Quantum™ LeaPs

innovating embedded systems

Application Note
Dining Philosophers Problem Example (DPP)

Document Revision B

August 2008

Copyright © Quantum Leaps, LLC

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

Table of Contents

1 Introduction

This Application Note describes an example application for the QP™ event-driven platform. QP™ is a family of very lightweight, open source, state machine-based frameworks for developing event-driven applications. QP enables building well-structured embedded applications as a set of concurrently executing hierarchical state machines (UML statecharts) directly in C or C++ **without big tools**. QP is described in great detail in the book "Practical UML Statecharts in C/C++, Second Edition" [PSiCC2] (Newnes, 2008).

As shown in Figure 1, QP consists of a universal UML-compliant event processor (QEP), a portable real-time framework (QF), a tiny run-to-completion kernel (QK), and software tracing instrumentation (QS). Current versions of QP include: QP/C™ and QP/C++™, which require about 4KB of code and a few hundred bytes of RAM, and the ultra-lightweight QP-nano, which requires only 1-2KB of code and just several bytes of RAM.

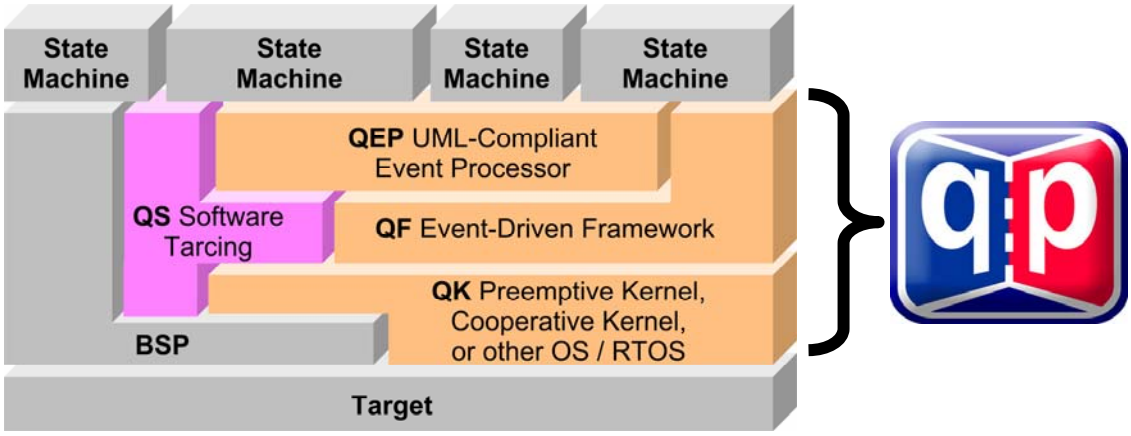


Figure 1 QP Components (in grey) and their relationship with the target hardware, board support package (BSP), and the application.

QP can work with or without a traditional RTOS or OS. In the simplest configuration, QP can completely replace a traditional RTOS. QP includes a simple non-preemptive scheduler and a fully preemptive kernel (QK). QK is smaller and faster than most traditional preemptive kernels or RTOS, yet offers fully deterministic, preemptive execution of embedded applications. QP can manage up to 63 concurrently executing tasks structured as state machines.

QP/C and QP/C++ can also work with a traditional OS/RTOS to take advantage of existing device drivers, communication stacks, and other middleware. QP has been ported to Linux/BSD, Windows, VxWorks, uC/OS-II, and other popular OS/RTOS.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

1.1 Scalability

All components of the QP event-driven platform are designed for scalability, so that your final application image contains only the services that you actually use. QP is designed for deployment as a fine-granularity object library, which you statically link with your applications. This strategy puts the onus on the linker to eliminate any unused code automatically at link-time, instead of burdening the application programmer with configuring the QF code for each application at compile time.

As shown in Figure 2, a minimal QP/C or QP/C++ system requires some 8KB of code space (ROM) and abut 1KB of data space (RAM), to leave enough room for a meaningful application code and data. This code size corresponds to the footprint of a typical, small, bare-bones RTOS application, except that the RTOS approach requires typically more RAM for the stacks.

