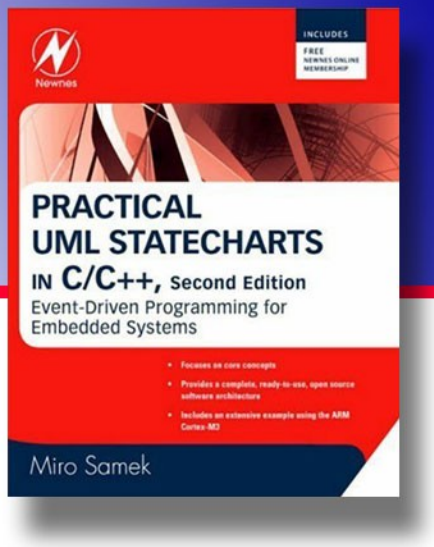
Quantum™ LeaPs
innovating embedded systems



Application Note
QP™ and Windows (Win32)

Document Revision G
November 2011

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

Table of Contents

1 Introduction

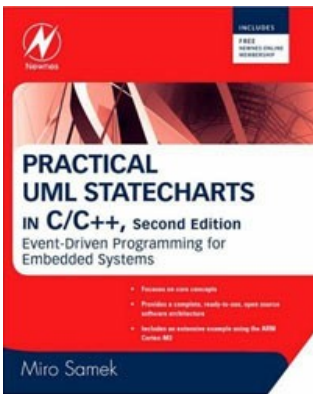
This Application Note describes how to use QP™ state machine frameworks with the Microsoft Windows including desktop Windows (e.g., **Windows XP**) as well as embedded Windows (**Windows CE**). This document covers console-style applications (without the GUI), as well as **Windows GUI applications** built with the raw Win32 API and with the MFC.

Integrating QP with the Win32 GUI API is interesting for at least two reasons. First, QP generally nicely complements any GUI system, such as the Win32 API, by providing the desperately needed high-level "screen logic" to control the overall behavior of the application, while the GUI system provides the low level screen rendering and GUI event generation. The multithreaded character of the QP enables building efficient, multithreaded Windows applications at a much higher level than Win32 threads and without fiddling directly with the troublesome low-level mechanisms such as Win32 critical sections, Win32 event objects, and so on.

The second compelling reason for integration between QP and GUI is emulation of embedded devices on the desktop. Windows-based desktop systems often make excellent platforms to **develop**, **test**, and **debug** embedded applications. Adding a GUI, opens the possibility of simulating the whole "look and feel" of the embedded device, including the front panel with buttons, knobs, LCDs (both segmented and graphical), and so on.

> **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.

1.1 About QP™

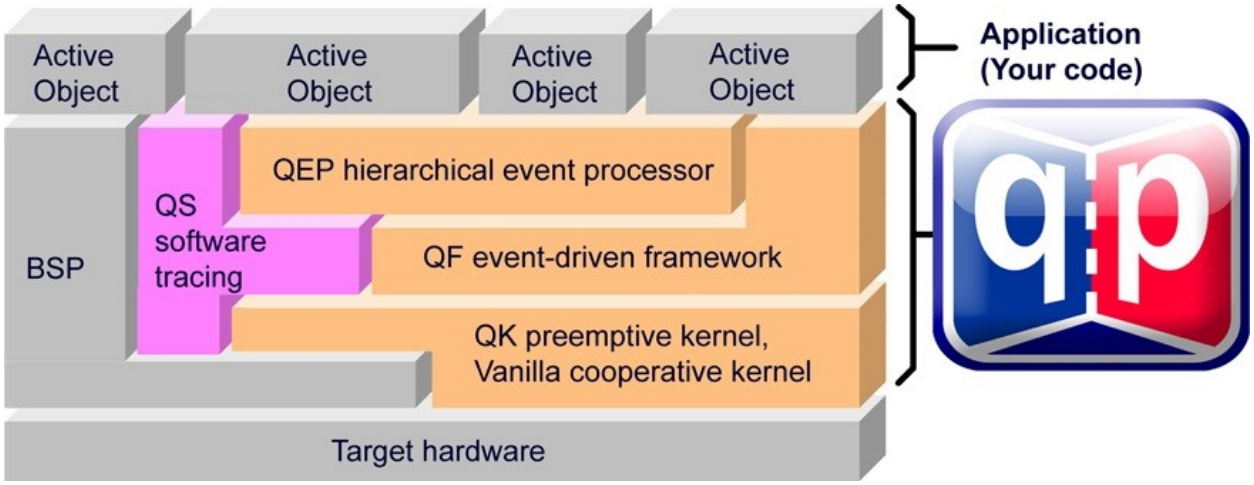


QP™ is a family of very lightweight, open source, state machine-based frameworks for developing event-driven applications. QP enables building well-structured embedded applications as a set of concurrently executing hierarchical state machines (UML statecharts) directly in C or C++ without big tools. QP is described in great detail in the book "Practical UML Statecharts in C/C++, Second Edition: 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. QP can work with or without a traditional RTOS or OS. In the simplest configuration, QP can completely replace a traditional RTOS. QP can manage up to 63 concurrently executing tasks structured as state machines (called active objects in UML).

