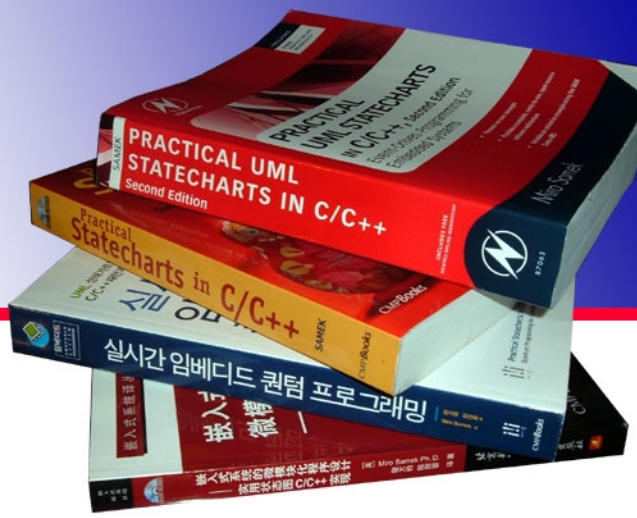
Quantum® LeaPs

innovating embedded systems



Application Note
Dining Philosophers Problem (DPP) Example

Document Revision D
August 2012

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

Table of Contents

1 Introduction

This Application Note describes the classic Dining Philosophers Problem (DPP) as an example application for the QP state machine framework. DPP was posed and solved by Edsger Dijkstra back in 1971 [Dijkstra 71]. The DPP application is relatively simple and can be tested only with a couple of LEDs on your target board. Still, DPP contains six concurrent active objects that exchange events via publish-subscribe and direct event posting mechanisms. The application uses five time events (timers), as well as dynamic and static events. This Application Note describes step-by-step how to design and implemented of DPP with QP.

NOTE: This Application Note assumes the QP/C framework and uses example code in C to explain implementation details. However, the discussion applies equally to QP/C++ version. The differences of the C++ implementation with respect to the C implementation will be discussed only when such differences become non-trivial and important.

2 Requirements

First, your always need to understand what your application is supposed to accomplish. In the case of a simple application, the requirements are conveyed through the problem specification, which for the DPP is as follows.

Five philosophers are gathered around a table with a big plate of spaghetti in the middle (see Figure 1). Between each philosopher is a fork. The spaghetti is so slippery that a philosopher needs two forks to eat it. The life of a philosopher consists of alternate periods of thinking and eating. When a philosopher wants to eat, he tries to acquire forks. If successful in acquiring two forks, he eats for a while, then puts down the forks and continues to think. The key issue is that a finite set of tasks (philosophers) is sharing a finite set of resources (forks), and each resource can be used by only one task at a time. (An alternative oriental version replaces spaghetti with rice and forks with chopsticks, which perhaps explains better why philosophers need two chopsticks to eat.)

As an additional feature, the Dining Philosophers can be paused for an arbitrary period of time. During this paused period, the Philosophers don't get permissions to eat. After the pause period, the Philosophers should resume normal operation.

Figure 1 The Dining Philosophers Problem.

3 Design and Implementation

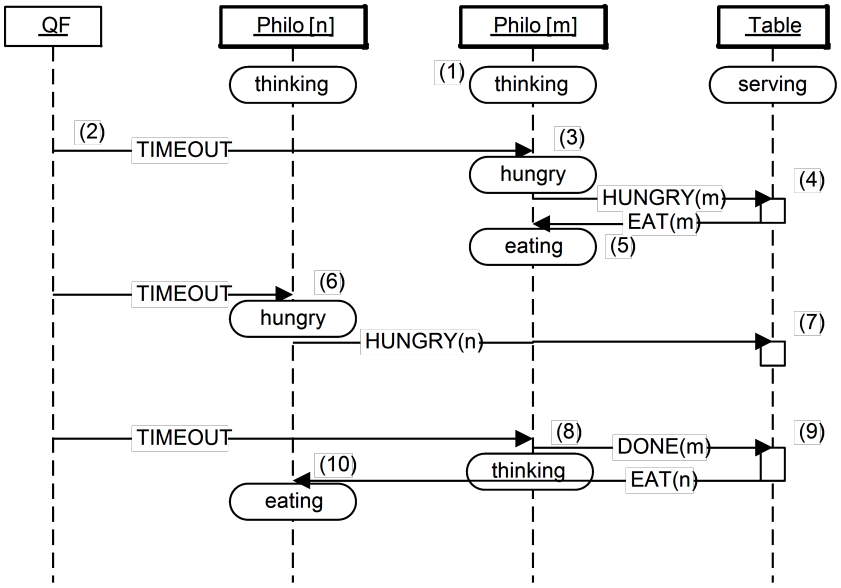
3.1 Step 1: Sequence Diagrams

A good starting point in designing any event-driven system is to draw sequence diagrams for the main scenarios (main use cases) identified from the problem specification. To draw such diagrams, you need to break up your problem into active objects with the main goal of minimizing the coupling among active objects. You seek a partitioning of the problem that avoids resource sharing and requires minimal communication in terms of number and size of exchanged events.

