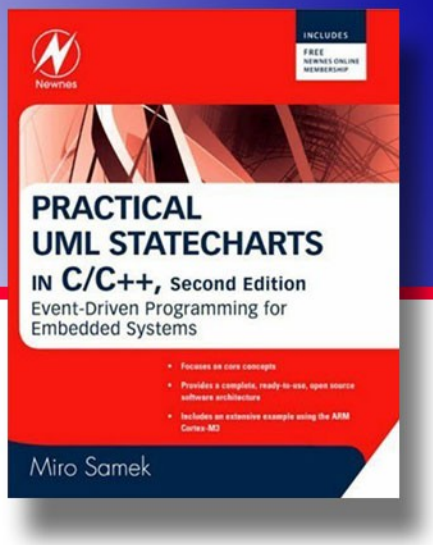
Quantum™ LeaPs
innovating embedded systems



Application Note
QP and ThreadX

Document Revision D
May 2014

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

1 Introduction

This document describes how to use QP™ event-driven framework with the ThreadX™ Real Time Operating System (RTOS) from Express Logic (www.rtos.com). The actual software used in this QDK is described below:

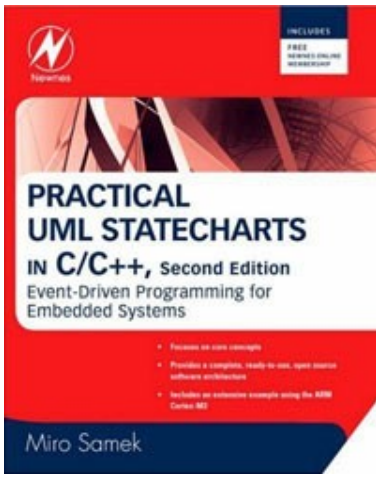
1. ThreadX® v**5.3** (ThreadX RTOS demo for Win32)
2. Microsoft Visual C++ Express 2013
3. QP/C/C++ v**5.3.0** or higher.

NOTE: This document is applicable to both C and C++ versions of QP. Special sections cover the C/C++ differences, whenever such differences are important or at least non-trivial.

NOTE: Even though the Application Note uses a specific ThreadX port (Win32 in this case), the QP-ThreadX integration has been designed **generically** to rely exclusively on the ThreadX API. In other words, the code should require minimal adaptation for any other CPU/compiler platform supported by ThreadX.

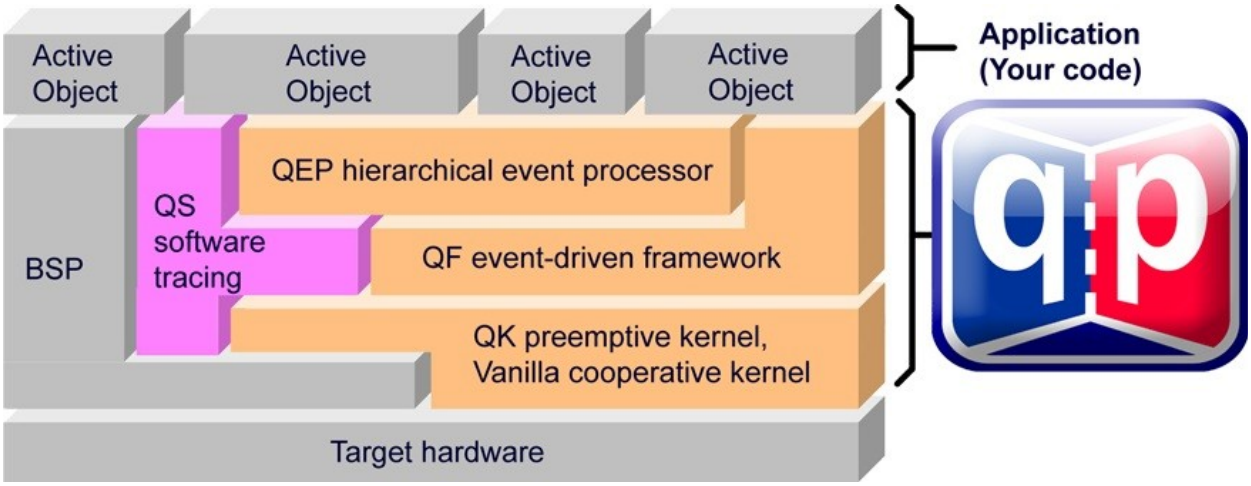
1.1 About QP™

QP™ is a family of very lightweight, open source, active object frameworks for developing event-driven applications based on state machines. QP enables embedded software developers to build 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: Event-Driven Programming for Embedded Systems*" [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 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 (called active objects in UML).