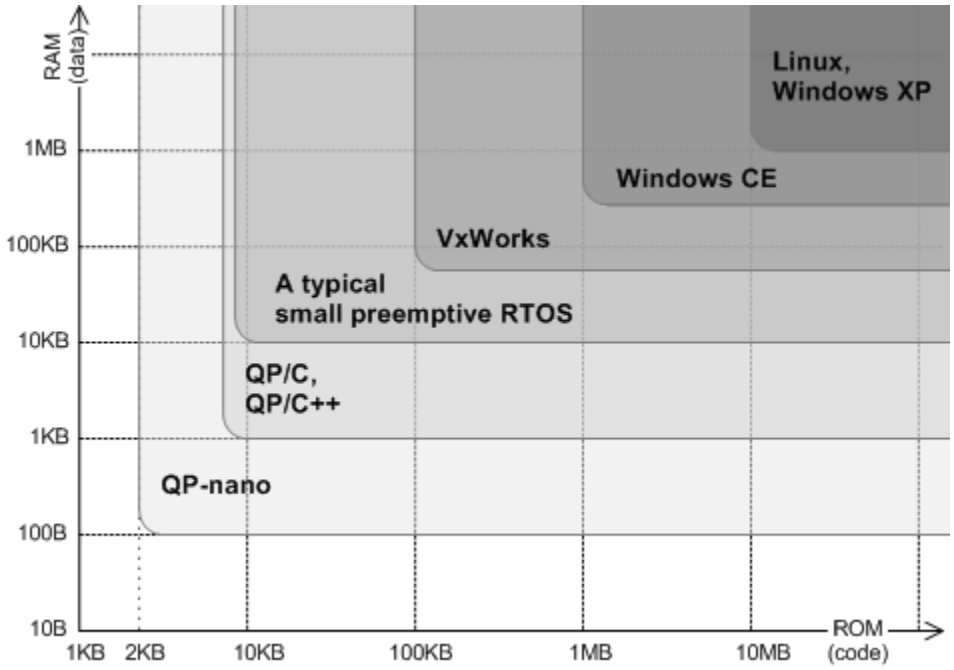


Figure 2 RAM/ROM footprints of QP/C, QP/C++, QP-nano, and other RTOS/OS. The chart shows approximate total system size, as opposed to just the RTOS/OS footprints. Please note logarithmic axes.

NOTE: A typical, standalone QP configuration with QEP, QF, and the "vanilla" scheduler or the QK preemptive kernel, with all major features enabled, requires around 4KB of code. Obviously, you need to budget additional ROM and RAM for your own application code and data. Figure 7.2 shows the application footprint.

1.2 Source Code

The Quantum Leaps website at www.quantum-leaps.com/downloads/ contains the complete source code for all QP components. The source code is very clean and consistent. The code has been written in strict adherence to the coding standard documented at www.quantum-leaps.com/doc/-AN_QL_Coding_Standard.pdf.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

All QP source code is "lint-free". The compliance was checked with PC-lint/FlexLint static analysis tool from Gimpel Software (www.gimpel.com). The QP distribution includes the <qp>\ports\lint\ subdirectory, which contains the batch script make.bat for compiling all the QP components with PC-lint.

The QP source code is also 98% compliant with the Motor Industry Software Reliability Association (MISRA) Guidelines for the Use of the C Language in Vehicle Based Software [MISRA 98]. These standards were created by MISRA to improve the reliability and predictability of C programs in critical automotive systems. Full details of this standard can be obtained directly from the MISRA web site at www.misra.org.uk. The PC-lint configuration used to analyze QP code includes the MISRA rule checker.

Finally and most importantly, simply giving you the source code is not enough. To gain real confidence in event-driven programming, you need to understand how a real-time framework is ultimately implemented and how the different pieces fit together. The [PSiCC2] book, and numerous Application Notes and QDKs, provide this kind of information.

1.3 Portability

All QP source code is written in portable ANSI-C, or in the Embedded C++ subset in case of QP/C++, with all processor-specific, compiler-specific, or operating system-specific code abstracted into a clearly defined platform abstraction layer (PAL).

In the simplest standalone configurations, QP runs on "bare-metal" target CPU completely replacing the traditional RTOS. As shown in Figure 1, the QP event-driven platform includes the simple non-preemptive "vanilla" scheduler as well as the fully preemptive kernel QK. To date, the standalone QF configurations have been ported to over 10 different CPU architectures, ranging from 8-bit (e.g., 8051, PIC, AVR, 68H(S)08), through 16-bit (e.g., MSP430, M16C, x86-real mode), to 32-bit architectures (e.g., traditional ARM, ARM Cortex-M3, ColdFire, Altera Nios II, x86).