DPP has been specifically conceived to make the philosophers contend for the forks, which are the shared resources in this case. In active object systems, the generic design strategy for handling such shared resources is to encapsulate them inside a dedicated active object and to let that object manage the shared resources for the rest of the system (i.e., instead of sharing the resources directly, the rest of the application shares the dedicated active object). When you apply this strategy to DPP, you will naturally arrive at a dedicated active object to manage the forks. This active object has been named “Table”.

The sequence diagram in Figure 2 shows the most representative event exchanges among any two adjacent Philosophers and the Table active objects.

Figure 2 The sequence diagram of the DPP application.



(1) Each Philosopher active object starts in the “thinking” state. Upon the entry to this state, the Philosopher arms a one-shot time event to terminate the thinking.
(2) The QF framework posts the time event (timer) to Philosopher[m].
(3) Upon receiving the TIMEOUT event, Philosopher[m] transitions to “hungry” state and posts the HUNGRY(m) event to the Table active object. The parameter of the event tells the Table which Philosopher is getting hungry.
(4) The Table active object finds out that the forks for Philosopher[m] are available and grants it the permission to eat by publishing the EAT(m) event.
(5) The permission to eat triggers the transition to “eating” in Philosopher[m]. Also, upon the entry to “eating”, the Philosopher arms its one-shot time event to terminate the eating.

(6) The Philosopher[n] receives the TIMEOUT event, and behaves exactly as Philosopher[m], that is, transitions to “hungry” and posts HUNGRY(n) event to the Table active object.
(7) This time, the Table active object finds out that the forks for Philosopher[n] are not available, and so it does not grant the permission to eat. Philosopher[n] remains in the “hungry” state.
(8) The QF framework delivers the timeout for terminating the eating arrives to Philosopher[m]. Upon the exit from “eating”, Philosopher[m] publishes event DONE(m), to inform the application that it is no longer eating.
(9) The Table active object accounts for free forks and checks whether any direct neighbors of Philosopher[m] are hungry. Table posts event EAT(n) to Philosopher[n].
(10) The permission to eat triggers the transition to “eating” in Philosopher[n].

3.2 Step 2: Signals, Events, and Active Objects

Sequence diagrams, like Figure 2, help you discover events exchanged among active objects. The choice of signals and event parameters is perhaps the most important design decision in any event-driven system. The events affect the other main application components: events and state machines of the active objects.

In QP, signals are typically enumerated constants and events with parameters are structures derived from the `QEvent` base structure. Listing 1 shows signals and events used in the DPP application. The DPP sample code for the DOS version (in C) is located in the `<qp>\qpc\examples\80x86\dos\watcom\l\dpp\` directory, where `<qp>` stands for the installation directory you chose to install the accompanying software.

NOTE: This section describes the platform-independent code of the DPP application. This code is actually *identical* in all DPP versions.

Listing 1 Signals and events used in the DPP application (file `dpp.h`)

```
    #ifndef dpp_h
    #define dpp_h

(1) enum DPPSignals {
(2)     EAT_SIG = Q_USER_SIG,     /* published by Table to let a philosopher eat */
        DONE_SIG,                   /* published by Philosopher when done eating */
        PAUSE_SIG,                  /* published by BSP to pause the application */
        TERMINATE_SIG,          /* published by BSP to terminate the application */
(3)     MAX_PUB_SIG,                                  /* the last published signal */

(4)     HUNGRY_SIG,          /* posted directly from hungry Philosopher to Table */
(5)     MAX_SIG                                             /* the last signal */
    };

    typedef struct TableEvtTag {
(6)     QEvent super;                                    /* derives from QEvent */
        uint8_t philoNum;                                 /* Philosopher number */
    } TableEvt;

    enum { N_PHILO = 5 };                               /* number of Philosophers */

(7) void Philo_ctor(void);           /* ctor that instantiates all Philosophers */
(8) void Table_ctor(void);

(9) extern QActive * const AO_Philo[N_PHILO]; /* "opaque" pointers to Philo AOs */
```

```
(10) extern QActive * const AO_Table;          /* "opaque" pointer  to Table AO */

     #endif                                                          /* dpp_h */
```

(1) For smaller applications, such as the DPP, all signals can be defined in one enumeration (rather than in separate enumerations or, worse, as preprocessor `#define` macros). An enumeration automatically guarantees the uniqueness of signals.

