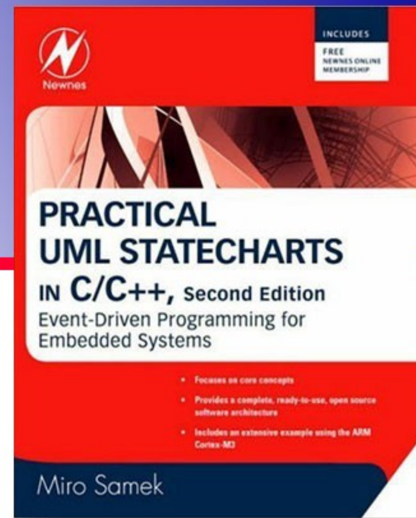
quantum™ LeaPs
innovating embedded systems



Application Note
QP™ and POSIX

Document Revision G
February 2016

Copyright © Quantum Leaps, LLC
www.state-machine.com

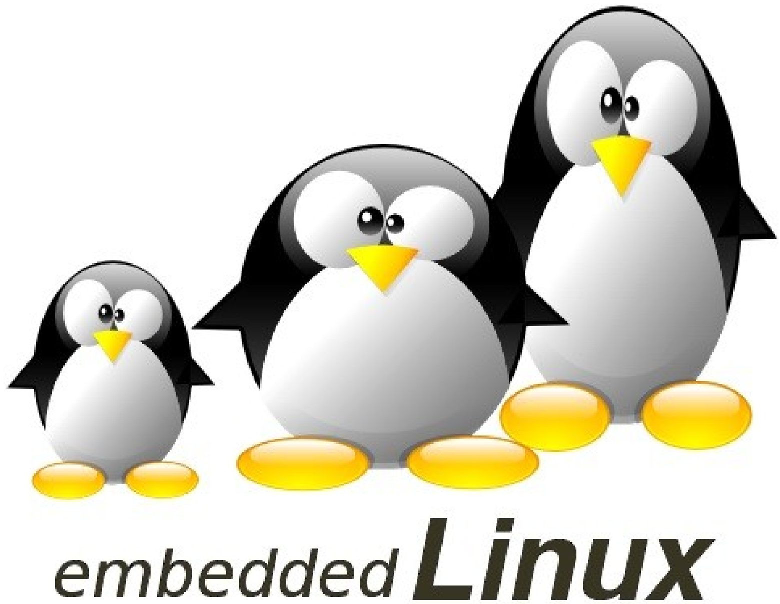
Table of Contents

Legal Disclaimers

Information in this document is believed to be accurate and reliable. However, Quantum Leaps does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information.

Quantum Leaps reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

All designated trademarks are the property of their respective owners.



1 Introduction

This Application Note describes how to use the QP™/C and QP™/C++ real-time embedded frameworks (RTEFs) version **5.x.x** or higher with the **POSIX** standard-compliant operating system, such as Linux, **embedded Linux**, BSD, Mac OS X, QNX, VxWorks, or INTEGRITY (with POSIX subsystem) as the QP port to Linux strictly adheres to the **POSIX 1003.1cn1995** standard.

To focus the discussion, the Application Note uses a console-based version of the Dining Philosopher Problem (DPP) test application running on standard 80x86-based PC running Linux (see the Application Note [QL AN-DPP 08] "Application Note: Dining Philosophers Application"). However, the QP port is applicable to any other hardware platform running Linux, embedded Linux, or any other POSIX-compatible OS, such as ARM, PowerPC, MIPS, etc. The same port also applies to applications with GUI as well as deeply embedded applications without a console.

NOTE: This Application Note pertains both to C and C++ versions of the QP™ real-time embedded frameworks. Most of the code listings in this document refer to the C version. Occasionally the C code is followed by the equivalent C++ implementation to show the C++ differences whenever such differences become important.

1.1 About QP™

QP™ (Quantum Platform) is a family of lightweight Real-Time Embedded Frameworks (RTEFs) for building reactive embedded software as systems of asynchronous event-driven active objects (actors). The QP™ family consists of QP/C, QP/C++, and QP-nano frameworks, which are all strictly quality controlled, thoroughly documented, and available in full source code.

The behavior of active objects is specified in QP™ by means of Hierarchical State Machines (UML Statecharts). The QP™ frameworks support manual coding of UML state machines in C or C++ as well as Model-Based Design (MBD) and automatic code generation by means of the free QM™ Model-Based Design tool.

All QP™ RTEFs can run on bare-metal single-chip microcontrollers, completely replacing a traditional RTOS. The frameworks contain a selection of built-in real-time kernels (RTOS kernels), such as the cooperative QV kernel, the preemptive non-blocking QK kernel, and the unique preemptive, dual-mode