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

1.2 About QM™

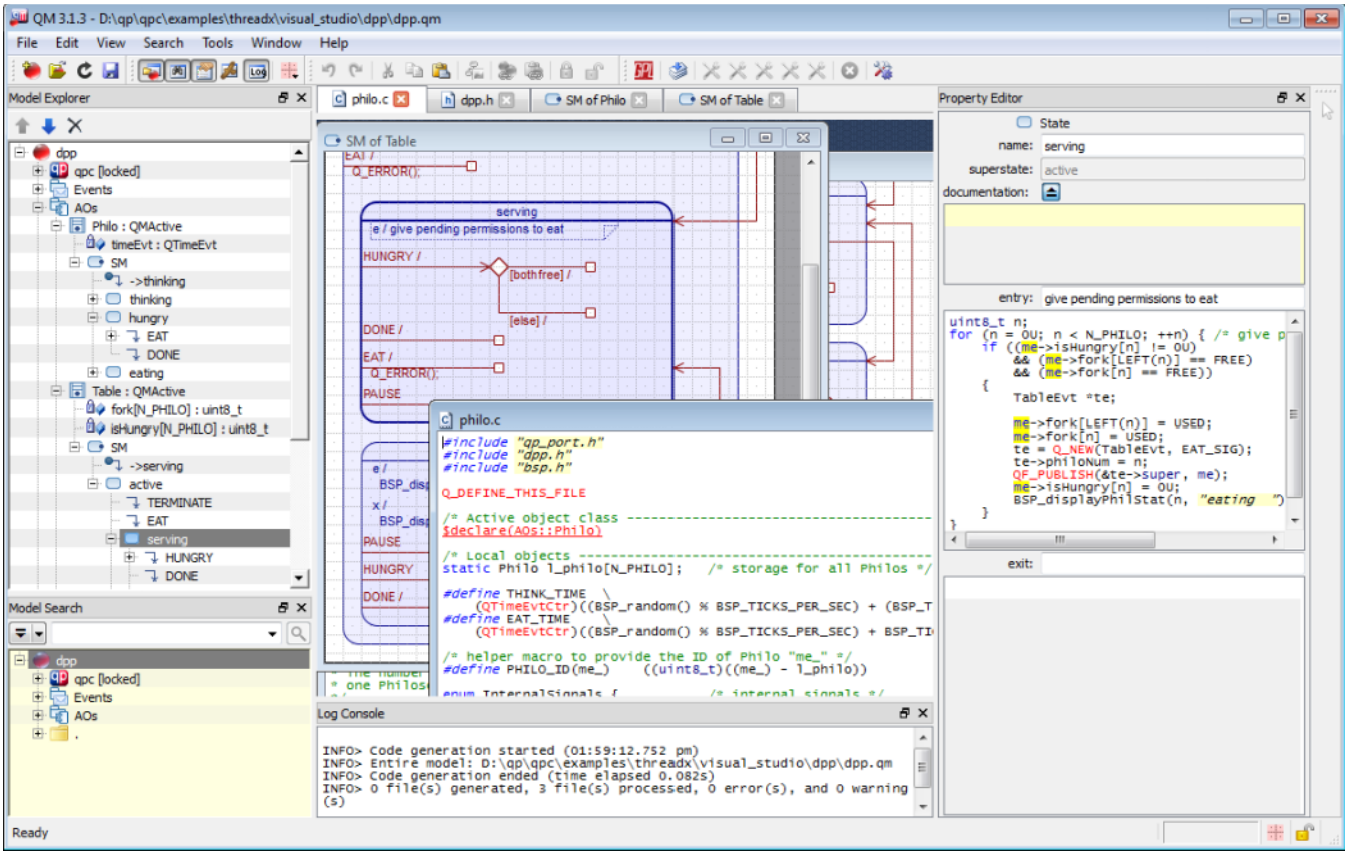
QM™ (QP™ Modeler) is a free, cross-platform, graphical UML modeling tool for designing and implementing real-time embedded applications based on the QP™ state machine frameworks. QM™ itself is based on the Qt framework and therefore runs naively on Windows, Linux, and Mac OS X.

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™.

The code accompanying this App Note contains three application examples: the Dining Philosopher Problem [AN-DPP] modeled with QM.

NOTE: The provided QM model files assume QM version **3.1.3** or higher.

Figure 2: The Dining Philosophers Problem (DPP) model opened in the QM™ modeling tool



1.3 About the QP-ThreadX Integration

- Each QP active object executes in a separate ThreadX thread.
- Each QP active object thread has a unique ThreadX priority.
- The critical section used in QF and QS is based on the ThreadX interrupt control API (`tx_interrupt_control()`). The critical section uses the "save and restore interrupt status" policy, which is safe to use in any context, including ISR context.
- The QP port uses the native ThreadX event queue (`TX_QUEUE`).
- The QP port uses the native ThreadX block pool (`TX_BLOCK_POOL`) to implement event pools.
- The uses ThreadX alarm facility to periodically execute the system clock tick QF_tick().

NOTE: Even though the document uses a specific ThreadX port (Win32 in this case), the QDK has been designed **generically** to rely exclusively on the ThreadX API. In other words, the described code should require minimal adaptation for any other CPU/compiler platform supported by ThreadX.

1.4 Licensing QP

The **Generally Available (GA)** distributions of QP available for download from the www.state-machine.com/downloads website are offered under the same licensing options as the QP baseline code. These available licenses are:

- 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.

2 Directories and Files

This section describes how to install, build, and use QP-ThreadX port. This information is intentionally included early in this document, so that you could start using the software as soon as possible.

To avoid unnecessary and difficult to maintain repetitions, the QP-ThreadX port contains only the code pertinent to the specific platform (ThreadX in this case), but do not contain the platform-independent QP baseline code, which is available separately from http://www.state-machine.com/downloads.

Each port is provided as a ZIP archive that is designed to "plug-into" the directory structure already established after the installation of the QP baseline code.

To install a QP port, download the ZIP file and unzip it into the **same** "QP Root Directory", in which you installed all other QP components. For the sake of further discussion, this directory will be referenced as `<qp>`.

The QP-ThreadX port also requires that you download and install the ThreadX RTOS Demo for Win32 demonstration CD, which is available for a free download from Express Logic at www.rtos.com.

NOTE: This QDK assumes the you have installed the ThreadX Win32 demo (**threadx_53_win32_microsoft.zip**) on your machine and that you have defined the environment variable **THREADX** pointing to the ThreadX installation directory.

2.1 Installation

The QDK code is distributed in a ZIP archive (`qdkc_threadx_<ver>.zip`, where `<ver>` stands for a specific QDK-ThreadX version, such as 4.2.04). You should uncompress the archive into the same directory into which you've installed all the standard QP components. The installation directory you choose will be referred henceforth as QP Root Directory (`<qp>`). The following Listing 1 shows the directory structure and selected files included in the QDK-ThreadX distribution.