The QF framework can also work with a traditional OS/RTOS to take advantage of the existing device drivers, communication stacks, middleware, or any legacy code that requires a conventional "blocking" kernel. To date, QF has been ported to six major operating systems and RTOSs, including Linux (POSIX) and Win32.

1.4 Support for Modern State Machines

As shown in Figure 1, the QF real-time framework is designed to work closely with the QEP hierarchical event processor. The two components complement each other in that QEP provides the UML-compliant state machine implementation, while QF provides the infrastructure of executing such state machines concurrently.

The design of QF leaves a lot of flexibility, however. You can configure the base class for derivation of active objects to be either the hierarchical state machine (HSM), the simpler non-hierarchical state machine (FSM), or even your own base class not defined in the QEP event processor. The latter option allows you to use QF with your own event processor.

1.5 Direct Event Posting and Publish-Subscribe Event Delivery

The QF real-time framework supports direct event posting to specific active objects with first-in-first-out (FIFO) and last-in-first-out (LIFO) policies. QF supports also the more advanced publish-subscribe event deliver mechanism, as described in Chapter 6 of [PSiCC2]. Both mechanisms can coexist in a single application.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

1.6 Zero-Copy Event Memory Management

Perhaps that most valuable feature provided by the QF real-time framework is the efficient "zero-copy" event memory management, as described in Chapter 6 of [PSiCC2]. QF supports event multicasting based on the reference-counting algorithm, automatic garbage collection for events, efficient static events, "zero-copy" event deferral, and up to three event pools with different block sizes for optimal memory utilization.

1.7 Open-Ended Number of Time Events

QP can manage open ended number of time events (timers). QF time events are extensible via structure derivation (inheritance in C++). Each time event can be armed as a one-shot or a periodic timeout generator. Only armed (active) time events consume CPU cycles.

1.8 Native Event Queues

QP provides two versions of native event queues. The first version is optimized for active objects and contains a portability layer to adapt it for either the traditional blocking kernels, the simple cooperative "vanilla" kernel, or the QK preemptive kernel (Chapter 10 in [PSiCC2]). The second native queue version is a simple "thread-safe" queue not capable of blocking and designed for sending events to interrupts as well as storing deferred events. Both native QF event queue types are lightweight, efficient, deterministic, and thread-safe. They are optimized for passing just the pointers to events and are probably smaller and faster than full-blown message queues available in a typical RTOS.

1.9 Native Memory Pool

QF provides a fast, deterministic, and thread-safe memory pool. Internally, QF uses memory pools as event pools for managing dynamic events, but you can also use memory pools for allocating any other objects in you application.

1.10 Built-in "Vanilla" Scheduler

The QF real-time framework contains a portable, cooperative "vanilla" kernel, as described in Section 6.3.7 of Chapter 6. The QF port to the "vanilla" kernel is described in Chapter 7 of [PSiCC2].

1.11 Tight Integration with the QK preemptive Kernel

The QF real-time framework can also work with a deterministic, preemptive, non-blocking QK kernel. As described in Section 6.3.8 in Chapter 6, run-to-completion kernels, like QK, provide preemptive multitasking to event-driven systems at a fraction of the cost in CPU and stack usage compared to traditional blocking kernels/RTOSs. The QK implementation is presented in Chapter 10 of [PSiCC2].

Copyright © Quantum Leaps, LLC. All Rights Reserved.

1.12 Low-Power Architecture

Most modern embedded microcontrollers (MCUs) provide an assortment of low-power sleep modes designed to conserve power by gating the clock to the CPU and various peripherals. The sleep-modes are entered under the software control and are exited upon an external interrupt.

The event-driven paradigm is particularly suitable for taking advantage of these power-savings features, because every event-driven system can easily detect situation when the system has no more events to process, which is called the idle condition (Chapter 6 in [PSiCC2]). In both standalone QF configurations, either with the cooperative "vanilla" kernel, or with the QK preemptive kernel, the QF framework provides callback functions for handling the idle condition. These callbacks are carefully designed to place the MCU into a low-power sleep mode safely and without creating race conditions with active interrupts.

1.13 Assertion-Based Error Handling

The QF real-time framework consistently uses the Design by Contract (DbC) philosophy described in Chapter 6 of [PSiCC2]. QF constantly monitors the application by means of assertions built into the framework. Among others, QF uses assertions to enforce the event delivery guarantee, which immensely simplifies event-driven application design.