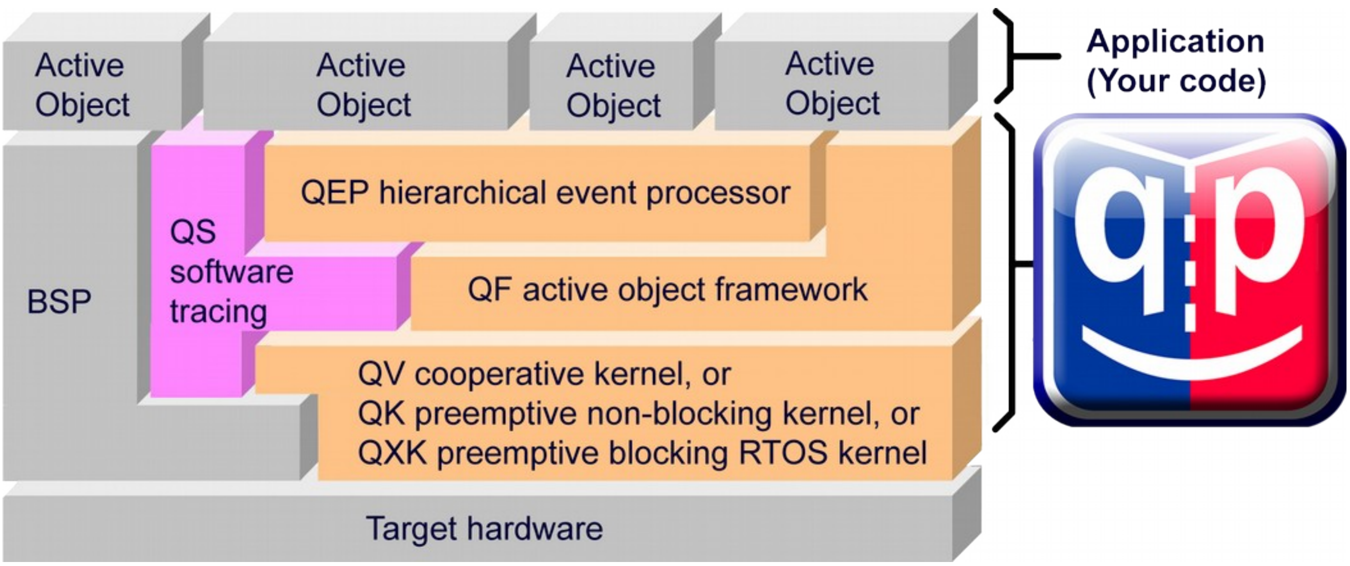
Copyright © Quantum Leaps, LLC. All Rights Reserved.

(blocking/non-blocking) QXK kernel. Native QP ports and ready-to-use examples are provided for ARM Cortex-M (M0/M0+/M3/M4F/M7) as well as other CPUs.

QP/C and QP/C++ RTEFs can also work with many traditional RTOSes and desktop OSes (such as Linux/POSIX and Windows).

With over 50,000 downloads a year, the QP™ RTEF family is the most popular such solution on the embedded software market. It provides a modern, reusable architecture of embedded applications, which combines the active-object model of concurrency with hierarchical state machines. This architecture is generally safer, more responsive and easier to understand than shared-state concurrency of a conventional Real-Time Operating System (RTOS). It also provides higher level of abstraction and the right abstractions to effectively apply modeling and code generation to deeply embedded systems, such as ARM Cortex-M-based microcontrollers.

Figure 1: QP components and their relationship with the target hardware, board support package (BSP), and the application



1.2 About QM™

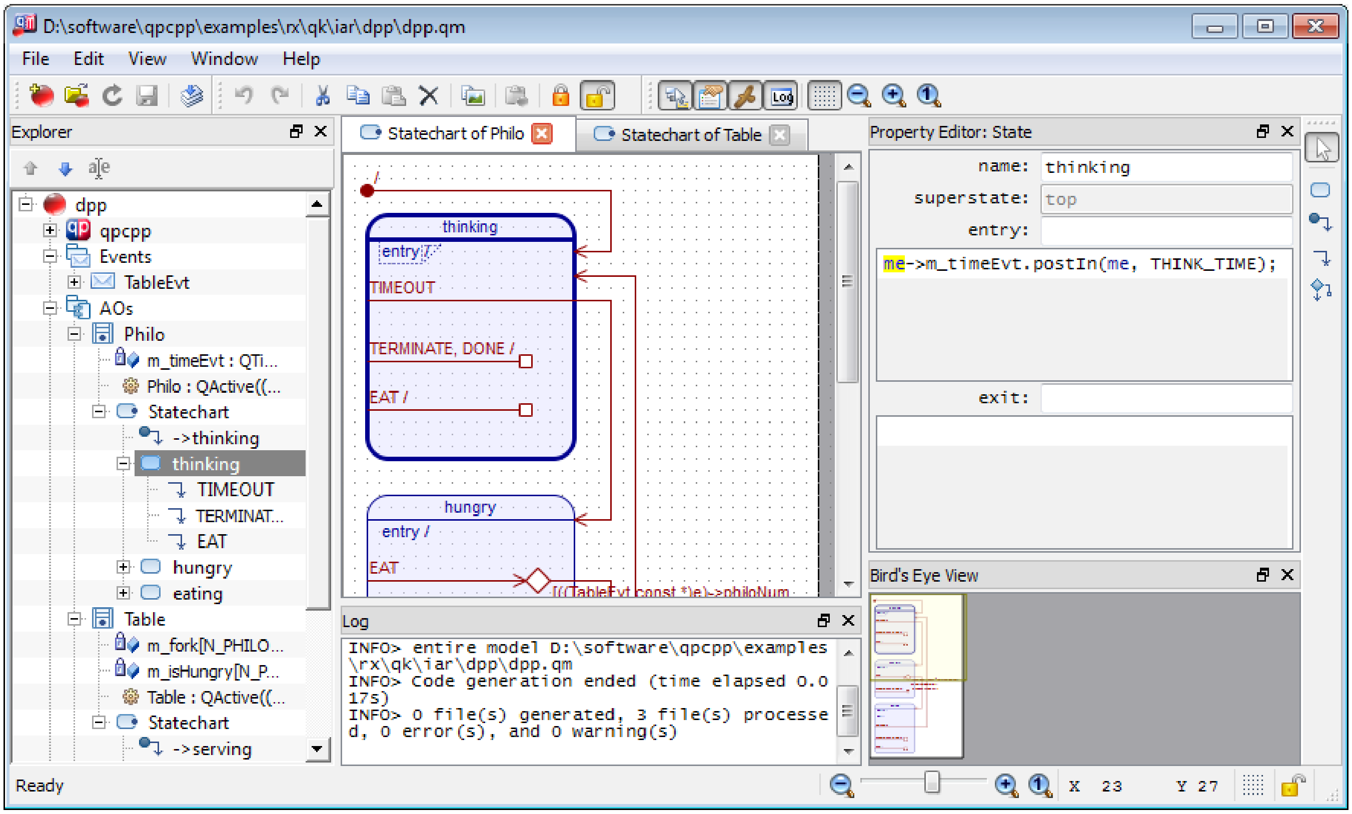
Although originally designed for manual coding, the QP RTEFs make also excellent targets for **automatic code generation**, which is provided by a graphical modeling tool called **QM™** (QP™ Modeler).