(2) Note that the user signals must start with the offset `Q_USER_SIG` to avoid overlapping the reserved QEP signals.

(3) The globally published signals are grouped at top of the enumeration. The `MAX_PUB_SIG` enumeration automatically keeps track of the maximum published signals in the application.

(4) The Philosophers post the `HUNGRY` event directly to the Table object rather than publicly publish the event (perhaps a Philosopher is “embarrassed” to be hungry, so it does not want other Philosophers to know about it). This demonstrates direct event posting and publish-subscribe mechanism coexisting in a single application.

(5) The `MAX_SIG` enumeration automatically keeps track of the total number of signals used in the application.

(6) Every event with parameters, such as the `TableEvt` derives from the `QEvent` base structure.

The listing shows how to keep the code and data structure of every active object strictly encapsulated within its own C-file. For example, all code and data for the active object `Table` are encapsulated in the file `table.c`, with the external interface consisting of the function `Table_ctor()` and the pointer `AO_Table`.

(7-8) These functions perform an early initialization of the active objects in the system. They play the role of static “constructors”, which in C you need to call explicitly, typically at the beginning of `main()`.

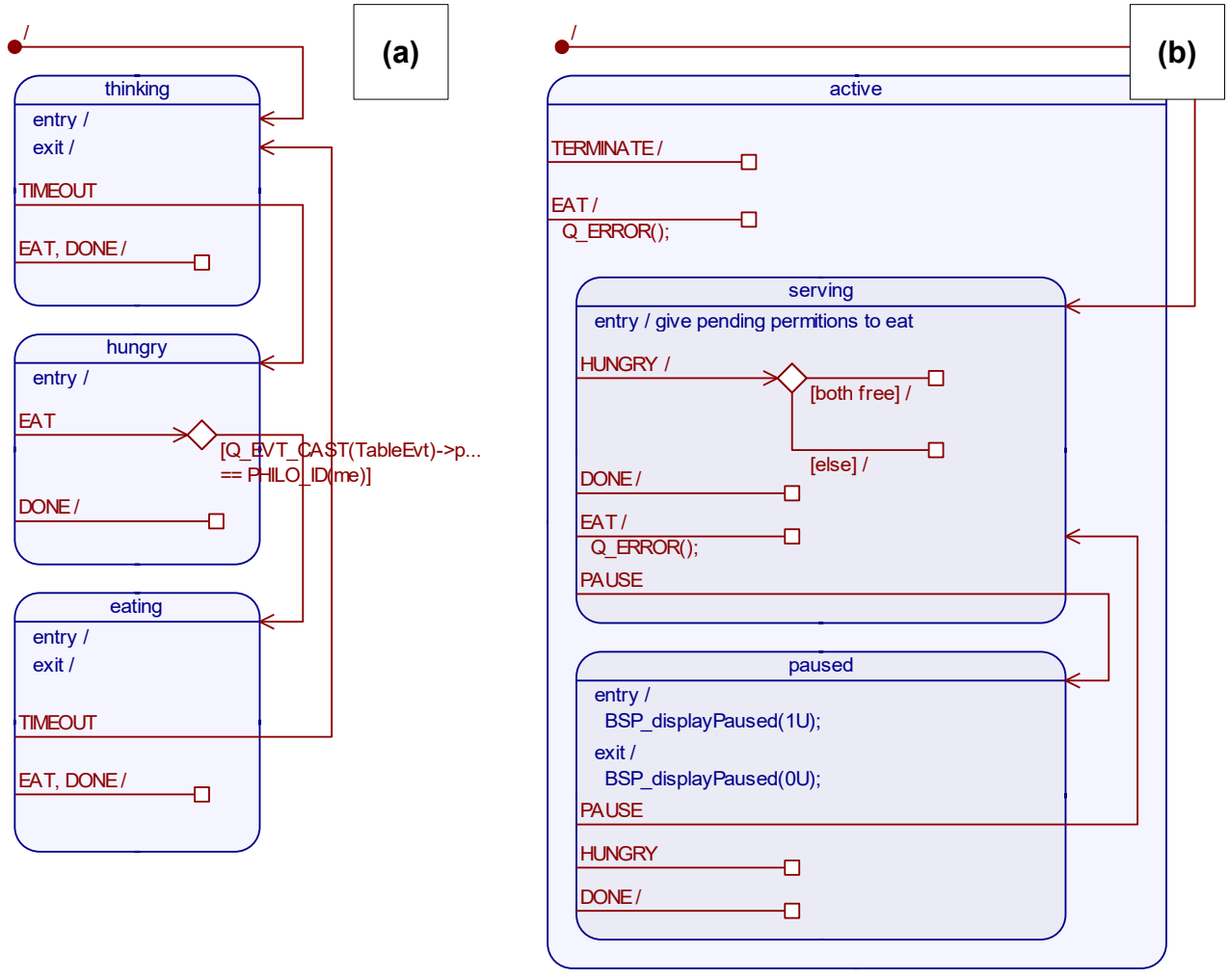
(9-10) These global pointers represent active objects in the application and are used for posting events directly to active objects. Because the pointers can be initialized at compile time, they are declared `const`, sot that they can be placed in ROM. The active object pointers are “opaque”, because they cannot access the whole active object, but only the part inherited from the `QActive` structure.