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



1.2 Key Features of this QP on Windows

1. The QP™ allows very straightforward development of **multithreaded**, event-driven application in the Windows environment (both desktop Windows and Windows CE). No knowledge of Win32 threads or the multithreading Win32 API is necessary.
2. QP™ allows using the Win32 graphics API, that is, the application can have a GUI.
3. QP™ allows tightly integration with the MFC (the QP/C++ version).

1.3 About the QP™-Port to Windows

1. A QP application under Windows is a single Win32 process.
2. Each active object executes in a separate Win32 thread.
3. The real-time framework (QF) does not use interrupts and handles everything at the task level (including the QF clock tick).
4. The critical section used in QF and QS is based on the Win32 critical-section object (CRITICAL_SECTION)
5. The Windows port uses the built-in QF event queue (QEQueue), with the Win32 event object used to block on an empty queue (via the WaitForSingleObject() Win32 API call).
6. The port uses the built-in QF memory pools (QMPool) to handle dynamically allocated events.

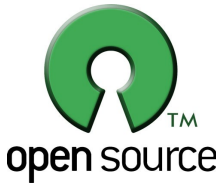
1.4 About the Tools

The QP ports have been prepared and tested with the following tools:

- Visual C++ 2008 and Visual C++ 2010 for the desktop Windows,
- GNU gcc under MinGW.
- eMmbedded Visual C++ for the embedded Windows

1.5 Licensing the QP™

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

2 Building the QP Libraries

All QP components are deployed as libraries that you statically link to your application. The pre-built libraries for QEP, QF, and QS are provided inside the `<qp>\ports\80x86\win32\` directory. Normally, you should have no need to re-build the libraries. However, if you want to modify QP code or you want to apply different settings, this section describes steps you need to take to rebuild the libraries yourself.

2.1 Building QP libraries with Visual Studio

For the Visual C++ port, you can use the provided Visual C++ solution `<qp>\ports\80x86\win32\-vc2008\all_qp.sln`. The solution imports correctly to Visual C++ 2010.

The build process should produce the QP libraries in the location: `<qp>\ports\-80x86\win32\vc2008\Debug\`. To build the Release and Spy configurations, please select them in the Visual Studio IDE and repeat the build for each configuration.

2.2 Building QP libraries with MinGW

For the MinGW port, you can use the provided Makefile in `<qp>\ports\80x86\win32\mingw\Makefile`.. This Makefile supports three build configurations: Debug, Release, and Spy. You choose the build configuration by providing the `CONF` argument to the `make`. The default configuration is "`dbg`". Other configurations are "`rel`", and "`spy`". The following table summarizes the commands to invoke `make`.

Table 1 Make commands for the debug, release, and spy software versions

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

NOTE: The provided Makefile assumes that the MinGW\bin directory is added to the PATH.

3 Executing the Examples

This standard QP distribution contains of several examples 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\80x86\win32\` directory should run on just about any Windows PC. Most examples are simple console applications, but some demonstrate the use of the Win32 GUI. The code can be also recompiled and executed on Windows CE.

NOTE: The WinCE port of QP is exactly identical to the desktop Windows port. The only differences between WinCE show up in the applications, because WinCE uses only **Unicode** strings.

Figure 2: “Dining Philosophers” GUI example executing in Windows CE emulator.

3.1 Executing the QP-Spy™ Examples

Most QP examples for Windows illustrate how to use the QP-Spy™ software tracing instrumentation. 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 QS 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 5.1).

```
qspy -t -S2 -E4 -Q4 -P4 -B4 -C4 -m\tmp\dpp.mat
```

This particular command line configures QSpy to produce a MATLAB output into the file `\tmp\dpp.mat`.

4 The QEP Port

The port of QEP to Win32 (both XP and CE) is very straightforward and requires merely the recompilation of the QEP sources and building the library `qep.lib`.

The configuration of QEP is achieved via the QEP-port header file. The QEP header file for the Win32 port to 80x86, Visual C++ 6.0 is located in `<qp>\ports\80x86\win32\vc2008\qep_port.h`. The Win32 port to MinGW is identical as to VC++ and is located in `<qp>\ports\80x86\win32\mingw\qep_port.h`.

The following Listing 1 shows the QEP configuration used in the Windows ports:

Listing 1: The QEP configuration for Windows

```
#ifndef qep_port_h
#define qep_port_h

#define QEP_SIGNAL_SIZE 2

            /* Exact-width types. WG14/N843 C99 Standard, Section 7.18.1.1 */
typedef signed   char      int8_t;
typedef signed   short     int16_t;
typedef signed   int       int32_t;
typedef unsigned char      uint8_t;
typedef unsigned short     uint16_t;
typedef unsigned int       uint32_t;

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

#endif                                                      /* qep_port_h */
```

The pre-configured signal size is 2-bytes, meaning that the dynamic range of signals is 16-bits (64K signals). This dynamic range plus the other data member of the `QEvent` structure add up to 4-bytes, which is very efficiently represented on 32-bit machines.