Listing 1 Selected Directories and files after installing QP baseline code and the QDK-ThreadX.

```
qpc/                        - QP/C installation directory (qpcpp for QP/C++)
+-3rd_party
| +-threadx/                - ThreadX RTOS Demo for Windows
| | +-docs/                 - ThreadX documentation
| | +-visual_studio/        - Visual Studio development tools
| | | +-demo_threadx/       - directory containing the Visual C++ project
| | | +-ThreadX_Workspace.sln - Visual C++ solution for building the ThreadX demo
| | | +-tx.lib              - ThreadX RTOS library pre-compiled for Win32
| | | +-tx_api.h            - Platform-independent ThreadX API
| | | +-tx_port.h           - Port of ThreadX to Win32adX RTOS demo for Win32
| | +-readme_threadx.txt    - ReadMe file for the ThreadX demo on Windows
|                             (describes the limitations of the demo)
| +-include/                - QP public include files
| | +-qassert.h             - Quantum Assertions platform-independent public include
| | +-qep.h                 - QEP platform-independent public include
| | +-qf.h                  - QF platform-independent public include
| | +-qequeue.h             - native QF event queue include
| | +-qmpool.h              - native QF memory pool include
| | +-qpset.h               - native QF priority set include
| |
| +-examples/               - subdirectory containing the examples
| | +-threadx/              - ThreadX ports
```

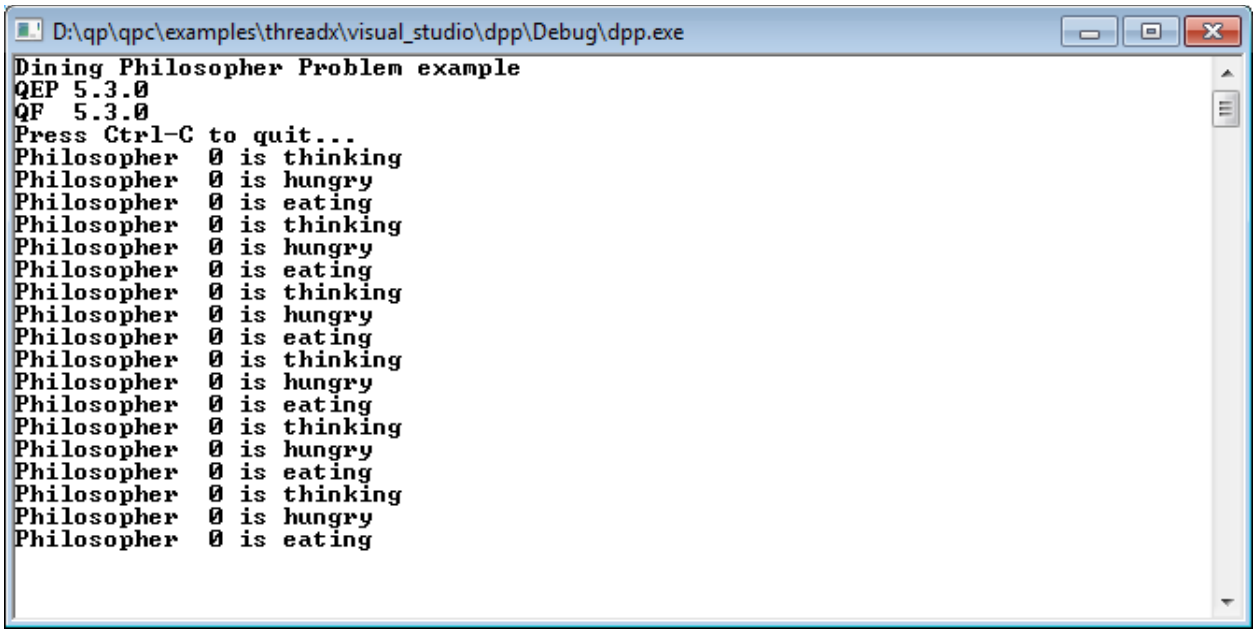
```
| | | +-visual_studio/     - Visual Studio compiler
| | | | +-dpp/             - Dining Philosophers example for ThreadX
| | | | | +-Debug/         - directory containing the Debug build
| | | | | | +-dpp.exe      - Windows executable
| | | | | +-Release/       - directory containing the Release build
| | | | | +-Spy/           - directory containing the Spy build
| | | | | +-dpp.sln        - Visual Studio solution to build the DPP example
| | | | | +-dpp.vcproj     - Visual Studio project to build the DPP example
| | | | | +-dpp.qm         - QM model file for the DPP application
| | | | | +-bsp.c          - Board Support Package for Win32
| | | | | +-bsp.h          - BSP header file
| | | | | +-main.c         - the main function
| | | | | +-philo.c        - the Philosopher active object (generated by QM)
| | | | | +-dpp.h          - the DPP header file (generated by QM)
| | | | | +-table.c        - the Table active object (generated by QM)
| |
| +-ports/                 - QP ports
| | +-threadx/             - ThreadX port (CPU and compiler-independent)
| | | +-qep_port.h         - QEP port
| | | +-qf_port.h          - QF port to ThreadX
| | | +-qf_port.c          - QF port to ThreadX source
| | | +-qs_port.h          - QS port
| | | +-qp_port.h          - QP port
```

NOTE: Most of the code for the DPP example application has been generated automatically by the QM modeling tool from the `dpp.qm` model included in the example directory. Specifically, the following files have been generated: `dpp.h`, `philo.c`, and `table.c`. These files should **not** be edited manually, but rather any changes should be made in QM and the files should be re-generated.

3 Executing the Examples

The QP-ThreadX port demonstrates the ThreadX version of Dining Philosopher example described in the book “Practical UML Statecharts in C/C++, Second Edition” [PSiCC2], as well as in the Quantum Leaps Application Note “Dining Philosophers Problem” [AN QP-DPP 08]. The executables located in the `<qp>\examples\threadx\visual_studio\dpp` directory should run on any Windows PC. The examples are simple console applications. You terminate the application by pressing **Ctrl-C**.

Figure 3 “Dining Philosophers” for ThreadX executing in a Windows console. Note only one Philosopher active object (Philosopher 0).



NOTE: The ThreadX RTOS demo for Win32 is severely limited (see the file readme_threadx.txt in the ThreadX directory (see Listing 1). In particular, the demo version contains a pre-built ThreadX library, which has the following limitations:

- 11 Threads
- 2 Timers
- 2 Event Flag Groups
- 2 Mutexes
- 2 Queues
- 2 Semaphores
- 1 Block Pool
- 1 Byte Pool

These limitations allow only 2 active objects (2 event queues), so the DPP application consists of the Table active object and just one Philosopher active object (Philosopher [0]).