3.3 Step 3: State Machines

At the application level, you can mostly ignore such aspects of active objects as the separate task contexts, or private event queues, and view them predominantly as state machines. In fact, your main job in developing QP application consists of elaborating the state machines of your active objects.

Figure 3(a) shows the state machines associated with Philosopher active object, which clearly shows the life cycle consisting of states “thinking”, “hungry”, and “eating”. This state machine generates the HUNGRY event on entry to the “hungry” state and the DONE event on exit from the “eating” state because this exactly reflects the semantics of these events. An alternative approach—to generate these events from the corresponding TIMEOUT transitions—would not guarantee the preservation of the semantics in potential future modifications of the state machine. This actually is the general guideline in state machine design.

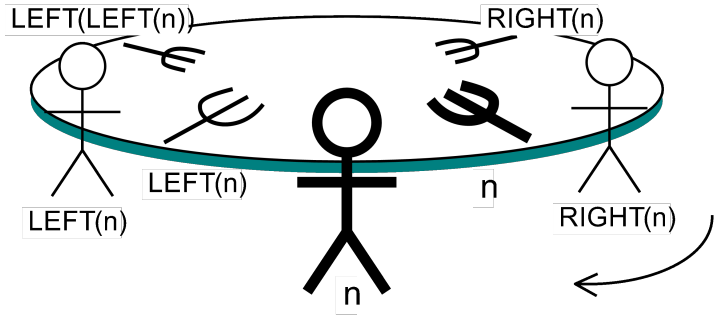
Figure 3 State machines associated with the Philosopher active object (a), and Table active object (b).



GUIDELINE: Favor entry and exit actions over actions on transitions.

Figure 3(b) shows the state machine associated with the Table active object. This state machine is trivial because Table keeps track of the forks and hungry philosophers by means of extended state variables, rather than by its state machine. The state diagram in Figure 3(b) obviously does not convey how the Table active object behaves, as the specification of actions is missing. The actions are omitted from the diagram, however, because including them required cutting and pasting most of the Table code into the diagram, which would make the diagram too cluttered. In this case, the diagram simply does not add much value over the code.

Figure 4 Numbering of philosophers and forks (see the macros LEFT() and RIGHT() in Listing 2).



As mentioned before, each active object is strictly encapsulated inside a dedicated source file (`.C` file). Listing 2 shows the declaration (active object structure) and complete definition (state handler functions) of the Table active object in the file `table.c`. The explanation section immediately following this listing describes the techniques of encapsulating active objects and using QF services. The recipes for coding state machine elements are not repeated here, because they are already described in the “QP Tutorials” available online.