You can still extend the `QSignal` range to 32-bits, but then the size of the `QEvent` structure will be less-optimal 6-bytes.

4.1 The Fixed-Width Integer Types

As described in PSiCC2, QP uses the standard fixed-width integers. The MinGW version uses the C-99 standard `<stdint.h>` header file defined in the header file. The Visual C++ compilers (both VC++ 2008and VC++ 2010) don't provide the `<stdint.h>` header file, so the exact-width integer types used in QP are simply typedef'ed in the qep_port.h header file.

5 The QF Port

As described in Chapter 8 of PSiCC2, porting QF consists of customizing files `qf_port.h` and `qf_port.c/qf_port.cpp`, which are located in the respective directories indicated in Error: Reference source not found. This section describes the most important elements of these files.

5.1 The QF Port Header File

The QF header file for the desktop Windows port is located in `<qp>\ports\80x86\win32\vc2008\-qf_port.h`.

Listing 2: qf_port.h header file for the QF port to Windows.

```
    #ifndef qf_port_h
    #define qf_port_h

                                          /* Win32 event queue and thread types */
(1) #define QF_EQUEUE_TYPE              QEQueue
(2) #define QF_OS_OBJECT_TYPE           HANDLE
(3) #define QF_THREAD_TYPE              HANDLE

                      /* The maximum number of active objects in the application */
(4) #define QF_MAX_ACTIVE               63
                    /* The maximum number of event pools in the application */
    #define QF_MAX_EPOOL                8
                         /* various QF object sizes configuration for this port */
    #define QF_EVENT_SIZ_SIZE           4
    #define QF_EQUEUE_CTR_SIZE          4
    #define QF_MPOOL_SIZ_SIZE           4
    #define QF_MPOOL_CTR_SIZE           4
    #define QF_TIMEEVT_CTR_SIZE         4

                                          /* Win32 critical section, see NOTE01 */
    /* QF_CRIT_STAT_TYPE not defined */
(5) #define QF_CRIT_ENTRY(dummy)        EnterCriticalSection(&QF_win32CritSect_)
(6) #define QF_CRIT_EXIT(dummy)         LeaveCriticalSection(&QF_win32CritSect_)

(7) #include <windows.h>                                          /* Win32 API */
    #include "qep_port.h"                                         /* QEP port */
    #include "qequeue.h"                              /* Win32 needs event-queue */
    #include "qmpool.h"                               /* Win32 needs memory-pool */
    #include "qf.h"                /* QF platform-independent public interface */


    /****************************************************************************
    * interface used only inside QF, but not in applications
    */
                                       /* Win32 OS object object implementation */
(8) #define QACTIVE_EQUEUE_WAIT_(me_) \
        while ((me_)->eQueue.frontEvt == (QEvent *)0) { \
            QF_INT_UNLOCK_(); \
            (void)WaitForSingleObject((me_)->osObject, (DWORD)INFINITE); \
            QF_INT_LOCK_(); \
        }

(9) #define QACTIVE_EQUEUE_SIGNAL_(me_) \
```

```
        (void)SetEvent((me_)->osObject)

(10) #define QACTIVE_EQUEUE_ONEMPTY_(me_) ((void)0)

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

(12) extern CRITICAL_SECTION QF_win32CritSect_;        /* Win32 critical section */

(13) void QF_setTickRate(uint32_t ticksPerSec);          /* set clock tick rate */

     #endif                                                        /* qf_port_h */
```

(1) This QF port to Windows uses the native QF event queue (see PSiCC2).

(2) The Win32 event object is used to block the calling active object thread when the event queue is empty.

(3) The Win32 thread handle is used to refer to the Win32 thread associated with each active object.

(4) The Win32 port is configured to the maximum sizes of parameters.

(5-6) QF, like all real-time frameworks needs to execute certain sections of code indivisibly to avoid data corruption. The most straightforward way of protecting such critical sections of code is disabling and enabling interrupts, which Win32 API really does not allow. Instead, this QF port to Windows uses a single Win32 **critical section object** `CRITICAL_SECTION` to protect all critical sections. The Win32 critical section has the ability to nest, so the QF mechanism of "saving and restoring interrupt status" is not used.

NOTE: The Win32 critical section implementation behaves differently than interrupt locking. A single global Win32 critical section ensures that only one thread at a time can execute a critical section, but it does not guarantee that a context switch cannot occur within the critical section. In fact, such context switches probably will happen, but they should not cause concurrency hazards because the critical section eliminates all race conditions.

Unlike simply disabling and enabling interrupts, the critical section approach is also subject to priority inversions. Various versions of Windows handle priority inversions differently, but it seems that most of the Windows flavors recognize priority inversions and dynamically adjust the priorities of threads to prevent it. Please refer to the MSN articles for more information.

---

(7) The `<windows.h>` contains the Win32 API.

(8) The actual blocking of the active object thread to wait on an empty queue is implemented through the macro `QACTIVE_OSOBJECT_WAIT_()` that the `QActive_get__()` function invokes when the event queue is empty. As described in Chapter 7 of PSiCC2, the macro `QACTIVE_OSOBJECT_WAIT_()` is called within a critical section.