QM™ is a **free**, cross-platform, graphical UML modeling tool for designing and implementing real-time embedded applications based on the QP™ state machine frameworks. QM™ is available for Windows, Linux, and Mac OS X.

QM™ provides intuitive diagramming environment for creating good looking hierarchical state machine diagrams and hierarchical outline of your entire application. QM™ eliminates coding errors by automatic generation of compact C or C++ code that is 100% traceable from your design. Please visit state-machine.com/qm for more information about QM™.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

Figure 2: The example model opened in the QM™ modeling tool



1.3 About the QP™ Port to POSIX

In this port, a QP application runs as a single POSIX process, with each QP active object executing in a separate lightweight POSIX thread (Pthread). The port uses a Pthread mutex to implement the QP critical section and the Pthread condition variables to provide the blocking mechanism for event queues of active objects.

The general assumption underlying the QP port to POSIX is that the application is going to be real-time or perhaps "soft real-time". This means that the port is trying to use as much as possible the real-time features available in the standard POSIX API. Since some of these features require the "superuser" privileges, the actual real-time behavior of the application will depend on the privilege level at which it is launched.

In POSIX, the scheduler policy closest to real-time is the `SCHED_FIFO` policy, available only with the "superuser" privileges. At initialization, QP attempts to set this policy. However, setting the `SCHED_FIFO` policy might fail, most probably due to insufficient privileges. In that case the, QP application will attempt to use the default scheduling policy `SCHED_OTHER`.

The QP port to POSIX uses one dedicated Pthread to periodically call the `QF_tick()` function to handle the armed time events. At startup, QP attempts to set the priority of this "ticker" thread to the maximum, so that the system clock tick occurs in the timely manner. However, again, the attempt to set the priority of the "ticker thread" can fail (due to insufficient privileges), in which case the thread priority is left unchanged.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

1.4 Licensing QP™ and QP port to POSIX

The **Generally Available (GA)** distribution of QP™ available for download from the www.state-machine.com/downloads website is offered with the following two licensing options:

- The GNU General Public License version 2 (GPL) as published by the Free Software Foundation and appearing in the file `GPL.TXT` included in the packaging of every Quantum Leaps software distribution. The GPL *open source* license allows you to use the software at no charge under the condition that if you redistribute the original software or applications derived from it, the complete source code for your application must be also available under the conditions of the GPL (GPL Section 2[b]).
- One of several Quantum Leaps commercial licenses, which are designed for customers who wish to retain the proprietary status of their code and therefore cannot use the GNU General Public License. The customers who license Quantum Leaps software under the commercial licenses do not use the software under the GPL and therefore are not subject to any of its terms.

For more information, please visit the licensing section of our website at: www.state-machine.com/licensing

1.5 Licensing QM™

The QM™ graphical modeling tool available for download from the www.state-machine.com/downloads website is **free** to use, but is not open source. During the installation you will need to accept a basic End-User License Agreement (EULA), which legally protects Quantum Leaps from any warranty claims, prohibits removing any copyright notices from QM, selling it, and creating similar competitive products.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

2 Directories and Files

The code for the QP port to POSIX is part of the standard QP distribution, which also contains example applications. The standard distribution is available in a platform-independent ZIP file that you can unzip into an arbitrary root directory. The QP Root Directory you choose for the installation will be henceforth referred to as `<qp>`.