Listing 2 Table active object (file table.c generated by the QM modeling tool).

```
    #include "qp_port.h"
    #include "dpp.h"
    #include "bsp.h"

    Q_DEFINE_THIS_FILE

    /* Active object class -----------------------------------------------------*/
    /* @(/2/1) .................................................................*/
(1) typedef struct TableTag {
    /* protected: */
(2)     QActive super;

    /* private: */
(3)     uint8_t fork[N_PHILO];
(4)     uint8_t isHungry[N_PHILO];
    } Table;

    /* protected: */
    static QState Table_initial(Table * const me, QEvt const * const e);
    static QState Table_active(Table * const me, QEvt const * const e);
    static QState Table_serving(Table * const me, QEvt const * const e);
    static QState Table_paused(Table * const me, QEvt const * const e);
```

```
(5) #define RIGHT(n_) ((uint8_t)(((n_) + (N_PHILO - 1U)) % N_PHILO))
(6) #define LEFT(n_)  ((uint8_t)(((n_) + 1U) % N_PHILO))
    #define FREE      ((uint8_t)0)
    #define USED      ((uint8_t)1)

    /* Local objects -----------------------------------------------------------*/
(7) static Table l_table; /* the single instance of the Table active object */

    /* Global-scope objects ----------------------------------------------------*/
(8) QActive * const AO_Table = &l_table.super; /* "opaque" AO pointer */

    /*..........................................................................*/
    /* @(/2/5) ..................................................................*/
(9) void Table_ctor(void) {
        uint8_t n;
        Table *me = &l_table;

(10)    QActive_ctor(&me->super, Q_STATE_CAST(&Table_initial));

(11)    for (n = 0U; n < N_PHILO; ++n) {
            me->fork[n] = FREE;
            me->isHungry[n] = 0U;
        }
    }
    /* @(/2/1) ..................................................................*/
    /* @(/2/1/2) ................................................................*/
    /* @(/2/1/2/0) */
    static QState Table_initial(Table * const me, QEvt const * const e) {
        uint8_t n;
        (void)e; /* suppress the compiler warning about unused parameter */

(12)    QS_OBJ_DICTIONARY(&l_table);
        QS_FUN_DICTIONARY(&QHsm_top);
        QS_FUN_DICTIONARY(&Table_initial);
        QS_FUN_DICTIONARY(&Table_serving);

        QS_SIG_DICTIONARY(DONE_SIG,      (void *)0); /* global signals */
        QS_SIG_DICTIONARY(EAT_SIG,       (void *)0);
        QS_SIG_DICTIONARY(PAUSE_SIG,     (void *)0);
        QS_SIG_DICTIONARY(TERMINATE_SIG, (void *)0);

        QS_SIG_DICTIONARY(HUNGRY_SIG,    me); /* signal just for Table */

(13)    QActive_subscribe(&me->super, DONE_SIG);
(14)    QActive_subscribe(&me->super, PAUSE_SIG);
(15)    QActive_subscribe(&me->super, TERMINATE_SIG);

        for (n = 0U; n < N_PHILO; ++n) {
            me->fork[n] = FREE;
            me->isHungry[n] = 0U;
(16)        BSP_displayPhilStat(n, "thinking");
        }
        return Q_TRAN(&Table_serving);
    }
    /* @(/2/1/2/1) ..............................................................*/
```

```
    static QState Table_active(Table * const me, QEvt const * const e) {
        QState status;
        switch (e->sig) {
            /* @(/2/1/2/1/0) */
            case TERMINATE_SIG: {
(17)            BSP_terminate(0);
                status = Q_HANDLED();
                break;
            }
            /* @(/2/1/2/1/1) */
            case EAT_SIG: {
(17)            Q_ERROR();
                status = Q_HANDLED();
                break;
            }
            default: {
                status = Q_SUPER(&QHsm_top);
                break;
            }
        }
        return status;
    }
    /* @(/2/1/2/1/2) .....................................................*/
    static QState Table_serving(Table * const me, QEvt const * const e) {
        QState status;
        switch (e->sig) {
            /* @(/2/1/2/1/2) */
            case Q_ENTRY_SIG: {
                uint8_t n;
                for (n = 0U; n < N_PHILO; ++n) { /* give permissions to eat... */
                    if ((me->isHungry[n] != 0U)
                        && (me->fork[LEFT(n)] == FREE)
                        && (me->fork[n] == FREE))
                    {
                        TableEvt *te;

                        me->fork[LEFT(n)] = USED;
                        me->fork[n] = USED;
                        te = Q_NEW(TableEvt, EAT_SIG);
                        te->philoNum = n;
                        QF_PUBLISH(&te->super, me);
                        me->isHungry[n] = 0U;
                        BSP_displayPhilStat(n, "eating  ");
                    }
                }
                status = Q_HANDLED();
                break;
            }
            /* @(/2/1/2/1/2/0) */
            case HUNGRY_SIG: {
                uint8_t n, m;

                n = Q_EVT_CAST(TableEvt)->philoNum;
                /* phil ID must be in range and he must be not hungry */
                Q_ASSERT((n < N_PHILO) && (me->isHungry[n] == 0U));

                BSP_displayPhilStat(n, "hungry  ");
```

```
            m = LEFT(n);
            /* @(/2/1/2/1/2/0/0) */
            if ((me->fork[m] == FREE) && (me->fork[n] == FREE)) {
                TableEvt *pe;
                me->fork[m] = USED;
                me->fork[n] = USED;
                pe = Q_NEW(TableEvt, EAT_SIG);
                pe->philoNum = n;
                QF_PUBLISH(&pe->super, me);
                BSP_displayPhilStat(n, "eating  ");
                status = Q_HANDLED();
            }
            /* @(/2/1/2/1/2/0/1) */
            else {
                me->isHungry[n] = 1U;
                status = Q_HANDLED();
            }
            break;
        }
        /* @(/2/1/2/1/2/1) */
        case DONE_SIG: {
            uint8_t n, m;
            TableEvt *pe;

            n = Q_EVT_CAST(TableEvt)->philoNum;
            /* phil ID must be in range and he must be not hungry */
            Q_ASSERT((n < N_PHILO) && (me->isHungry[n] == 0U));

            BSP_displayPhilStat(n, "thinking");
            m = LEFT(n);
            /* both forks of Phil[n] must be used */
            Q_ASSERT((me->fork[n] == USED) && (me->fork[m] == USED));

            me->fork[m] = FREE;
            me->fork[n] = FREE;
            m = RIGHT(n); /* check the right neighbor */

            if ((me->isHungry[m] != 0U) && (me->fork[m] == FREE)) {
                me->fork[n] = USED;
                me->fork[m] = USED;
                me->isHungry[m] = 0U;
                pe = Q_NEW(TableEvt, EAT_SIG);
                pe->philoNum = m;
                QF_PUBLISH(&pe->super, me);
                BSP_displayPhilStat(m, "eating  ");
            }
            m = LEFT(n); /* check the left neighbor */
            n = LEFT(m); /* left fork of the left neighbor */
            if ((me->isHungry[m] != 0U) && (me->fork[n] == FREE)) {
                me->fork[m] = USED;
                me->fork[n] = USED;
                me->isHungry[m] = 0U;
                pe = Q_NEW(TableEvt, EAT_SIG);
                pe->philoNum = m;
                QF_PUBLISH(&pe->super, me);
                BSP_displayPhilStat(m, "eating  ");
            }
```

```
                status = Q_HANDLED();
                break;
            }
            /* @(/2/1/2/1/2/2) */
            case EAT_SIG: {
                Q_ERROR();
                status = Q_HANDLED();
                break;
            }
            /* @(/2/1/2/1/2/3) */
            case PAUSE_SIG: {
                status = Q_TRAN(&Table_paused);
                break;
            }
            default: {
                status = Q_SUPER(&Table_active);
                break;
            }
        }
        return status;
    }
    /* @(/2/1/2/1/3) ..........................................................*/
    static QState Table_paused(Table * const me, QEvt const * const e) {
        QState status;
        switch (e->sig) {
            . . .
            default: {
                status = Q_SUPER(&Table_active);
                break;
            }
        }
        return status;
    }
```