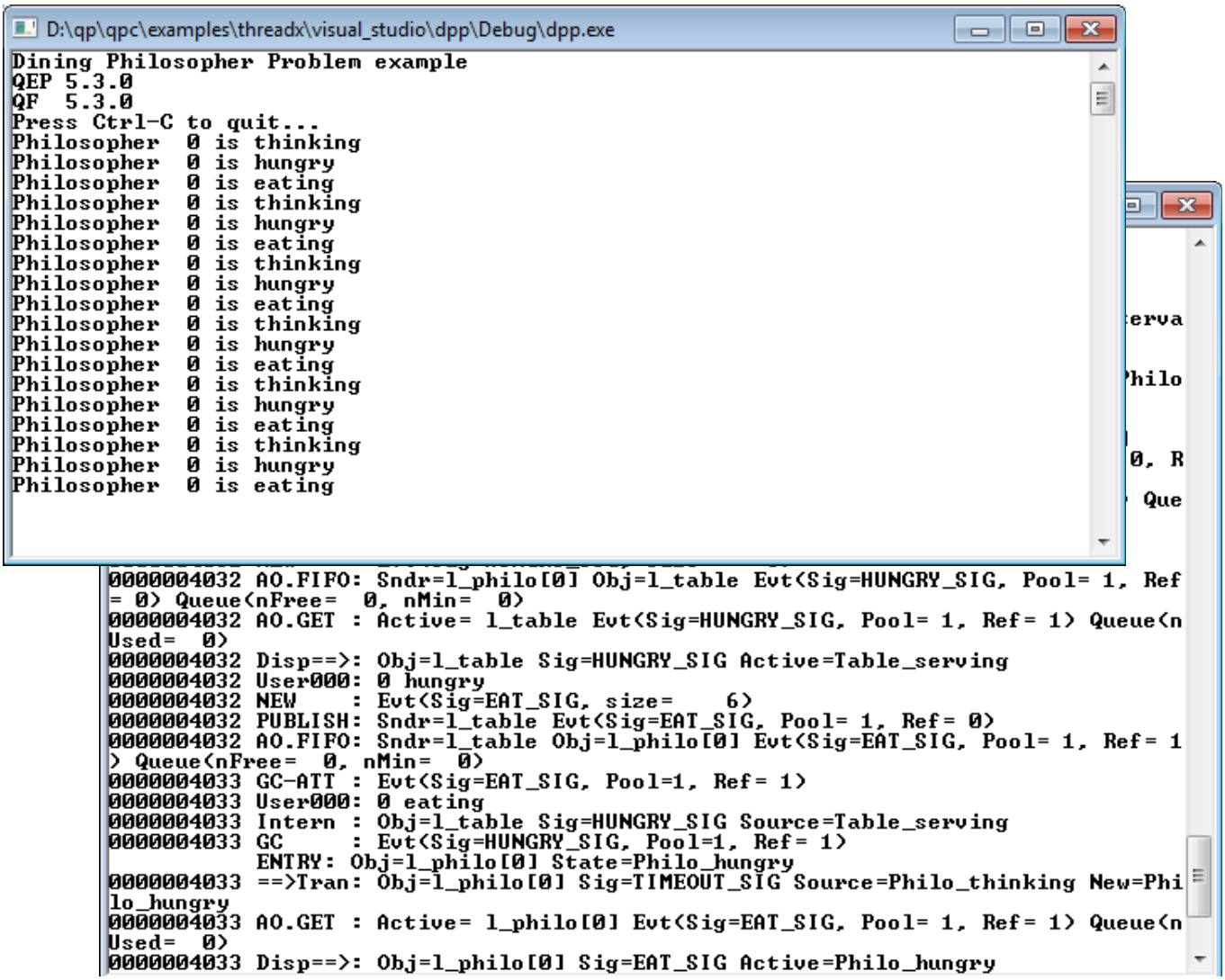
3.1 Executing the QSPY Example

The DPP example included in this QDK™ illustrates how to use the QS™ software tracing instrumentation with ThreadX on Windows. The QS instrumentation in this case sends the QS trace data to the TCP/IP socket specified in the command-line (e.g., `localhost:6602`). The default is `localhost:6601`.

NOTE: The QS instrumentation is only active in the Spy configuration and inactive in the Release and Debug configurations. The Spy executable attempts to open a client connection to the specified TCP/IP server socket. The server socket is provided by the QSPY™ host application, which must be launched **before** starting the DPP application. Failure to launch QSpy™ will cause an abort and immediate exit from the QS example application.

The following command line shows how to configure and launch the QSPY™ host application (given the configuration described in Section 4.2).

```
qspy -t
```



4 The QP-ThreadX Port

As described in Chapter 8 of PSiCC2, porting QF consists of customizing files qep_port.h, qf_port.h, qs_port.h and qf_port.c/.cpp source files, which are located in the respective directories indicated in Listing 1. This section describes the most important points of the QP-ThreadX port.

NOTE: The provided QP-ThreadX port is **generic**, meaning that typically, you don't need to change any of the files of the `qpc\port\threadx\` directory to run QP-ThreadX port on any CPU/Compiler to which ThreadX has been ported, because all the CPU and compiler specifics are handled by the ThreadX RTOS.

4.1 The qep_port.h header file

The port of QEP to ThreadX reuses the ThreadX configuration defined in the `tx_api.h` header file. In particular, the exact-width integer types are defined in terms of the ThreadX types. All this means that you don't need to change the `qep_port.h` header file in any way to port it to a different CPU or compiler. The whole porting effort is limited to porting ThreadX itself and all other QP components adapt automatically. The QEP port header file for ThreadX port is located in `qpc\ports\threadx\-qep_port.h`. The following listing shows the QEP configuration for ThreadX:

Listing 2 The QEP configuration for Windows

```
    #ifndef qep_port_h
    #define qep_port_h

(1) #include <stdint.h>   /* Exact-width types. WG14/N843 C99 Standard */
(2) #include <stdbool.h>  /* Boolean type.      WG14/N843 C99 Standard */

(3) #include "qep.h"      /* QEP platform-independent public interface */

    #endif  /* qep_port_h */
```

(1) The `<stdint.h>` header contains the C99-standard exact-width integer types.

