtual machine (infrastructure for executing ROOM models) provides the generic methods `defer()` and `recall()`, which clients need to call explicitly. The ROOM virtual machine also takes care of event queuing. Operations `defer()` and `recall()` in ROOM are specific to the interface component through which an event was received.

Copyright © Quantum Leaps, LLC. All Rights Reserved.