(1) To achieve true encapsulation, The declaration of the active object structure is placed in the source file (.C file).

(2) Each active object in the application derives from the `QActive` base structure.

(3) The Table active object keeps track of the forks in the array `fork[]`. The forks are numbered as shown in Figure 4.

(4) Similarly, the Table active object needs to remember which philosophers are hungry, in case the forks aren't immediately available. Table keeps track of hungry philosophers in the array `isHungry[]`. Philosophers are numbered as shown in Figure 4.

(5-6) The helper macros `LEFT()` and `RIGHT()` access the left and right philosopher or fork, respectively, as shown in Figure 4.

(7) The Table active object is allocated statically, which makes it inaccessible outside of the .C file.

(8) Externally, the Table active object is known only through the "opaque" pointer `AO_Table`. The pointer is declared '`const`' (with the const after the '*'), which means that the pointer itself cannot change. This ensures that the active object pointer cannot change accidentally and also allows the compiler to allocate the active object pointer in ROM.

(9) The function `Table_ctor()` performs the instantiation of the Table active object. It plays the role of the static "constructor", which in C you need to call explicitly, typically at the beginning of `main()`.

NOTE: In C++, static constructors are invoked automatically before `main()`. This means that in the C++ version of DPP (found in `<qp>\qpcpp\examples\80x86\dos\watcom\l\dpp\`), you provide a regular constructor for the Table class and don't bother with calling it explicitly. However, you must make sure that the startup code for your particular embedded target includes the additional steps required by the C++ initialization.

(10) The constructor must first instantiate the `QActive` superclass.
(11) The constructor can then initialize the internal data members of the active object.
(12) The macros staring with QS_ pertain to the Q-SPY software tracing instrumentation and are active only in the SPY build configuration.
(13-15) The active object subscribes to all interesting to it signals in the top-most initial transition. Please note that Table does not subscribe to the HUNGRY event, because this event is posted directly.

NOTE: New QP users often forget to subscribe to events and then the application appears "dead" when you first run it.

(16) The output to the screen is a BSP (board support package) operation. The different BSPs implement this operation differently, but the code of the Table state machine does not need to change.
(17) Upon receiving the `TERMINATE` event, the Table active object calls `BSP_terminate()` to stop QF and return to the underlying operating system.
(18) The Table state machine extensively uses assertions to monitor correct execution of the DPP application. For example, in line (19) both forks of a philosopher that just finished eating must be used.