NOTE: The `WaitForsingleObject()` Win32 API is called within a `do-while` loop rather than being called only once. The reason is that the event-queue object `osObject` can be occasionally signaled more than once between successive invocations of the `QActive_get_()`.

Consider the following scenario. Active object A calls `QActive_get_()` to retrieve an event from its event queue that holds one event (most typical case). The queue becomes empty, which is indicated by setting the "front event" `eQueue.frontEvt` to `NULL`. After retrieving the event, active object A starts processing the

---

event. However, active object B preempts A and posts a new event for active object A. `QActive_postFIFO()` signals the `osObject` (`QActive_postFIFO()` invokes `QACTIVE_OSOBJECT_SIGNAL_()` macro). When active object A comes to retrieve the next event, the queue is not empty (the "front event" `eQueue.frontEvt` is not `NULL`), so the queue does not attempt to block, but instead `QActive_get_()` returns the event immediately. This time around, active object A processes the event to completion without any additional events being posted to its event queue. When active object A comes to retrieve yet another event (its thread calls `QActive_get_()` again), it attempts to block because the "front event" correctly indicates that the event queue is empty. However, the Win32 event object osObject is signaled, and `WaitForSingleObject()` in macro `QACTIVE_OSOBJECT_WAIT_()` does not block! The do-while loop in the `QACTIVE_OSOBJECT_WAIT_()` macro saves the day, because it causes another call to `WaitForSingleObject()` to truly block the active object A until it receives an event to process.

(9) The signaling of the active object thread blocked on the empty event queue is implemented through the Win32 API call `SetEvent((me_)->osObject)`

(10) There is nothing to do in this port when the queue becomes empty.

(11) This port uses the native QF memory pools, described in Chapter 7 of PSiCC2.

(12) The global Win32 critical section is declared.

(13) The function `QF_setTickRate()` allows you to adjust the tick rate within the granularity provided by the hardware system clock tick.

5.2 The QF Port Implementation File

The QF implementation file for the desktop Windows port is located in `<qp>\qf\80x86\win32\vc2008\-qf_port.c`.

Listing 3: qf_port.c implementation file for the QF port to Windows. The Win32 API calls are shown in bold.

```
    /* Global objects ----------------------------------------------------------*/
    CRITICAL_SECTION QF_win32CritSect_;

    /* Local objects -----------------------------------------------------------*/
    static DWORD WINAPI thread_function(LPVOID arg);
    static DWORD l_tickMsec = 10;  /* clock tick in msec (argument for Sleep()) */
    static uint8_t l_running;

    /*..........................................................................*/
    const char Q_ROM *QF_getPortVersion(void) {
        static const char Q_ROM version[] =  "4.0.00";
        return version;
    }
    /*..........................................................................*/
    void QF_init(void) {
(1)     InitializeCriticalSection(&QF_win32CritSect_);
    }
    /*..........................................................................*/
    void QF_stop(void) {
(2)     l_running = (uint8_t)0;
    }
    /*..........................................................................*/
    void QF_run(void) {
```

```
        l_running = (uint8_t)1;
(3)     QF_onStartup();                                 /* startup callback */
            /* raise the priority of this (main) thread to tick more timely */
(4)     SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_TIME_CRITICAL);
        while (l_running) {
(5)         QF_tick();                                  /* process a time tick */
(6)         Sleep(l_tickMsec);                  /* wait for the tick interval */
        }
(7)     QF_onCleanup();                                 /* cleanup callback */
(8)     DeleteCriticalSection(&QF_win32CritSect_);
    }
    /*..........................................................................*/
    void QF_setTickRate(uint32_t ticksPerSec) {
        l_tickMsec = 1000UL / ticksPerSec;
    }
    /*..........................................................................*/
    void QActive_start(QActive *me, uint8_t prio,
                       QEvent const *qSto[], uint32_t qLen,
                       void *stkSto, uint32_t stkSize,
                       QEvent const *ie)
    {
        int p;

 (9)    Q_REQUIRE((stkSto == (void *)0)    /* Windows allocates stack internally */
(10)              && (stkSize != 0));

(11)    QEQueue_init(&me->eQueue, qSto, (QEQueueCtr)qLen);
(12)    me->osObject = CreateEvent(NULL, FALSE, FALSE, NULL);
        me->prio = prio;
(13)    QF_add_(me);                        /* make QF aware of this active object */
(14)    QF_ACTIVE_INIT_(&me->super, ie);          /* execute initial transition */

(15)    me->thread = CreateThread(NULL, stkSize,
                                  &thread_function, me, 0, NULL);
        Q_ASSERT(me->thread != (HANDLE)0);            /* thread must be created */

(16)    switch (me->prio) {                 /* remap QF priority to Win32 priority */
            case 1:
                p = THREAD_PRIORITY_IDLE;
                break;
            case 2:
                p = THREAD_PRIORITY_LOWEST;
                break;
            case 3:
                p = THREAD_PRIORITY_BELOW_NORMAL;
                break;
            case (QF_MAX_ACTIVE - 1):
                p = THREAD_PRIORITY_ABOVE_NORMAL;
                break;
            case QF_MAX_ACTIVE:
                p = THREAD_PRIORITY_HIGHEST;
                break;
            default:
                p = THREAD_PRIORITY_NORMAL;
                break;
        }
(17)    SetThreadPriority(me->thread, p);
```

```
    }
    /*..........................................................................*/
    void QActive_stop(QActive *me) {
(18)    me->running = (uint8_t)0;                        /* stop the run() loop */
(19)    CloseHandle(me->osObject);                      /* cleanup the OS event */
    }
    /*..........................................................................*/
(20) static DWORD WINAPI thread_function(LPVOID arg) {     /* for CreateThread() */
        ((QActive *)arg)->running = (uint8_t)1; /* allow the thread loop to run */
(21)    while (((QActive *)arg)->running) {    /* QActive_stop() stops the loop */
(22)        QEvent const *e = QActive_get_((QActive *)arg);   /* wait for event */
(23)        QF_ACTIVE_DISPATCH_((QHsm *)arg, e);     /* dispatch to the AO's SM */
(24)        QF_gc(e);    /* check if the event is garbage, and collect it if so */
        }
(25)    QF_remove_((QActive *)arg);/* remove this object from any subscriptions */
        return 0;                                             /* return success */
    }
```