(2) The `<stdbool.h>` header contains the C99-standard Boolean type.

(3) The `qep_port.h` header file must always include the platform-independent QEP interface `qep.h`.

4.2 The QF Port Header File

The QF port header file for ThreadX port is located in `qpc\ports\threadx\qf_port.h`.

Listing 3 qf_port.h header file for the QF port to ThreadX

```
    /* ThreadX event queue and thread types */
(1) #define QF_EQUEUE_TYPE      TX_QUEUE
(2) #define QF_THREAD_TYPE      TX_THREAD

    /* QF priority offset within ThreadX priority numbering scheme, see NOTE1 */
(3) #define QF_TX_PRIO_OFFSET   8

    /* The maximum number of active objects in the application, see NOTE2 */
(4) #define QF_MAX_ACTIVE       (31 - QF_TX_PRIO_OFFSET)

    /* QF interrupt disabling for ThreadX */
(5) #define QF_CRIT_STAT_TYPE    UINT
(6) #define QF_CRIT_ENTRY(stat_) ((stat_) = tx_interrupt_control(TX_INT_DISABLE))
(7) #define QF_CRIT_EXIT(stat_)  ((void)tx_interrupt_control(stat_))

(8) #include "tx_api"       /* ThreadX API */
(9) #include "qep_port.h"   /* QEP port */
(10) #include "qequeue.h"    /* used for event deferral */
(11) #include "qf.h"         /* QF platform-independent public interface */


    /****************************************************************************
    * interface used only inside QF, but not in applications
    */
(12) #ifdef QP_IMPL

        /* TreadX block pool operations */
(13)    #define QF_EPOOL_TYPE_              TX_BLOCK_POOL
        #define QF_EPOOL_INIT_(pool_, poolSto_, poolSize_, evtSize_) \
            Q_ALLEGE(tx_block_pool_create(&(pool_), "P", (evtSize_), \
                     (poolSto_), (poolSize_)) == TX_SUCCESS)

        #define QF_EPOOL_EVENT_SIZE_(pool_) \
            ((uint_fast16_t)(pool_).tx_block_pool_block_size)

(14)    #define QF_EPOOL_GET_(pool_, e_, margin_) do { \
            QF_CRIT_STAT_ \
            QF_CRIT_ENTRY_(); \
            if ((pool_).tx_block_pool_available > (margin_)) { \
                Q_ALLEGE(tx_block_allocate(&(pool_), (void **)&(e_), TX_NO_WAIT) \
                         == TX_SUCCESS); \
            } \
            else { \
                (e_) = (QEvt *)0; \
            } \
            QF_CRIT_EXIT_(); \
        } while (0)

        #define QF_EPOOL_PUT_(dummy, e_) \
```

```
        Q_ALLEGE(tx_block_release((e_)) == TX_SUCCESS)

#endif /* ifdef QP_IMPL */
```

(1) This QF port uses the native ThreadaX event queue `TX_QUEUE`, which is embedded directly in the `QActive` class (see [ThreadX]).

(2) This QF port uses the native ThreadaX thread `TX_THREAD`, which is embedded directly in the `QActive` class (see [ThreadX]).

(3) QF_TX_PRIO_OFFSET specifies the number of highest-urgency ThreadX priorities not available to QP active objects. These highest-urgency priorities might be used by ThreadX threads that run "above" QP active objects. Because the ThreadX priority numbering is "upside down" compared to the QP priority numbering, the ThreadX priority for an active object thread is calculated as follows (see `qf_port.c` in the next section):

```
tx_prio = QF_TX_PRIO_OFFSET + QF_MAX_ACTIVE - qf_prio
```

(4) The maximum number of active objects in QP can be increased to 63, inclusive, but it can be reduced to save some memory. Also, the number of active objects cannot exceed the number of TreadX thread priorities `TX_MAX_PRIORITIES`, because each QP active object requires a unique priority level. Due to the artificial limitations of the ThreadX demo for Windows, `QF_MAX_ACTIVE` is set here lower to not exceed the total available priority levels.

(5-7) One of the most important aspects of the port is the interrupt disabling policy (QF critical section). The QF critical section is defined in terms of the ThreadX critical section. This QF port defines the macro `QF_CRIT_STAT_TYPE`, which means that the policy of “saving and restoring critical section status” is used (see Section 7.3.1 of [PSiCC2])

NOTE: The policy of “saving and restoring critical section status” permits nesting critical sections and is safe in any context, including calling QF services from within a critical section or from ISRs.

(8) The QF port uses the ThreadX services, so it needs to include the `tx_api.h` header file.

(9) The QF port uses the QEP event processor, so it needs to include the `qep_port.h` header file.

(10) The `qequeue.h` header file is included to allow using deferring and recalling events in QF (see Chapter 5 in [PSiCC2]).

(11) The `qf_port.h` header file must always include the platform-independent QF interface.

(12) The following code is used only the internal QF implementation.

(13) This QF port uses the native ThreadaX block pool `TX_BLOCK_POOL` (see [ThreadX]).

(14) The macro for obtaining a block from a pool uses a critical section to check how many blocks are still available. Later, the ThredX call `tx_block_allocate()` is used to allocate the memory block. This call is made from a critical section, but this is OK, because the chosen implementation of critical sections is allowed to nest.

4.3 The QF Port Implementation File

The QF implementation file for ThreadX port is located in `qpc\ports\threadx\qf_port.c`.