The Philosopher active objects bring no essentially new techniques, so the listing of the `philo.c` file is not reproduced here. One interesting aspect of philosophers is that all five philosopher active objects are instances of the same active object class. The philosopher state machine also uses a few assertions to monitor correct execution of the application according to the problem specification.

3.4 Step 4: Initializing and Starting the Application

Most of the system initialization and application startup can be written in a platform-independent way. In other words, you can use essentially the same `main()` function for the DPP application with many QP ports.

Typically, you start all your active objects from `main()`. The signature of the `QActive_start()` function forces you to make several important decisions about each active object upon startup. First, you need to decide the relative priorities of the active objects. Second, you need to decide the size of the event queues you pre-allocate for each active object. The correct size of the queue is actually related to the priority, as described in Chapter 9 of PSiCC2. Third, in some QF ports, you need to give each active object a separate stack, which also needs to be pre-allocated adequately. And finally, you need to decide the order in which you start your active objects.

The order of starting active objects becomes important when you use an OS or RTOS, in which a spawned thread starts to run immediately, possibly preempting the `main()` thread from which you launch your application. This could cause problems, if for example the newly created active object attempts to post an event directly to another active object that has not been yet created. Such situation does not occur in DPP, but if it is an issue for you, you can try to lock the scheduler until all active objects are started. You can then unlock the scheduler in the `QF_onStartup()` callback, which is invoked right before QF takes over control. Some RTOSs (e.g., µC/OS-II) allow you to defer starting multitasking until after you start active objects. Another alternative is to start active objects from within other active objects, but this design increases coupling because the active object that serves as the launch pad must know the priorities, queue sizes, and stack sizes for all active objects to be started.

Listing 3 Initializing and Starting the DPP Application (file main.c).

```
         #include "qp_port.h"
    #include "dpp.h"
    #include "bsp.h"

    /* Local-scope objects -----------------------------------------------------*/
(1) static QEvt const *l_tableQueueSto[N_PHILO];
(2) static QEvt const *l_philoQueueSto[N_PHILO][N_PHILO];
(3) static QSubscrList l_subscrSto[MAX_PUB_SIG];

    /* storage for event pools... */
(4) static QF_MPOOL_EL(TableEvt) l_smlPoolSto[2U*N_PHILO];        /* small pool */

    /*..........................................................................*/
    int_t main(void) {
        uint8_t n;

(5)     Philo_ctor();             /* instantiate all Philosopher active objects */
(6)     Table_ctor();                    /* instantiate the Table active object */

(7)     QF_init();     /* initialize the framework and the underlying RT kernel */
(8)     BSP_init();                                      /* initialize the BSP */

                                                  /* object dictionaries... */
(9)     QS_OBJ_DICTIONARY(l_smlPoolSto);
        QS_OBJ_DICTIONARY(l_tableQueueSto);
        QS_OBJ_DICTIONARY(l_philoQueueSto[0]);
        QS_OBJ_DICTIONARY(l_philoQueueSto[1]);
        QS_OBJ_DICTIONARY(l_philoQueueSto[2]);
        QS_OBJ_DICTIONARY(l_philoQueueSto[3]);
        QS_OBJ_DICTIONARY(l_philoQueueSto[4]);
```

```
(10)     QF_psInit(l_subscrSto, Q_DIM(l_subscrSto));   /* init publish-subscribe */

                                                    /* initialize event pools... */
(11)     QF_poolInit(l_smlPoolSto, sizeof(l_smlPoolSto), sizeof(l_smlPoolSto[0]));

         for (n = 0U; n < N_PHILO; ++n) {         /* start the active objects... */
(12)         QActive_start(AO_Philo[n], (uint8_t)(n + 1U),
                           l_philoQueueSto[n], Q_DIM(l_philoQueueSto[n]),
                           (void *)0, 0U, (QEvt *)0);
         }
(13)     QActive_start(AO_Table, (uint8_t)(N_PHILO + 1U),
                       l_tableQueueSto, Q_DIM(l_tableQueueSto),
                       (void *)0, 0U, (QEvt *)0);

(14)     return (int_t)QF_run();                          /* run the QF application */
     }
```

(1-2) The memory buffers for all event queues are statically allocated.

(3) The memory space for subscriber lists is also statically allocated. The `MAX_PUB_SIG` enumeration comes in handy here.

(4) The macro `QF_MPOOL_EL(TableEvt)` provides correctly aligned memory block of the size at least as big as the `sizeof(TableEvt)` for all events that are served by the "small" event pool.

(5-6) The `main()` function starts with calling all static "constructors" (see Listing 1(7-8)). This step is not necessary in C++.