(1) The initialization of the framework consists of calling `InitializeCriticalSection()` Win32 API to initialize the global critical section object `QF_win32CritSect_`.

(2) Clearing the flag `l_running_` causes the exit from `QF_run()` and natural termination of the "ticker thread".

(3) `QF_run()` invokes `QF_onStartup()` callback.

(4) The priority of the "ticker thread" is raised to maximum, in order to provide timely clock tick with minimal jitter.

(5) `QF_tick()` handles all armed time events (timers) for the application. In the Win32 port, the `QF_tick()` is called from the task context. QP v4 allows calling `QF_tick()` from the task context.

(6) The "ticker thread" is throttled by the `Sleep()` Win32 API call. Typically, the hardware clock tick in Windows is 10ms (100 Hz), and the "ticker thread" uses this rate.

NOTE: You can adjust the argument of `Sleep()` by calling the function `QF_setTickRate()`. Please note, however, that the actual granularity of the delay depends on the hardware rate of the system clock-tick interrupt for a given Windows platform (typically 10ms on 80x86).

(7) After the ticker thread terminates (by clearing the `l_running` flag), the callback `QF_onCleanup()` is called to perform any application-level cleanup.

(8) Before exiting, the main thread deletes the global critical section object.

(9) The `QActive_start()` parameter `stkSto` must be `NULL` to avoid double-allocation of per-thread stack since Windows allocates the stack internally.

(10) The `stkSize` parameter must be specified since Windows uses the parameter to initially size the stack.

(11) The native QF event queue structure must be initialized.

(12) The Win32 event object is initialized with the `CreateEvent()` Win32 API.

(13) The active object priority is added to the QF so that the framework knows to manage the active object.

(14) The initial transition inside the active object's state machine is triggered. The macro `QF_ACTIVE_INIT_()` resolves to `QHsm_init()` when `QActive` is derived from `QHsm` and to `QFsm_init()` if its derived from `QFsm`.

(15) The Win32 API `CreateThead()` is used to create the private Win32 thread of the active object.

(16) The QF priority (numbered `1..QF_MAX_ACTIVE`) is mapped to the Windows priority. Note that the default priority is "`THREAD_PRIORITY_ABOVE_NORMAL`". The QF priorities 1..3 are mapped to the lowest Windows priority levels, while the two highest QF priorities (`QF_MAX_ACITVE-1`, `QF_MAX_ACTIVE`) are mapped to the highest Windows priority levels.

(17) The priority of the active object thread just created is set with the call to `SetThreadPrioritity()` Win32 API.

(18) The function `QActive_stop()` clears the active object's `running` flag. Clearing this flag causes exit from the active object event loop and a natural termination of the active object's thread.

(19) The Win32 event object (`osObject`) is closed and recycled with the Win32 API call `CloseHandle()`.

(20) The static function `thread_function()` is the active object's thread routine.

(21) The event loop continues as long as the "`running`" flag of the active object is set.

(22) The event loop blocks until an event is posted to the active object's event queue.

(23) The event is dispatched to the state machine of the active object. The macro `QF_ACTIVE_DISPATCH_()` resolves to `QHsm_dispatch()` if the base class for `QActive` is `QHsm` (default), or to `QFsm_dispatch()` if the base class is `QFsm`. The `dispatch()` function returns only after complete processing of the event, which constitutes the Run-To-Completion step.

(24) After processing, the QF garbage collector decrements the reference counter of the event and recycles the event if the reference counter drops to zero.

(25) The active object is removed from the framework, meaning that QF no longer manages this active object. The QF priority level occupied by this active object becomes available to another active object that can be started in the future.

6 The QS Port

As described in Chapter 11 of PSiCC2, porting QS consists of customizing files `qs_port.h` header file and implementing the QS callbacks for sending the trace data to the host.