Listing 1: Directories and files pertaining to the QP port to POSIX included in the standard QP distribution.

```
<qp>/                       - QP-root directory for Quantum Platform (QP)
  |
  +-include/                - QP public include files
  | +-qassert.h             - QP assertions public include file
  | +-qep.h                 - QEP platform-independent public include
  | +-qf.h                  - QF platform-independent public include
  | +-qk.h                  - QK platform-independent public include
  | +-qs.h                  - QS platform-independent public include
  | +-. . .
  +-ports/                  - QP ports
  | +-posix/                - POSIX port
  | | +-qep_port.h          - QEP platform-dependent public include
  | | +-qf_port.h           - QF platform-dependent public include
  | | +-qf_port.c           - QF port to POSIX
  | | +-qs_port.h           - QS platform-dependent public include
  | | +-qs_port.c           - QS port to POSIX
  |
  +-examples/               - subdirectory containing the QP example files
  | +-workstation/          - Examples for workstations
  | | +- . . .
  | | +-dpp/                - Dining Philosopher Problem application
  | | | +-build/            - directory containing the Debug build
  | | | +-build_rel/        - directory containing the Release build
  | | | +-build_spy/        - directory containing the Spy build
  | | | |
  | | | +-Makefile          - Makefile for building the application
  | | | +-bsp.c             - Board Support Package (console application)
  | | | +-bsp.h             - BSP header file
  | | | +-main.c            - the main function
  | | | +-philo.c           - the Philosopher active object
  | | | +-table.c           - the Table active object
  | | | +-dpp.h             - the DPP header file
  | | | +-dpp.qm            - the DPP model file
  | +-qutest/               - Examples for QUTest unit testing
  | | +- . . .
  | | +-dpp/                - Dining Philosopher Problem application
  | | | +-src/              - Source (Code Under Test)
  | | | | +-. . .
  | | | | +-philo.c         - the Philosopher active object
  | | | | +-table.c         - the Table active object
  | | | +-test_dpp/         - Test fixture and test build
  | | | | +-Makefile        - Makefile for building and running the tests
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

2.1 Building the QP Applications

As shown in Listing 1, the DPP application example for POSIX is located in the directory `<qp>/examples/workstation/dpp/`. This directory contains the `Makefile` to build the example. The provided `Makefile` supports three build configurations: debug, release, and spy (`make`, `make CONF=rel`, `make CONF=spy`, respectively.

NOTE: The QP applications can be built in the following three **build configurations**:

Debug - this configuration is built with full debugging information and minimal optimization. When the QP framework finds no events to process, the framework busy-idles until there are new events to process.

Release - this configuration is built with no debugging information and high optimization. Single-stepping and debugging is effectively impossible due to the lack of debugging information and optimized code, but the debugger can be used to download and start the executable. When the QP framework finds no events to process, the framework puts the CPU to sleep until there are new events to process.

Spy - like the debug variant, this variant is built with full debugging information and minimal optimization. Additionally, it is build with the QP's Q-SPY trace functionality built in. The on-board serial port and the Q-Spy host application are used for sending and viewing trace data. Like the Debug configuration, the QP framework busy-idles until there are new events to process.

Table 1: Make targets for the Debug, Release, and Spy software configurations

Software Version	Build command	Clean command
Debug (default)	`make`	`make clean`
Release	`make CONF=rel`	`make CONF=rel clean`
Spy	`make CONF=spy`	`make CONF=spy clean`

2.2 Executing the Example

The DPP example is a console application, which you can launch from the command prompt. The following listing shows the console output from the test run (debug build). You "pause" the philosophers by pressing the '**p**' key, you terminate the application by pressing the **Esc key** on the keyboard.

Listing 2: Console output from the run of the DPP application

```
$dbg/dpp
Dining Philosopher Problem example
QP 5.3.0
Press 'p' to pause/un-pause
Press ESC to quit...
Philosopher  0 is thinking
Philosopher  1 is thinking
Philosopher  2 is thinking
Philosopher  3 is thinking
Philosopher  4 is thinking
Philosopher  4 is hungry
Philosopher  4 is eating
Paused is ON
Philosopher  0 is hungry
Philosopher  2 is hungry
Philosopher  1 is hungry
Philosopher  3 is hungry
Philosopher  4 is thinking
Philosopher  4 is hungry
Paused is OFF
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

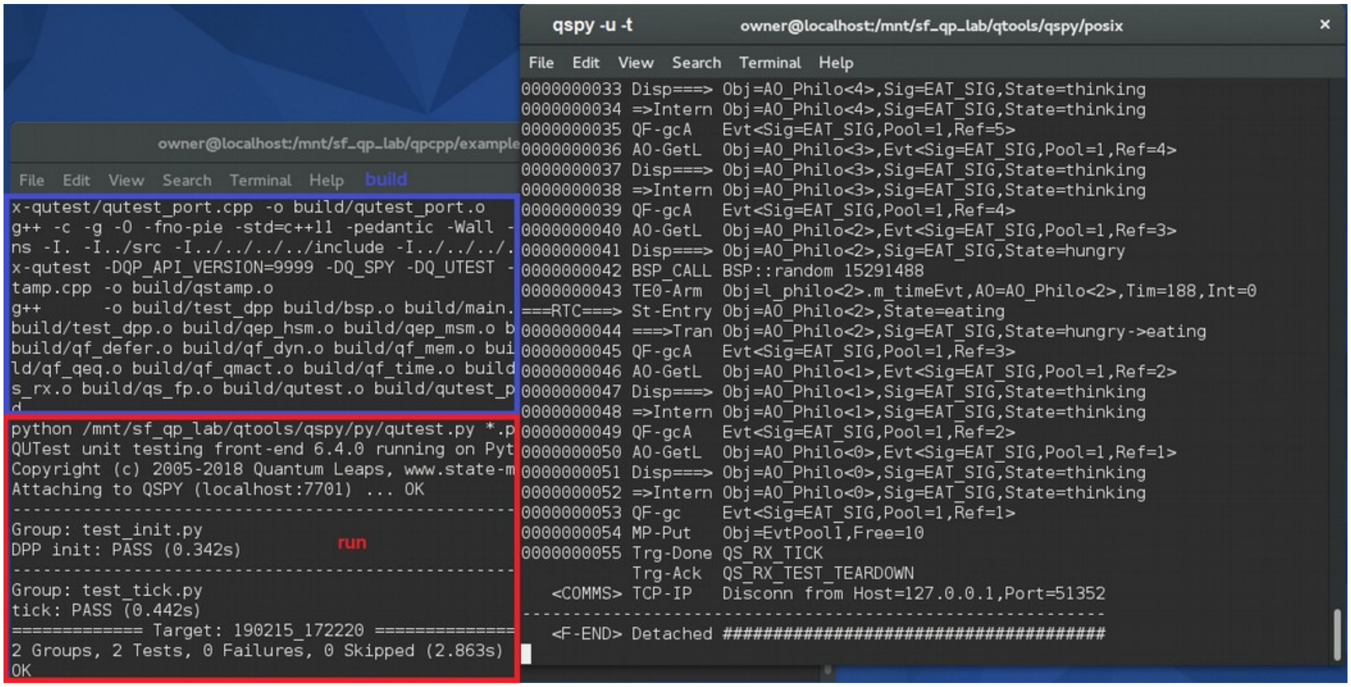
```
Philosopher  0 is eating
Philosopher  2 is eating
Philosopher  0 is thinking
Philosopher  4 is eating
Philosopher  2 is thinking
Philosopher  1 is eating
```