**Listing 4 qf_port.c implementation file for the QF port to Windows.
The ThreadX API calls are shown in bold.**

```
(1) #define QP_IMPL           /* this is QP implementation */
    #include "qf_port.h"      /* QF port */
    #include "qf_pkg.h"
    #include "qassert.h"
    #ifdef Q_SPY              /* QS software tracing enabled? */
        #include "qs_port.h"  /* include QS port */
    #else
        #include "qs_dummy.h" /* disable the QS software tracing */
    #endif /* Q_SPY */

    Q_DEFINE_THIS_MODULE("qf_port")

    /*..........................................................................*/
(2) void QF_init(void) {
    }
    /*..........................................................................*/
(3) int_t QF_run(void) {
        QF_onStartup();
        return (int_t)0; /* return success */
    }
    /*..........................................................................*/
(4) void QF_stop(void) {
        QF_onCleanup(); /* cleanup callback */
    }
    /*..........................................................................*/
(5) static void thread_function(ULONG thread_input) { /* ThreadX signature */
        QActive *act = (QActive *)thread_input;
        act->osObject = (uint8_t)1;/* enable thread-loop */
        while (act->osObject) {
(6)         QEvt const *e = QActive_get_(act);
(7)         QMSM_DISPATCH(&act->super, e);
(8)         QF_gc(e); /* check if the event is garbage, and collect it if so */
        }

        QF_remove_(act); /* remove this object from QF */
        Q_ALLEGE(tx_queue_delete(&act->eQueue) == TX_SUCCESS); /* cleanup queue */
        Q_ALLEGE(tx_thread_delete(&act->thread) == TX_SUCCESS);/*cleanup thread */
    }
    /*..........................................................................*/
    void QActive_start_(QActive * const me, uint_fast8_t prio,
                        QEvt const *qSto[], uint_fast16_t qLen,
                        void *stkSto, uint_fast16_t stkSize,
                        QEvt const *ie)
    {
        UINT tx_prio; /* ThreadX priority corresponding to the QF priority prio */

        /* allege that the ThreadX queue is created successfully */
(9)     Q_ALLEGE(tx_queue_create(&me->eQueue,
                    "Q",
```

```
                    TX_1_ULONG,
                    (VOID *)qSto,
                    (ULONG)(qLen * sizeof(ULONG)))
                == TX_SUCCESS);

        me->prio = prio;  /* save the QF priority */
        QF_add_(me);      /* make QF aware of this active object */
        QMSM_INIT(&me->super, ie); /* execute initial transition */

        QS_FLUSH(); /* flush the trace buffer to the host */

        /* convert QF priority to the ThreadX priority */
(10)    tx_prio = QF_TX_PRIO_OFFSET + QF_MAX_ACTIVE - prio;

(11)    Q_ALLEGE(tx_thread_create(&me->thread, /* ThreadX thread control block */
                    "AO",                      /* thread name */
                    &thread_function,          /* thread function */
                    (ULONG)me,                 /* thread input */
                    stkSto,                    /* stack start */
                    stkSize,                   /* stack size in bytes */
                    tx_prio,                   /* ThreadX priority */
                    tx_prio,/*preemption threshold disabled (same as priority) */
                    TX_NO_TIME_SLICE,
                    TX_AUTO_START)
                == TX_SUCCESS);
    }
    /*..........................................................................*/
    void QActive_stop(QActive * const me) {
(12)    me->osObject = (uint8_t)0; /* stop the thread loop */
    }
    /*..........................................................................*/
    #ifndef Q_SPY
(13) bool QActive_post_(QActive * const me, QEvt const * const e,
                        uint_fast16_t const margin)
    #else
    bool QActive_post_(QActive * const me, QEvt const * const e,
                        uint_fast16_t const margin, void const * const sender)
    #endif /* Q_SPY */
    {
        QEQueueCtr nFree;
        bool status;
        QF_CRIT_STAT_

(14)    QF_CRIT_ENTRY_();
(15)    nFree = (QEQueueCtr)me->eQueue.tx_queue_available_storage;

        if (nFree > margin) {

            QS_BEGIN_NOCRIT_(QS_QF_ACTIVE_POST_FIFO, QS_priv_.aoObjFilter, me)
                QS_TIME_();             /* timestamp */
                QS_OBJ_(sender);        /* the sender object */
                QS_SIG_(e->sig);        /* the signal of the event */
                QS_OBJ_(me);            /* this active object (recipient) */
                QS_U8_(e->poolId_);     /* the pool Id of the event */
                QS_U8_(e->refCtr_);     /* the ref count of the event */
                QS_EQC_(nFree);         /* # free entries still available */
                QS_EQC_((QEQueueCtr)0); /* min # free entries (unknown) */
```

```
            QS_END_NOCRIT_()

            if (e->poolId_ != (uint8_t)0) { /* is it a pool event? */
                QF_EVT_REF_CTR_INC_(e); /* increment the reference counter */
            }
            /* posting to the ThreadX message queue must succeed, see NOTE1 */
(16)        Q_ALLEGE(tx_queue_send(&me->eQueue, (VOID *)&e, TX_NO_WAIT)
                     == TX_SUCCESS);

            status = true; /* return success */
        }
        else {
            /* can tolerate dropping evts? */
(17)        Q_ASSERT(margin != (uint_fast16_t)0);

            QS_BEGIN_NOCRIT_(QS_QF_ACTIVE_POST_ATTEMPT, QS_priv_.aoObjFilter, me)
                QS_TIME_();             /* timestamp */
                QS_OBJ_(sender);        /* the sender object */
                QS_SIG_(e->sig);        /* the signal of the event */
                QS_OBJ_(me);            /* this active object (recipient) */
                QS_2U8_(e->poolId_, e->refCtr_); /* pool Id & ref Count */
                QS_EQC_(nFree);         /* # free entries still available */
                QS_EQC_((QEQueueCtr)margin); /* margin requested */
            QS_END_NOCRIT_()

            status = false; /* return failure */
        }
(18)    QF_CRIT_EXIT_();

        return status;
    }
    /*..........................................................................*/
    void QActive_postLIFO_(QActive * const me, QEvt const * const e) {
        QF_CRIT_STAT_
        QF_CRIT_ENTRY_();

        QS_BEGIN_NOCRIT_(QS_QF_ACTIVE_POST_LIFO, QS_priv_.aoObjFilter, me)
            QS_TIME_();             /* timestamp */
            QS_SIG_(e->sig);        /* the signal of this event */
            QS_OBJ_(me);            /* this active object */
            QS_2U8_(e->poolId_, e->refCtr_); /* pool Id & ref Count */
            /* # free entries */
            QS_EQC_((QEQueueCtr)me->eQueue.tx_queue_available_storage);
            QS_EQC_((QEQueueCtr)0); /* min # free entries (unknown) */
        QS_END_NOCRIT_()

        if (e->poolId_ != (uint8_t)0) { /* is it a pool event? */
            QF_EVT_REF_CTR_INC_(e); /* increment the reference counter */
        }

        /* LIFO posting must succeed, see NOTE1 */
(19)    Q_ALLEGE(tx_queue_front_send(&me->eQueue, (VOID *)&e, TX_NO_WAIT)
                 == TX_SUCCESS);

        QF_CRIT_EXIT_();
    }
    /*..........................................................................*/
```

```
    QEvt const *QActive_get_(QActive * const me) {
        QEvt const *e;
        QS_CRIT_STAT_

(20)     Q_ALLEGE(tx_queue_receive(&me->eQueue, (VOID *)&e, TX_WAIT_FOREVER)
                 == TX_SUCCESS);

        QS_BEGIN_(QS_QF_ACTIVE_GET, QS_priv_.aoObjFilter, me)
            QS_TIME_();             /* timestamp */
            QS_SIG_(e->sig);        /* the signal of this event */
            QS_OBJ_(me);            /* this active object */
            QS_2U8_(e->poolId_, e->refCtr_); /* pool Id & ref Count */
            /* # free entries */
            QS_EQC_((QEQueueCtr)me->eQueue.tx_queue_available_storage);
        QS_END_()

        return e;
    }
```