6.1 The QS Port Header File

The QS header file is located in `<qp>\ports\80x86\win32\vc2008\qs_port.h`. The following shows the QS configuration used in the Windows ports:

Listing 4: qs_port.h header file for the QS port to Windows

```
#ifndef qs_port_h
#define qs_port_h

#define QS_TIME_SIZE            4
#define QS_OBJ_PTR_SIZE         4
#define QS_FUN_PTR_SIZE         4

/*****************************************************************************
* NOTE:
* QS might be used with or without other QP components, in which case
* the separate definitions of the macros Q_ROM, Q_ROM_VAR, Q_ROM_BYTE,
* QF_INT_KEY_TYPE, QF_INT_LOCK, and QF_INT_UNLOCK are needed. In this
* port QS is configured to be used with the other QP component, by
* simply including "qf_port.h" *before* "qs.h".
*/
#include "qf_port.h"                                    /* use QS with QF */
#include "qs.h"                 /* QS platform-independent public interface */

#endif                                                      /* qs_port_h  */
```

The QS port must use exactly the same interrupt locking policy as QF (if it is to be used with other QP components, as it is in this case). Please refer to Section 5.1 for more details on the QF critical section implementation.

6.2 The QS Callbacks for Sending the Trace Data over TCP/IP

The QS callbacks for sending the trace data over TCP/IP is located in the file `<qp>\examples\80x86\-win32\vc2008\dpp\bsp.c`. The following Listing shows the details relevant for the QS implementation.

Listing 5 Implementation of QS software trace in the dpp example.

```
    /*--------------------------------------------------------------------------*/
    #ifdef Q_SPY                                        /* define QS callbacks */

(1) uint8_t QS_onStartup(void const *arg) {
        static uint8_t qsBuf[1024];               /* 1K buffer for Quantum Spy */
        static WSADATA wsaData;
        char host[64];
        char const *src;
        char *dst;
        USHORT port = 6601;                                 /* default port */
        ULONG ioctl_opt = 1;
```

```
        struct sockaddr_in servAddr;
        struct hostent *server;

(2)     QS_initBuf(qsBuf, sizeof(qsBuf));

        /* initialize Windows sockets */
(3)     if (WSAStartup(MAKEWORD(2,0), &wsaData) == SOCKET_ERROR) {
            printf("Windows Sockets cannot be initialized.");
            return (uint8_t)0;
        }

        src = (char const *)arg;
        dst = host;
        while ((*src != '\0') && (*src != ':') && (dst < &host[sizeof(host)])) {
            *dst++ = *src++;
        }
        *dst = '\0';
        if (*src == ':') {
            port = (USHORT)strtoul(src + 1, NULL, 10);
        }

        l_sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);       /* TCP socket */
        if (l_sock == INVALID_SOCKET){
            printf("Socket cannot be created.\n"
                   "Windows socket error 0x%08X.",
                   WSAGetLastError());
            return (uint8_t)0;
        }

        server = gethostbyname(host);
        if (server == NULL) {
            printf("QSpy host name cannot be resolved.\n"
                   "Windows socket error 0x%08X.",
                   WSAGetLastError());
            return (uint8_t)0;
        }
        memset(&servAddr, 0, sizeof(servAddr));
        servAddr.sin_family = AF_INET;
        memcpy(&servAddr.sin_addr, server->h_addr, server->h_length);
        servAddr.sin_port = htons(port);
        if (connect(l_sock, (struct sockaddr *)&servAddr, sizeof(servAddr))
            == SOCKET_ERROR)
        {
            printf("Socket cannot be connected to the QSpy server.\n"
                   "Windows socket error 0x%08X.",
                   WSAGetLastError());
            QS_EXIT();
            return (uint8_t)0;
        }

        /* Set the socket to non-blocking mode. */
(4)     if (ioctlsocket(l_sock, FIONBIO, &ioctl_opt) == SOCKET_ERROR) {
            printf("Socket configuration failed.\n"
                   "Windows socket error 0x%08X.",
                   WSAGetLastError());
            QS_EXIT();
            return (uint8_t)0;
```

```
        }

        return (uint8_t)1;                                      /* success */
    }
    /*..........................................................................*/
(5) void QS_onCleanup(void) {
        if (l_sock != INVALID_SOCKET) {
            closesocket(l_sock);
        }
        WSACleanup();
    }
    /*..........................................................................*/
(6) void QS_onFlush(void) {
        uint16_t nBytes = 1000;
        uint8_t const *block;
        QF_INT_LOCK(dummy);
        while ((block = QS_getBlock(&nBytes)) != (uint8_t *)0) {
            QF_INT_UNLOCK(dummy); \
            send(l_sock, (char const *)block, nBytes, 0);
            nBytes = 1000;
            QF_INT_LOCK(dummy);
        }
        QF_INT_UNLOCK(dummy);
    }
    /*..........................................................................*/
(7) QSTimeCtr QS_onGetTime(void) {
        return (QSTimeCtr)clock();
    }
    #endif                                                      /* Q_SPY */
    /*--------------------------------------------------------------------------*/
    /*..........................................................................*/
(8) static DWORD WINAPI idleThread(LPVOID par) {/* signature for CreateThread() */
        (void)par;
        l_running = (uint8_t)1;
        while (l_running) {
            Sleep(10);                                  /* wait for a while */
            if (_kbhit()) {                             /* any key pressed? */
                if (_getch() == '\33') {          /* see if the ESC key pressed */
                    QF_publish(Q_NEW(QEvent, TERMINATE_SIG));
                }
            }
    #ifdef Q_SPY
            {
                uint16_t nBytes = 1024;
                uint8_t const *block;
                QF_INT_LOCK(dummy);
                block = QS_getBlock(&nBytes);
                QF_INT_UNLOCK(dummy);
                if (block != (uint8_t *)0) {
                    send(l_sock, (char const *)block, nBytes, 0);
                }
            }
    #endif
        }
        return 0;                                       /* return success */
    }
    /*..........................................................................*/
```

```
    void BSP_init(int argc, char *argv[]) {
        DWORD threadId;
        HANDLE hIdle;
        char const *hostAndPort = "localhost:6601";

        if (argc > 1) {                                    /* port specified? */
            hostAndPort = argv[1];
        }
(9)     if (!QS_INIT(hostAndPort)) {
            printf("\nUnable to open QSpy socket\n");
            exit(-1);
        }

(10)    hIdle = CreateThread(NULL, 1024, &idleThread, (void *)0, 0, &threadId);
        Q_ASSERT(hIdle != (HANDLE)0);                  /* thread must be created */
        SetThreadPriority(hIdle, THREAD_PRIORITY_IDLE);

        QF_setTickRate(BSP_TICKS_PER_SEC);         /* set the desired tick rate */

        printf("Dining Philosopher Problem example"
               "\nQEP %s\nQF  %s\n"
               "Press ESC to quit...\n",
               QEP_getVersion(),
               QF_getVersion());
    }
```