1.14 Built-in Software Tracing Instrumentation

The QP code contains the software tracing instrumentation, so it can provide unprecedented visibility into the system. Nominally, the instrumentation is inactive, meaning that it does not add any code size or runtime overhead. But by defining the macro `Q_SPY`, you can activate the instrumentation. Chapter 9 in [PSiCC2] is devoted to software tracing.

NOTE: The QF code is instrumented with QS (Q-spy) macros to generate software trace output from active object execution. However, the instrumentation is disabled by default and will not be shown in the listings discussed in this chapter for better clarity. Please refer to Chapter 9 for more information about the QS software tracing implementation.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

2 The Dining Philosopher Problem Example

Many QP™ Development Kits (QDKs) use the classic Dining Philosophers Problem (DPP) as an example application. 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.

2.1 Step 1: 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 3). 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.)

Figure 3 The Dining Philosophers Problem.

2.2 Step 2: 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.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

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 4 shows the most representative event exchanges among any two adjacent Philosophers and the Table active objects.

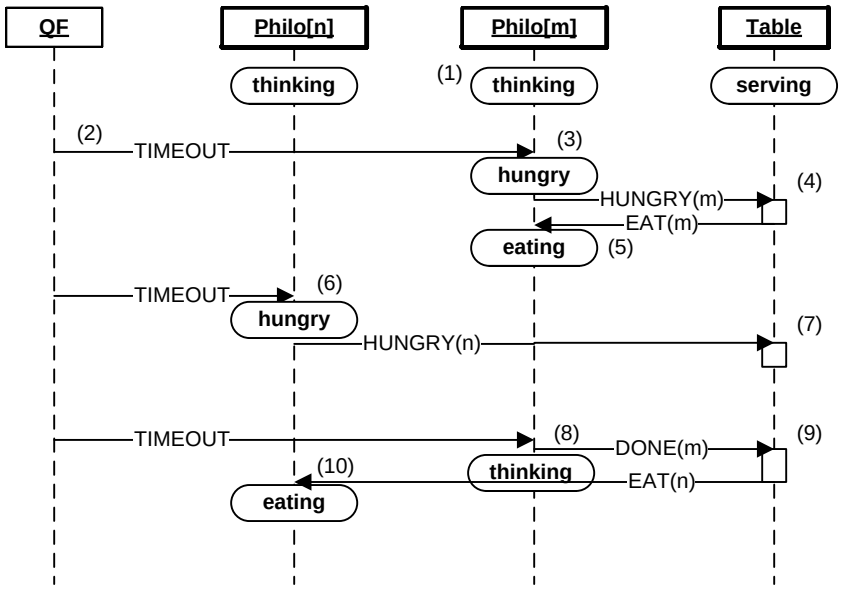


Figure 4 The sequence diagram of the DPP application.

(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.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

(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].

2.3 Step 3: Signals, Events, and Active Objects

Sequence diagrams, like Figure 4, 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\tcpp101\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.

```
    #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 */
        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 */
```

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

Copyright © Quantum Leaps, LLC. All Rights Reserved.

(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.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

2.4 Step 4: 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.

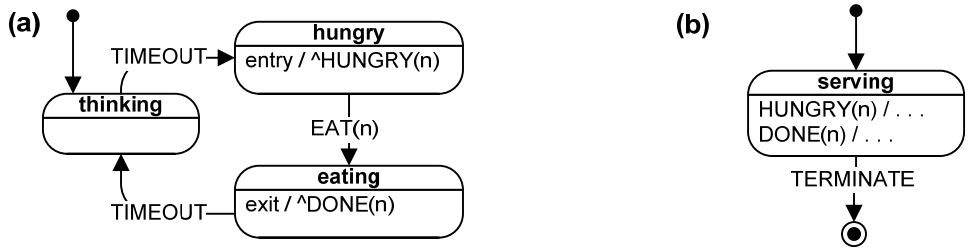


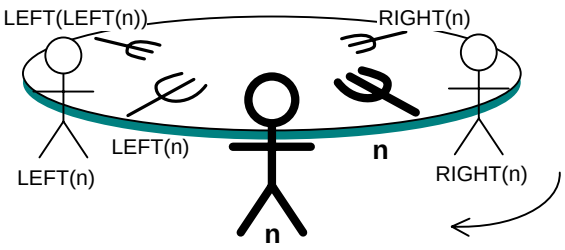
Figure 5 State machines associated with the Philosopher active object (a), and Table active object (b).