(1) The `qf_port.c` file is considered part of the QP implementation, because it needs access to the facilities used inside the QP framework, but not in the applications.

(2) The `QF_init()` function is empty, because ThreadX does not need to be initialized from the QF framework, but rather it uses a different initialization mechanism (see Section 5).

(3) `QF_run()` calls the `QF_onStartup()` callback and returns to the caller. Typically, the `QF_onStartup()` callback should start the clock tick service to call `QF_tickX_()` to service the QF time events.

NOTE: The `QF_run()` function does not contain any blocking call, so it quickly returns. This is rather atypical for QF and is specific to the ThreadX port (see the next section “Application Initialization”)

(4) The `QF_stop()` function has not much use in the ThreadX port, because it makes little sense to stop the QF framework. But just in case `QF_stop()` simply calls the cleanup callback.

(5) Under a traditional RTOS, like ThreadX, all active object threads execute the same function `thread_function()`, which implements the basic event loop of an active object. The thread function has the exact signature expected by ThreadX. The parameter `thread_input` is set to the pointer to the active object owning the thread.

(6-8) The three operations of the active object run-to-completion (RTC) step are performed in the event loop of the active object thread function.

(9) The first step in starting an active object is creating the event queue by the ThreadX call `tx_queue_create()`.

(10) The QF priority (argument ‘prio’) is mapped to the corresponding ThreadX priority.

NOTE: ThreadX uses the “upside down” priority scheme in which 0 represents the highest possible urgency and higher numerical values represent lower urgency of the ThreadX threads. This happens to be exactly the opposite of the QF priority numbering scheme.

(11) The active object thread is created by the ThreadX call `tx_thread_create()`. The thread creation must be successful, otherwise the application cannot continue.

(12) To stop an active object the flag `me->running` is cleared, which causes exit from the event loop and termination of the thread routine. Additionally, to clean up the TheradX queue is deleted.

This QF port uses the ThreadX message queues as event queues for active objects, instead of the native QF event queue. Therefore, the native QF implementations of `QActive_post_()`, `QActive_postLIFO()`, and `QActive_get_()` cannot be used and must be replaced with the ThreadX-specific code. The rest of the listing defines these three functions for the ThreadX port.

(13) The signature of the `QActive_post_()` operation depends on the build configuration. In the Spy configuration, the function takes additional '`sender`' argument.

(14) The QF/ThreadX critical section is entered.

(15) The number of free (available) entries in the queue is extracted from the `tx_queue_available_storage` member of the ThreadX `TX_QUEUE` structure.

(16) In QF only pointers to events are sent through the ThreadX queues using the `tx_queue_send()` operation, so the whole message for the ThreadX queue consists of the pointer to event pointer. This ThreadX API is called inside a critical section, but this is okay, because the QF/ThreadX critical section implementation allows nesting of critical sections. The `Q_ALLEGE()` assertion ensures that the ThreadX queue is able to accept the message immediately and **without blocking**.

NOTE: The `QActive_post_()` and `QActive_postLIFO_()` functions can be called from the ISR context, which cannot block.

(17) If the queue has not enough of available entries, the assertion checks whether the caller can tolerate failures in posting events, which is the case when `margin!=0`.

(18) The critical section is exited.

(19) The `QActive_postLIFO_()` operation is implemented with the ThreadX function `tx_queue_front_send()`. Again, this ThreadX API is called inside a critical section, but this is okay, because the QF/ThreadX critical section implementation allows nesting of critical sections.

(20) The `QActive_get_()` operation is implemented blocks indefinitely on an empty queue using the `tx_queue_receive()` ThreadX API.

5 Application Initialization

The ThreadX RTOS requires a very specific application initialization. In particular, the control must be given to ThreadX by a call to the `tx_kernel_enter()` function [ThreadX]. This function performs some initialization of ThreadX and then calls the `tx_application_define()` callback function, which typically starts the ThreadX threads. The following listing shows how to apply this initialization sequence to QF-based application.

NOTE: The application can contain any number of ThreadX threads that run outside of the QF framework. The raw ThreadX threads can communicate with QF active objects by posting or publishing QF events. The QF facilities `QACTIVE_POST()` or `QF_PUBLISH()` are designed to be callable from any context, not necessarily QF active object.