2.3 QP/Spy Software Tracing and QUTest Unit Testing

The QP port to POSIX provides the support for the QS (QP/Spy) software tracing as well as the unit testing with QUTest. In the POSIX port, the software tracing data is sent from the Target via TCP/IP.

2.3.1 Example QUTest Session with QP/Spy output

Figure 3: The example test run of the DPP application on Linux (left terminal: test build and run; right terminal: qspy output)



Copyright © Quantum Leaps, LLC. All Rights Reserved.

3 The QP Port to POSIX

3.1 The qep_port.h Header File

Listing 3 shows the `qep_port.h` header file for POSIX. The GNU gcc compiler supports the C99 standard, so the standard `<stdint.h>` header file is available.

Listing 3: The `qep_port.h` header file for POSIX.

```
#ifndef qep_port_h
#define qep_port_h

#include <stdint.h> /* C99-standard exact-width integers */
#include "qep.h"    /* QEP platform-independent public interface */

#endif /* qep_port_h */
```

3.2 The qs_port.h Header File and 64-bit Considerations

Listing 4 shows the `qs_port.h` header file for POSIX. The sizes of pointers are determined based on the machine word size. The 64-bit OS versions are detected by checking the `__LP64__` and `_LP64` preprocessor macros.

NOTE: The `qs_port.h` header file is the only part of the QP framework dependent on the pointer representation. So, with this dependency taken care for, the provided QP port code does not need to change in any way to run in 64-bit POSIX implementations.

Listing 4: The `qs_port.h` header file for POSIX.

```
#ifndef qs_port_h
#define qs_port_h

#define QS_TIME_SIZE            4

#if defined(__LP64__) || defined(_LP64)  /* 64-bit architecture? */
    #define QS_OBJ_PTR_SIZE     8
    #define QS_FUN_PTR_SIZE     8
#else                                    /* 32-bit architecture */
    #define QS_OBJ_PTR_SIZE     4
    #define QS_FUN_PTR_SIZE     4
#endif

#include "qf_port.h" /* use QS with QF */
#include "qs.h"      /* QS platform-independent public interface */

#endif /* qs_port_h  */
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

3.3 The qf_port.h Header File

Listing 5 shows the `qf_port.h` header file for POSIX. You typically should not need to change this file as you move to a different POSIX-compliant OS.

Listing 5: The `qf_port.h` header file for POSIX. Boldface indicates elements of the Pthread API

```
     #ifndef qf_port_h
     #define qf_port_h

     /* POSIX event queue and thread types */
 (1) #define QF_EQUEUE_TYPE                QEQueue
 (2) #define QF_OS_OBJECT_TYPE             pthread_cond_t
 (3) #define QF_THREAD_TYPE                uint8_t

     /* The maximum number of active objects in the application */
 (4) #define QF_MAX_ACTIVE                 64

     /* various QF object sizes configuration for this port */
 (6) #define QF_EVENT_SIZ_SIZE             4
 (7) #define QF_EQUEUE_CTR_SIZE            4
 (8) #define QF_MPOOL_SIZ_SIZE             4
 (9) #define QF_MPOOL_CTR_SIZE             4
(10) #define QF_TIMEEVT_CTR_SIZE           4

     /* QF critical section entry/exit for POSIX, see NOTE01 */
(11) /* QF_CRIT_STAT_TYPE not defined */
(12) #define QF_CRIT_ENTRY(dummy)          pthread_mutex_lock(&QF_pThreadMutex_)
(13) #define QF_CRIT_EXIT(dummy)           pthread_mutex_unlock(&QF_pThreadMutex_)

(14) #include <pthread.h>  /* POSIX-thread API */
(15) #include "qep_port.h" /* QEP port */
(16) #include "qequeue.h"  /* POSIX needs event-queue */
(17) #include "qmpool.h"   /* POSIX needs memory-pool */
(18) #include "qf.h"       /* QF platform-independent public interface */

(19) void QF_setTickRate(uint32_t ticksPerSec); /* set clock tick rate */
(20) void QF_onClockTick(void); /* clock tick callback (provided in the app) */

(21) extern pthread_mutex_t QF_pThreadMutex_; /* mutex for QF critical section */


     /*****************************************************************************
     * interface used only inside QF, but not in applications
     */
     #ifdef qf_pkg_h

         /* OS-object implementation for POSIX */
(22)     #define QACTIVE_EQUEUE_WAIT_(me_) \
             while ((me_)->eQueue.frontEvt == (QEvent *)0) \
                 pthread_cond_wait(&(me_)->osObject, &QF_pThreadMutex_)

(23)     #define QACTIVE_EQUEUE_SIGNAL_(me_) \
             pthread_cond_signal(&(me_)->osObject)
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

```
(24)      #define QACTIVE_EQUEUE_ONEMPTY_(me_) ((void)0)

          /* native QF event pool operations */
(25)      #define QF_EPOOL_TYPE_               QMPool
(26)      #define QF_EPOOL_INIT_(p_, poolSto_, poolSize_, evtSize_) \
              QMPool_init(&(p_), poolSto_, poolSize_, evtSize_)
(27)      #define QF_EPOOL_EVENT_SIZE_(p_)     ((p_).blockSize)
(28)      #define QF_EPOOL_GET_(p_, e_)        ((e_) = (QEvent *)QMPool_get(&(p_)))

      #endif /* qf_pkg_h */
```

(1) The POSIX port employs the QF native `QEQueue` as the event queue for active objects.