(7) QF is initialized together with the underlying OS/RTOS.

(8) The target board is initialized.

(9) The macros staring with QS_ pertain to the Q-SPY software tracing instrumentation and are active only in the SPY build configuration.

(10) The publish-subscribe mechanism is initialized. You don't need to call QF_psInit() if your application does not use publish-subscribe.

(11) Up to three event pools can be initialized by calling `QF_poolInit()` up to three times. The subsequent calls must be made in the order of increasing block-sizes of the event pools. You don't need to call `QF_poolInit()` if your application does not use dynamic events.

(12-13) All active objects are started using the "opaque" active object pointers (see Listing 1(9-10)). In this particular example, the active objects are started without private stacks. However, some RTOSs, such as µC/OS-II, require pre-allocating stacks for all active objects.

(14) The control is transferred to QF to run the application. `QF_run()` might never actually return.

3.5 Step 5: Gracefully Terminating the Application

Terminating an application is not really a big concern in embedded systems, because embedded programs almost never have a need to terminate gracefully. The job of a typical embedded system is never finished and most embedded software runs forever or until the power is removed, whichever comes first.

NOTE: You still need to carefully design and test the fail-safe mechanism triggered by a CPU exception or assertion violation in your embedded system. However, such situation represents a catastrophic shutdown, followed perhaps by a reset. The subject of this section is the graceful termination, which is part of the normal application life cycle.

However, in desktop programs, or when embedded applications run on top of a general-purpose operating system, such as Linux, Windows, or DOS, the shutdown of a QP application becomes important. The problem is that in order to terminate gracefully, the application must cleanup all resources allocated by the application during its lifetime. Such a shutdown is always application-specific and cannot be pre-programmed generically at the framework level.

The DPP application uses the following mechanism to shut down. When the user decides to terminate the application, the global TERMINATE event is published. In DPP, only the Table active object subscribes to this event (Listing 2(13)), but in general all active objects that need to cleanup anything before exiting should subscribe to the TERMINATE event. The last subscriber, which is typically the lowest-priority subscriber, calls the `QF_stop()` function. As described in Chapter 8 of PSiCC2, `QF_stop()` is implemented in the QF port. Often, `QF_stop()` causes the `QF_run()` function to return. Right before transferring control to the underlying operating system, QF invokes the `QF_onCleanup()` callback. This callback gives the application the last chance to cleanup globally (e.g., the DOS version restores the original DOS interrupt vectors).

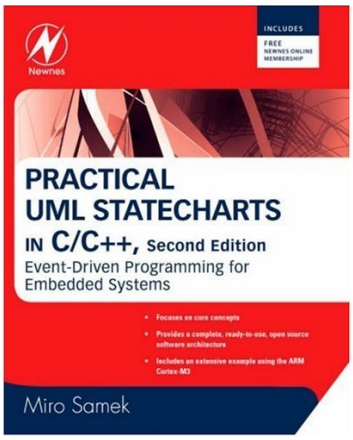
Finally, you can also stop individual active objects and let the rest of the application continue execution. The cleanest way to end an active object's thread is to have it stop itself by calling `QActive_stop(me)`, which should cause a return from the active object's thread routine. Of course to "commit a suicide" voluntarily, the active object must be running, and cannot be waiting for an event. In addition, before disappearing, the active object should release all the resources acquired during its lifetime. Additionally, the active object should unsubscribe from receiving all signals, and somehow should make sure that no more events will be posted to it directly. Unfortunately, all these requirements cannot be pre-programmed generically and always require some work on the application programmer's part.

4 References

Document	Location
[PSiCC2] “Practical UML Statecharts in C/C++, Second Edition”, Miro Samek, Newnes, 2008, ISBN 0750687061	Available from most online book retailers, such as amazon.com. See also: http://www.state-machine.com/psicc2
[QP/C 08] “QP/C Reference Manual”, Quantum Leaps, LLC, 2008	http://www.state-machine.com/qpc/
[QP/C++ 08] “QP/C++ Reference Manual”, Quantum Leaps, LLC, 2008	http://www.state-machine.com/qpcpp/
[QP-nano 08] “QP-nano Reference Manual”, Quantum Leaps, LLC, 2008	http://www.quantum-leaps.com/qpn/

5 Contact Information

Quantum Leaps, LLC
103 Cobble Ridge Drive
Chapel Hill, NC 27516
USA
+1 866 450 LEAP (toll free, USA only)
+1 919 869-2998 (FAX)
e-mail: info@quantum-leaps.com
WEB : www.state-machine.com



"Practical UML Statecharts in C/C++, Second Edition" (**PSiCC2**),

by Miro Samek,
Newnes, 2008,
ISBN 0750687061