(1) The callback `QS_onStartup()` specifies how to initialize the QS software tracing facility. The function returns success (non-zero) or failure (zero).

(2) The `QS_onStartup()` callback must always call `QS_initBuf()` to initialize the QS trace buffer.

(3) The standard Windows-sockets initialization is followed by opening a client TCP/IP socket to the host:port specified in the argument `arg` to `QS_init()`. The `QS_init()` function returns failure (zero) if the opening of the client socket fails.

(4) The client socket is set to non-blocking mode. In this mode the driver will never block on sending data to the socket.

(5) The `QS_onCleanup()` callback specifies how to close and cleanup the QS connection.

(6) The `QS_onFlush()` callback also uses the "block-oriented" interface and calls `QS_getBlock()` service inside an explicit critical section.

(7) The `QS_getTime()` callback specifies how to obtain a time-stamp from the system. This implementation uses only the coarse-granularity (1ms) standard `clock()` API. You might want to improve the granularity of the time-stamp if your particular Windows implementation provides a better free-running timer/counter.

(8,10) This particular QS implementation uses a dedicated `idleThread()` to poll the keyboard for the ESC key (termination) and QS trace dumping.

(9) This QS software tracing is initialized from the `BSP_init()` function.

7 Console Applications

An example of a Win32 console application (application without a GUI) is provided in `<qp>\examples\80x86\win32\vc2008\dpp\`. This application is the Dining Philosophers Problem (DPP) example described in Chapter 9 of PSiCC as well as in the Application Note "Dining Philosophers Application" [QP AN-DPP 08]

7.1 main()

Listing 6 shows the `main.c` module of the DPP example located in `<qp>\examples\80x86\win32\vc2008\dpp\`.

Listing 6: main() for the console DPP application.

```
(1) #include "qp_port.h"
(2) #include "dpp.h"
(3) #include "bsp.h"

    static QEvent const *l_philoQueueSto[N_PHILO][N_PHILO];
    static QSubscrList   l_subscrSto[MAX_PUB_SIG];

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

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

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

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

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

(8)     QS_FILTER_ON(QS_ALL_RECORDS);
(9)     QS_FILTER_OFF(QS_QF_INT_LOCK);
        QS_FILTER_OFF(QS_QF_INT_UNLOCK);
        QS_FILTER_OFF(QS_QF_ISR_ENTRY);
        QS_FILTER_OFF(QS_QF_ISR_EXIT);
        QS_FILTER_OFF(QS_QF_TICK);
        QS_FILTER_OFF(QS_QK_SCHEDULE);
                                                  /* object dictionaries... */
(10)    QS_OBJ_DICTIONARY(l_smlPoolSto);
        QS_OBJ_DICTIONARY(l_tableQueueSto);
        QS_OBJ_DICTIONARY(l_philoQueueSto[0]);
        QS_OBJ_DICTIONARY(l_philoQueueSto[1]);
        QS_OBJ_DICTIONARY(l_philoQueueSto[2]);
        QS_OBJ_DICTIONARY(l_philoQueueSto[3]);
        QS_OBJ_DICTIONARY(l_philoQueueSto[4]);
```