(2) The Pthread condition variable is used for blocking the QF native event queue. Please note that each active object has its own private condition variable.

(3) Each active object also holds a handle to its Pthread.

(4) The POSIX port is configured to use the maximum allowed number of active objects.

(6-10) POSIX requires at least a 32-bit CPU, so all sizes of internal QF objects are set to 4 bytes.

(11) The `QF_CRIT_STAT_TYPE` macro is not defined. This means that the critical section status is not preserved across the QF critical section.

(12) The QF critical section is implemented with a single global Pthread mutex `QF_pThreadMutex_`. The mutex is locked upon the entry to a critical section.

(13) The global mutex `QF_pThreadMutex_` is unlocked upon the exit from a critical section.

NOTE: The global mutex `QF_pThreadMutex_` is configured as a normal "fast" Pthread mutex, which cannot handle nested locks. Consequently, the QF port to POSIX does not support nesting of critical sections. This QF port is designed to never nest critical sections internally, but you should be careful not to call QF services from critical sections at the application level.

---

(14) The system header file `<pthread.h>` contains the Pthread API.

(15) This QF port uses the QEP event processor.

(16) This QF port uses the native QF event queue `QEQueue`.

(17) This QF port uses the native QF memory pool `QMPool`.

(18) The platform-independent `qf.h` header file must be always included.

(19) The helper function `QF_setTickRate(` allows you to change the system clock tick rate from the default value to the multiple of the default value.

(20) The callback function `QF_onClockTick()` is called from `QF_run()` to process the system clock tick. This function must call `QF_TICKX()`, but can also perform other useful tasks.

(21) The platform-independent `qf.h` header file must be always included.

The following three macros `QACTIVE_EQUEUE_WAIT_()`, `QACTIVE_EQUEUE_SIGNAL_()`, and `QACTIVE_EQUEUE_ONEMPTY_()` customize the native QF event queue to use the Pthread condition variable for blocking and signaling the active object's thread. (See Section 7.8.3 in [PSiCC2] for the context in which QF calls these macros.)

Copyright © Quantum Leaps, LLC. All Rights Reserved.

(22) As long as the queue is empty, the private condition variable osObject blocks the calling thread. Please note that the macro ACTIVE_EQUEUE_WAIT_() is called from critical section, that is, with the global mutex QF_pThreadMutex_ locked.

The behavior of the pthread_cond_wait() function requires explanation. Here is the description from the POSIX-thread standard:

"The function pthread_cond_wait() *atomically releases the associated mutex and causes the calling thread to block on the condition variable. Atomically here means "atomically with respect to access by another thread to the mutex and then the condition variable". That is, if another thread is able to acquire the mutex after the about-to-block thread has released it, then a subsequent call to* pthread_cond_signal() *or* pthread_cond_broadcast() *in that thread behaves as if it were issued after the about-to-block thread has blocked"*.

The bottom line is, that the global mutex QF_pThreadMutex_ remains unlocked only as long as pthread_cond_wait() blocks. The mutex gets locked again as soon as the function unblocks. This means that the macro ACTIVE_EQUEUE_WAIT_() returns within critical section, which is exactly what the intervening code in QActive_get_() expects.

The while-loop around the pthread_cond_wait() call is necessary because of the following comment in the POSIX-thread documentation:

"Since the return from pthread_cond_wait() *does not imply anything about the value of the predicate, the predicate should be re-evaluated upon such return"*.

(23) The macro QACTIVE_EQUEUE_SIGNAL_() is called when an event is inserted into an empty event queue (so the queue becomes not-empty). Please note that this macro is called form a critical section.

(24) The macro QACTIVE_EQUEUE_ONEMPTY_() is called when the queue is becoming empty. This macro is defined to nothing in this port.

(25-28) The POSIX port uses QMPool as the QF event pool. The platform abstraction layer (PAL) macros are set to access the QMPool operations (see Section 7.9 in [PSiCC2]).

3.4 The qf_port.c Source File

The qf_port.c source file shown in Listing 6 provides the "glue-code" between QF and the POSIX API. The general assumption I make here is that QF is going to be used in real-time applications (perhaps "soft real-time"). This means that I'm trying to use as much as possible the real-time features available in the standard POSIX API. Since some of these features require the "superuser" privileges, the actual real-time behavior of the application will depend on the privilege level at which it is launched. As always with a general-purpose OS used for real-time applications, your actual mileage may vary.