Figure 5(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.

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

Figure 5(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 5(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 6 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 han-

Copyright © Quantum Leaps, LLC. All Rights Reserved.

dler 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.

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

    Q_DEFINE_THIS_FILE

    /* Active object class -----------------------------------------------------*/
(1) typedef struct TableTag {
(2)     QActive super;                                  /* derives from QActive */
(3)     uint8_t fork[N_PHILO];                           /* states of the forks */
(4)     uint8_t isHungry[N_PHILO];             /* remembers hungry philosophers */
    } Table;

    static QState Table_initial(Table *me, QEvent const *e);     /* pseudostate */
    static QState Table_serving(Table *me, QEvent const *e);   /* state handler */

(5) #define RIGHT(n_) ((uint8_t)(((n_) + (N_PHILO - 1)) % N_PHILO))
(6) #define LEFT(n_)  ((uint8_t)(((n_) + 1) % N_PHILO))
    enum ForkState { FREE, USED };

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

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

    /*..........................................................................*/
(9) void Table_ctor(void) {
        uint8_t n;
        Table *me = &l_table;
(10)    QActive_ctor(&me->super, (QStateHandler)&Table_initial);

(11)    for (n = 0; n < N_PHILO; ++n) {
            me->fork[n] = FREE;
            me->isHungry[n] = 0;
        }
    }
    /*..........................................................................*/
    QState Table_initial(Table *me, QEvent const *e) {
        (void)e;           /* avoid the compiler warning about unused parameter */

(12)    QActive_subscribe((QActive *)me, DONE_SIG);
(13)    QActive_subscribe((QActive *)me, TERMINATE_SIG);
(14)    /* signal HUNGRY_SIG is posted directly */

        return Q_TRAN(&Table_serving);
    }
    /*..........................................................................*/
    QState Table_serving(Table *me, QEvent const *e) {
        uint8_t n, m;
        TableEvt *pe;

        switch (e->sig) {
            case HUNGRY_SIG: {
(15)            BSP_busyDelay();
                n = ((TableEvt const *)e)->philoNum;
                        /* phil ID must be in range and he must be not hungry */
(16)            Q_ASSERT((n < N_PHILO) && (!me->isHungry[n]));
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

```
(17)            BSP_displyPhilStat(n, "hungry  ");
                m = LEFT(n);
                if ((me->fork[m] == FREE) && (me->fork[n] == FREE)) {
                    me->fork[m] = me->fork[n] = USED;
                    pe = Q_NEW(TableEvt, EAT_SIG);
                    pe->philoNum = n;
                    QF_publish((QEvent *)pe);
                    BSP_displyPhilStat(n, "eating  ");
                }
                else {
                    me->isHungry[n] = 1;
                }
                return Q_HANDLED();
            }
            case DONE_SIG: {
                BSP_busyDelay();
                n = ((TableEvt const *)e)->philoNum;
                        /* phil ID must be in range and he must be not hungry */
(18)            Q_ASSERT((n < N_PHILO) && (!me->isHungry[n]));

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

                me->fork[m] = me->fork[n] = FREE;
                m = RIGHT(n);                       /* check the right neighbor */
                if (me->isHungry[m] && (me->fork[m] == FREE)) {
                    me->fork[n] = me->fork[m] = USED;
                    me->isHungry[m] = 0;
                    pe = Q_NEW(TableEvt, EAT_SIG);
                    pe->philoNum = m;
(20)                QF_publish((QEvent *)pe);
                    BSP_displyPhilStat(m, "eating  ");
                }
                m = LEFT(n);                         /* check the left neighbor */
                n = LEFT(m);                  /* left fork of the left neighbor */
                if (me->isHungry[m] && (me->fork[n] == FREE)) {
                    me->fork[m] = me->fork[n] = USED;
                    me->isHungry[m] = 0;
                    pe = Q_NEW(TableEvt, EAT_SIG);
                    pe->philoNum = m;
(21)                QF_publish((QEvent *)pe);
                    BSP_displyPhilStat(m, "eating  ");
                }
                return Q_HANDLED();
            }
            case TERMINATE_SIG: {
(22)            QF_stop();
                return Q_HANDLED();
            }
        }
        return Q_SUPER(&QHsm_top);
    }
```

Listing 2 Table active object (file table.c). Boldface indicates the QF services

(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.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

(3) The Table active object keeps track of the forks in the array `fork[]`. The forks are numbered as shown in Figure 6.