The QF active objects can also communicate with the raw ThreadX threads using any of the ThreadX APIs, such as sending a message to a ThreadX queue, signaling a semaphore, or any other mechanism which does **not** block the caller.

Listing 5 Initialization of a ThreadX application

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

    /* Local-scope objects -----------------------------------------------------*/
    static QEvt const *l_tableQueueSto[N_PHILO];
    static QEvt const *l_philoQueueSto[N_PHILO][N_PHILO];
    static QSubscrList l_subscrSto[MAX_PUB_SIG];
    static union SmallEvents {
        void   *e0;  /* minimum event size */
        uint8_t e1[sizeof(TableEvt)];
        /* ... other event types to go into this pool */
    } l_smlPoolSto[2*N_PHILO + 10];   /* storage for the small event pool */

    static ULONG l_philoStk[N_PHILO][256]; /* stacks for the Philosophers */
    static ULONG l_tableStk[256];          /* stack for the Table */

    /*..........................................................................*/
    int main(int argc, char *argv[]) {
(1)     Philo_ctor();  /* instantiate all Philosopher active objects */
        Table_ctor();  /* instantiate the Table active object */

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

(3)     tx_kernel_enter(); /* transfer control to the ThreadX RTOS */

        return 0; /* NOTE: tx_kernel_enter() does not actually return */
    }
    /*..........................................................................*/
(4) void tx_application_define(void *first_unused_memory) {
        uint8_t n;

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

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

```
                                                    /* initialize event pools... */
        QF_poolInit(l_smlPoolSto, sizeof(l_smlPoolSto), sizeof(TableEvt));

        for (n = 0; n < N_PHILO; ++n) {          /* start the active objects... */
(6)         QActive_start(AO_Philo[n], (uint8_t)(n + 1),
                          l_philoQueueSto[n], Q_DIM(l_philoQueueSto[n]),
                          l_philoStk[n], sizeof(l_philoStk[n]), (QEvt *)0);
        }
(7)     QActive_start(AO_Table, (uint8_t)(N_PHILO + 1),
                      l_tableQueueSto, Q_DIM(l_tableQueueSto),
                      l_tableStk, sizeof(l_tableStk), (QEvt *)0);

        QS_OBJ_DICTIONARY(l_smlPoolSto);

(8)     QF_run();  /* run the QF application */
    }
```

(1) The active object “constructors” in C should be called explicitly as early as possible (e.g., at the beginning of `main()`). In C++, static constructors execute even before `main()`.

(2) The BSP can be also initialized before giving control to ThreadX

(3) The control is given to ThreadX via the `tx_kernel_enter()` API call.

(4) ThreadX invokes the callback function `tx_application_define()` to start ThreadX threads.

(5) The QF framework is initialized.

(6-7) The active objects are started.

(8) The `QF_run()` function passes control to QF. `QF_run()` returns quickly. This causes the callback `tx_application_define()` to return to ThreadX, which eventually starts multitasking.

5.1 Calling the QF System Clock Tick

Any QF application that uses QF time events needs to call the `QF_tickX_()` function periodically.

NOTE: Starting with QP 5.x, QF supports multiple clock tick rates. The QF applications are then responsible to call the `QF_tickX_()` function for **each tick rate** that they use.

In ThreadX, the QF applications can use the ThreadX timer facilities to call the `QF_tickX_()` function to handle the armed QF time events.

NOTE: .The ThreadX timers execute in the context of either ISR or a ThreadX thread, depending on the setting of the `TX_TIMER_PROCESS_IN_ISR` macro. In the QP5, the `QF_tickX_()` function can be called in both context **safely**.

The following listing shows how to create a ThreadX timer to call the `QF_tickX_()` function directly as a callback (see file `qpc\examples\threadx\visual_studio\dpp\bsp.c`)

Listing 6 Setting up a ThreadX timer to periodically call QF_tickX_()

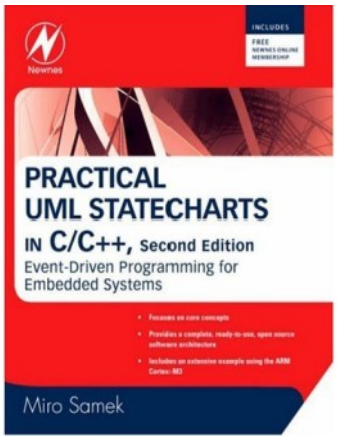
```
void QF_onStartup(void) {
 /*
 * NOTE:
 * This application uses the ThreadX timer to periodically call
 * the QF_tickX_(0) function. Here, only the clock tick rate of 0
 * is used, but other timers can be used to call QF_tickX_() for
 * other clock tick rates, if needed.
 *
 * The choice of a ThreadX timer is not the only option. Applications
 * might choose to call QF_tickX_() directly from timer interrupts
 * or from active object(s).
 */
 Q_ALLEGE(tx_timer_create(&l_tick_timer, /* ThreadX timer object */
              "QF",      /* name of the timer */
              (VOID (*)(ULONG))&QF_tickX_, /* expiration function */
              0U,        /* expiration function input (tick rate) */
              1U,        /* initial ticks */
              1U,        /* reschedule ticks */
              TX_AUTO_ACTIVATE) /* automatically activate timer */
          == TX_SUCCESS);
    . . .
}
```

6 References

Document	Location
[Samek 08] “Practical UML Statecharts in C/C++, Second Edition: Event Driven Programming for Embedded Systems”, Miro Samek, Newnes, 2008	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 AN-DPP 08] “Application Note: Dining Philosophers Application”, Quantum Leaps, LLC, 2008	http://www.state-machine.com/doc/AN_DPP.pdf
[QP AN-DIR 06] “Application Note: QP Directory Structure”, Quantum Leaps, LLC, 2006	http://www.state-machine.com/doc/AN_QP_Directory_Structure.pdf
[ThreadX] “ThreadX User Guide”, Express Logic, 2011	Included in the ThreadX RTOS demo for Win32 as PDF file: ThreadX_User_Guide.pdf

7 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
http://www.state-machine.com



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