Listing 6: The qf_port.c header file for POSIX. Boldface indicates elements of the Pthread API.

```
#include "qf_pkg.h"
#include "qassert.h"

#include <sys/mman.h>   /* for mlockall() */
#include <sys/select.h> /* for select() */

Q_DEFINE_THIS_MODULE("qf_port")
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

```
     /* Global objects ----------------------------------------------------------*/
(1)  pthread_mutex_t QF_pThreadMutex_ = PTHREAD_MUTEX_INITIALIZER;

     /* Local objects -----------------------------------------------------------*/
     static long int l_tickUsec = 10000UL; /* clock tick in usec (for tv_usec) */
     static uint8_t l_running;

     /*..........................................................................*/
     int16_t QF_init(void) {
         /* lock memory so we're never swapped out to disk */
(2)      /*mlockall(MCL_CURRENT | MCL_FUTURE); uncomment when supported */
     }
     /*..........................................................................*/
(3)  void QF_run(void) {
         struct sched_param sparam;
         struct timeval timeout = { 0 }; /* timeout for select() */

(4)      QF_onStartup(); /* invoke startup callback */

         /* try to maximize the priority of the ticker thread, see NOTE01 */
(5)      sparam.sched_priority = sched_get_priority_max(SCHED_FIFO);
(6)      if (pthread_setschedparam(pthread_self(), SCHED_FIFO, &sparam) == 0) {
                          /* success, this application has sufficient privileges */
         }
         else {
             /* setting priority failed, probably due to insufficient privieges */
         }
         l_running = (uint8_t)1;
(7)      while (l_running) {
(8)          QF_onClockTick(); /* clock tick callback (must call QF_TICK()) */

(9)          timeout.tv_usec = l_tickUsec; /* set the desired tick interval */
(10)         select(0, 0, 0, 0, &timeout); /* sleep for the full tick , NOTE05 */
         }
(11)     QF_onCleanup(); /* invoke cleanup callback */
(12)     pthread_mutex_destroy(&QF_pThreadMutex_);
(13)     return (uint16_t)0;
     }
     /*..........................................................................*/
     void QF_stop(void) {
(14)     l_running = (uint8_t)0;                     /* stop the loop in QF_run() */
     }
     /*..........................................................................*/
(15) static void *thread_routine(void *arg) { /* the expected POSIX signature */
(16)     ((QActive *)arg)->running = (uint8_t)1; /* allow the thread loop to run */
(17)     while (((QActive *)arg)->running) { /* QActive_stop() stopps the loop */
(18)         QEvent const *e = QActive_get_((QActive *)arg);/*wait for the event */
(19)         QF_ACTIVE_DISPATCH_(&((QActive *)arg)->super, e);/* dispatch to  SM */
(20)         QF_gc(e); /* check if the event is garbage, and collect it if so */
         }
(21)     QF_remove_((QActive *)arg);/* remove this object from any subscriptions */
         return (void *)0; /* return success */
(22) }
     /*..........................................................................*/
     void QActive_start(QActive *me, uint8_t prio,
                        QEvent const *qSto[], uint32_t qLen,
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

```
                      void *stkSto, uint32_t stkSize,
                      QEvent const *ie)
     {
         pthread_attr_t attr;
         struct sched_param param;

(23)     Q_REQUIRE(stkSto == (void *)0); /* p-threads allocate stack internally */

(24)     QEQueue_init(&me->eQueue, qSto, (QEQueueCtr)qLen);
(25)     pthread_cond_init(&me->osObject, 0);

(26)     me->prio = prio;
(27)     QF_add_(me); /* make QF aware of this active object */
(28)     QF_ACTIVE_INIT_(&me->super, ie); /* execute the initial transition */


         /* SCHED_FIFO corresponds to real-time preemptive priority-based scheduler
         * NOTE: This scheduling policy requires the superuser privileges
         */
(29)     pthread_attr_init(&attr);
(30)     pthread_attr_setschedpolicy(&attr, SCHED_FIFO);

         /* see NOTE04 */
(31)     param.sched_priority = prio
                               + (sched_get_priority_max(SCHED_FIFO)
                                  - QF_MAX_ACTIVE - 3);

(32)     pthread_attr_setschedparam(&attr, &param);
(33)     pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

(34)     if (pthread_create(&me->thread, &attr, &thread_routine, me) != 0) {
             /* Creating the p-thread with the SCHED_FIFO policy failed.
             * Most probably this application has no superuser privileges,
             * so we just fall back to the default SCHED_OTHER policy
             * and priority 0.
             */
(35)         pthread_attr_setschedpolicy(&attr, SCHED_OTHER);
(36)         param.sched_priority = 0;
(37)         pthread_attr_setschedparam(&attr, &param);
(38)         Q_ALLEGE(pthread_create(&me->thread, &attr, &thread_routine, me)== 0);
         }
(39)     pthread_attr_destroy(&attr);
     }
     /*..........................................................................*/
     void QActive_stop(QActive *me) {
(40)     me->running = (uint8_t)0; /* stop the event loop in QActive_run() */
(41)     pthread_cond_destroy(&me->osObject); /* cleanup the condition variable */
     }
```

(1) The global Pthread mutex `QF_pThreadMutex_` variable for the QF critical section is defined.

(2) On POSIX systems that support it, you might want to call the `mlockall()` function to lock in physical memory all of the pages mapped by the address space of a process. This prevents non-deterministic swapping of the process memory to disk and back. The standard desktop POSIX does not support `mlockall()`, so it is commented out.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

(3) The `QF_run()` function is called from main() to let the framework execute the application. In this QF port, the `QF_run()` function is used as the "ticker thread" to periodically call the `QF_tick()` function.

(4) The callback function `QF_onStartup()` is called to give the application a chance to perform startup.

(5-6) These two lines of code attempt to set the current thread (the "ticker thread") to the `SCHED_FIFO` scheduling policy and to the maximum priority within that policy.

In POSIX, the scheduler policy closest to real-time is the `SCHED_FIFO` policy, available only with the "superuser" privileges. `QF_run()` attempts to set this policy as well as to maximize its priority, so that the system clock tick ccurrs in the most timely manner. However, setting the `SCHED_FIFO` policy might fail, most probably due to insufficient privileges.