(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 6.

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

(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\tcpp101\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.

(12-13) The active object subscribes to all interesting to it signals in the top-most initial transition.

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

(14) Please note that Table does not subscribe to the HUNGRY event, because this event is posted directly.

(15) The state machine is sprinkled with calls to the function `BSP_busyDelay()` in order to artificially prolong the RTC processing. The function `BSP_busyDelay()` busy-waits in a counted loop, whereas you can adjust the number of iterations of this loop from the command-line or through a debugger. This technique lets me increase the probability of various preemptions and thus helps me use the DPP application for stress-testing various QP ports.

(16,18,19) 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.

(17) 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.

(20,21) It is possible that the Table active object publishes two events in a single RTC step.

(22) Upon receiving the TERMINATE event, the Table active object calls `QF_stop()` to stop QF and return to the underlying operating system.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

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.

2.5 Step 5: 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.

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

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

(4) static union SmallEvent {
(5)     void *min_size;
        TableEvt te;
(6)     /* other event types to go into this pool */
(7) } l_smlPoolSto[2*N_PHILO];              /* storage for the small event pool */

    /*..........................................................................*/
    int main(int argc, char *argv[]) {
        uint8_t n;

(8)     Philo_ctor();             /* instantiate all Philosopher active objects */
(9)     Table_ctor();                    /* instantiate the Table active object */

(10)    BSP_init(argc, argv);           /* initialize the Board Support Package */

(11)    QF_init();     /* initialize the framework and the underlying RT kernel */
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

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

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

         for (n = 0; n < N_PHILO; ++n) {          /* start the active objects... */
(14)         QActive_start(AO_Philo[n], (uint8_t)(n + 1),
                           l_philoQueueSto[n], Q_DIM(l_philoQueueSto[n]),
                           (void *)0, 0,                    /* no private stack */
                           (QEvent *)0);
         }
(15)     QActive_start(AO_Table, (uint8_t)(N_PHILO + 1),
                       l_tableQueueSto, Q_DIM(l_tableQueueSto),
                       (void *)0, 0,                        /* no private stack */
                       (QEvent *)0);
(16)     QF_run();                                      /* run the QF application */

         return 0;
     }
```

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

(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 union `SmallEvent` contains all events that are served by the "small" event pool.

(5) The union contains a pointer-size member to make sure that the union size will be at least that big.

(6) You add all events that you want to be served from this event pool.

(7) The memory buffer for the "small" event pool is statically allocated.

(8-9) The `main()` function starts with calling all static "constructors" (see Listing 1(7-8)). This step is not necessary in C++.

(10) The target board is initialized.

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

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

(13) 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.

(14-15) 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.

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

Copyright © Quantum Leaps, LLC. All Rights Reserved.

2.6 Step 6: 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 lifecycle.

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 (Listing 9.2(22)). 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 preprogrammed generically and always require some work on the application programmer's part.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

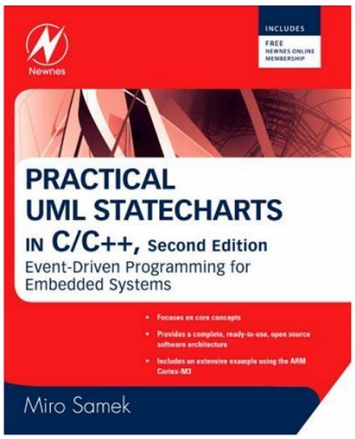
3 References

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.htm
[QP/C 08] "QP/C Reference Manual", Quantum Leaps, LLC, 2008	http://www.state-machine.com/doxygen/qpc/
[QP/C++ 08] "QP/C++ Reference Manual", Quantum Leaps, LLC, 2008	http://www.state-machine.com/doxygen/qpcpp/
[QP-nano 08] "QP-nano Reference Manual", Quantum Leaps, LLC, 2008	http://www.state-machine.com/doxygen/qpn/
[QL AN-Directory 07] "Application Note: QP Directory Structure", Quantum Leaps, LLC, 2007	http://www.state-machine.com/doc/-AN_QP_Directory_Structure.pdf

Copyright © Quantum Leaps, LLC. All Rights Reserved.

4 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 : http://www.quantum-leaps.com



"Practical UML State-charts in C/C++, Second Edition" (**PSiCC2**), by Miro Samek, Newnes, 2008, ISBN 0750687061

Copyright © Quantum Leaps, LLC. All Rights Reserved.