(7) The "ticker" thread runs in loop, as long as the `l_running` flag is set.

(8) The "ticker" thread calls `QF_onClockTick()` outside of any critical section.

(9-10) The "ticker" thread is put to sleep for the rest of the time slice.

The `select()` system call is used here as a fairly portable way to sleep because it seems to deliver the shortest sleep time of just one clock tick. The timeout value passed to `select()` is rounded up to the nearest tick (10 milliseconds on desktop POSIX). The timeout cannot be too short, because the system might choose to busy-wait for very short timeouts. An obvious alternative—the POSIX `nanosleep()` system call—seems to be unable to block for less than two clock ticks (20 milliseconds). Also according to the man pages, the function `select()` on POSIX modifies the timeout argument to reflect the amount of time not slept. Most other implementations do not do this. This quirk is handled in a portable way by always setting the microsecond part of the structure before each `select()` call (see (9))

(11) When the loop exits, the callback function `QF_onCleanup()` is called to give the application a chance to perform cleanup.

(12) The global Pthread mutex `QF_pThreadMutex_` is cleaned up before exit.

(13) The `QF_run()` function exits, which causes the `main()` function to exit. The system terminates the process and shuts down all Pthreads spawned from `main()`.

(14) The exit sequence just described in triggered when the application calls `QF_stop()`, which stops the loop in `QF_run()`.

The following static function thread_routine() specifies the thread function of all active objects.

(15) In this POSIX port, all active object threads execute the same function `thread_routine()`, which has the exact signature expected by POSIX API `pthread_create()`. The parameter `arg` is set to the active object owning in the thread.

(16) The thread routine sets the `QActive.running` flag to continue the local event loop.

(17) The event loop continues as long as the `QActive.running` flag is set.

(18-20) These are the three steps of the active object thread.

(21) After the event loop terminates, the active object is removed from the framework.

(22) The return from the thread routine cleans up the POSIX-thread.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

(23) The `pthread_create()` function allocates the stack space for the thread internally. This assertion makes sure that the stack storage is not provided, because that would be wasteful.

(24) The native QF event queue of the active object is initialized.

(25) The Pthread condition variable is initialized.

(26) The active object's priority is set.

(27) The active object is registered with the QF framework.

(28) The active object's state machine is initialized.

(29-33) The attribute structure for the active object thread is initialized. In the first attempt, the thread is created with the SCHED_FIFO policy.

According to the man pages (for pthread_attr_setschedpolicy()) the only value supported in the POSIX Pthread implementation is PTHREAD_SCOPE_SYSTEM, meaning that the threads contend for CPU time with all processes running on the machine. In particular, thread priorities are interpreted relative to the priorities of all other processes on the machine. This is good, because it seems that if we set the priorities high enough, no other process (or threads running within) can gain control over the CPU. However, QF limits the number of priority levels to QF_MAX_ACTIVE. Assuming that a QF application will be real-time, this port reserves the three highest POSIX priorities for the system threads (e.g., the ticker, I/O), and the rest highest-priorities for the active objects.

(34) The active object Pthread is created. If the thread creation fails, it is most likely due to insufficient privileges to use the real-time policy `SCHED_FIFO`.

(35-37) The thread attributes are modified to use the default scheduling policy `SCHED_OTHER` and priority zero.

(38) The Pthread creation is attempted again. This time it must succeed, or the application cannot continue.

(39) The Pthread attribute structure is cleaned up.

(40) To stop an active object, the `QActive_stop()` function clears the `QActive.running` flag. This stops the active object event loop at line (17), and causes the thread routine to exit.

(41) The condition variable is cleaned up.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

4 Related Documents and References

Document	Location
[PSiCC2] "Practical UML Statecharts in C/C++, Second Edition", Miro Samek, Newnes, 2008	Available from most online book retailers, such as amazon.com. See also: https://www.state-machine.com/psicc2
[QP/C] "QP/C Reference Manual", Quantum Leaps, LLC, 2016	https://www.state-machine.com/qpc
[QP/C++] "QP/C++ Reference Manual", Quantum Leaps, LLC, 2016	https://www.state-machine.com/qpcpp
[QL AN-DPP 08] "Application Note: Dining Philosophers Application", Quantum Leaps, LLC, 2012	https://www.state-machine.com/doc/AN_DPP.pdf

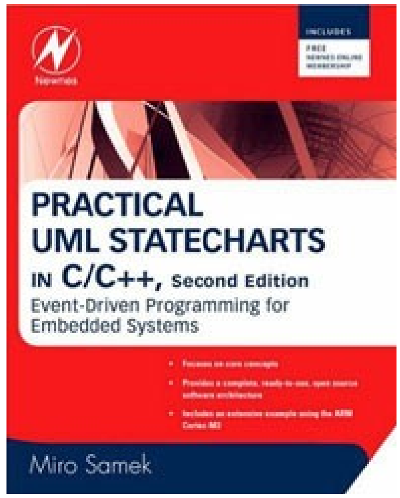
Copyright © Quantum Leaps, LLC. All Rights Reserved.

5 Contact Information

Quantum Leaps, LLC
103 Cobble Ridge Drive
Chapel Hill, NC 27516
USA

+1 919 360-5668 (9-5 EST)
+1 919 869-2998 (FAX)

e-mail: mailto:info@state-machine.com
WEB : https://www.state-machine.com



"*Practical UML Statecharts in C/C++, Second Edition: Event Driven Programming for Embedded Systems*",
by Miro Samek,
Newnes, 2008

Copyright © Quantum Leaps, LLC. All Rights Reserved.