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

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

         for (n = 0; n < N_PHILO; ++n) {          /* start the active objects... */
(13)         QActive_start(AO_Philo[n], (uint8_t)(n + 1),
                           l_philoQueueSto[n], Q_DIM(l_philoQueueSto[n]),
                           (void *)0, 1024, (QEvent *)0);         /* 1K of stack */
         }
(14)     QActive_start(AO_Table, (uint8_t)(N_PHILO + 1),
                       l_tableQueueSto, Q_DIM(l_tableQueueSto),
                       (void *)0, 1024, (QEvent *)0);             /* 1K of stack */
(15)     QF_run();                                        /* run the QF application */

(16)     return 0;
     }
```

(1-3) The module starts with including the platform-specific QF header file `qf_port.h` (see Section 5.1) followed by the `qassert.h` header and the application header file `dpp.h`.

(4-5) All active objects are instantiated by calling the "constructor" functions.

> **NOTE:** This step is unnecessary in the C++ version, because the C++ runtime invokes all static constructors before calling `main()`.

(6) The BSP is initialized with the call `BSP_init()` (see the next section).

(7) The framework is initialized with the call `QF_init()`.

(8-9) The QS software tracing filters are enabled and subsequently the specific trace records are disabled.

(10) The QS "dictionary records" are generated for objects known only locally in this translation unit.

(11) The publish-subscribe functionality is initialized with `QF_psInit()`, where you provide the memory for the "subscription lists" and the size of this list.

(12) The QF event pools are initialized with call(s) to `QF_poolInit()`. This particular application used only one event pool, but you can initialize up to 3 pools of different sizes.

(13) All `N_PHILO` Philosopher active objects are started. Note that the Win32 port does not need pre-allocated stack space (`(void *)0` passed as stack space pointer), but requires specifying the initial stack size.

(14) The `Table` active object is started. At this point all active object threads are created.

(15) The call to `QF_run()` transfers control to the framework, which starts the "clock tick" thread. In the Win32 port the `QF_run()` API actually returns back to main(), after the call to `QF_stop()`.

(16) The `main()` function returns terminating all the spawned active object threads. This exits the application.

7.2 Board Support Package (BSP)

The "Board Support Package" consists of initialization and platform-specific callbacks used in the DPP application to display the status of Dining Philosophers.

Listing 7: BSP for the console DPP application.

```
(1) void BSP_init(int argc, char *argv[]) {
        DWORD threadId;
        HANDLE hIdle;
        char const *hostAndPort = "localhost:6601";

        if (argc > 1) {                                    /* port specified? */
            hostAndPort = argv[1];
        }
(1)     if (!QS_INIT(hostAndPort)) {
            printf("\nUnable to open QSpy socket\n");
            exit(-1);
        }

(2)     hIdle = CreateThread(NULL, 1024, &idleThread, (void *)0, 0, &threadId);
        Q_ASSERT(hIdle != (HANDLE)0);                  /* thread must be created */
(3)     SetThreadPriority(hIdle, THREAD_PRIORITY_IDLE);


(4)     QF_setTickRate(BSP_TICKS_PER_SEC);         /* set the desired tick rate */

        printf("Dining Philosopher Problem example"
               "\nQEP %s\nQF  %s\n"
               "Press ESC to quit...\n",
               QEP_getVersion(),
               QF_getVersion());
    }
    /*..........................................................................*/
(5) static DWORD WINAPI idleThread(LPVOID par) {/* signature for CreateThread() */
        (void)par;
        l_running = (uint8_t)1;
(6)     while (l_running) {
(7)         Sleep(10);                                     /* wait for a while */
            if (_kbhit()) {                                /* any key pressed? */
                if (_getch() == '\33') {          /* see if the ESC key pressed */
(8)                 QF_publish(Q_NEW(QEvent, TERMINATE_SIG));
                }
             }
    #ifdef Q_SPY
            {
                uint16_t nBytes = 1024;
                uint8_t const *block;
                QF_INT_LOCK(dummy);
                block = QS_getBlock(&nBytes);
                QF_INT_UNLOCK(dummy);
                if (block != (uint8_t *)0) {
                    send(l_sock, (char const *)block, nBytes, 0);
                }
            }
    #endif
        }
        return 0;                                           /* return success */
    }
    /*..........................................................................*/
(9) void BSP_displyPhilStat(uint8_t n, char const *stat) {
        printf("Philosopher %2d is %s\n", (int)n, stat);
```

```
(10)      QS_BEGIN(PHILO_STAT, AO_Philo[n])  /* application-specific record begin */
              QS_U8(1, n);                                  /* Philosopher number */
              QS_STR(stat);                                 /* Philosopher status */
          QS_END()
      }
      /*..........................................................................*/
(11)  void Q_onAssert(char const Q_ROM * const Q_ROM_VAR file, int line) {
          fprintf(stderr, "Assertion failed in %s, line %d", file, line);
          QF_stop();
      }
```

(1) The Q-SPY software tracing instrumentation is initialized. This part of the code is active only when you build the Spy configuration in which the macro `Q_SPY` is defined.

(2-3) The `BSP_init()` function creates a low-priority "idle thread" that polls the keyboard for the ESC key to terminate the application. The "idle thread" also performs the Q-SPY output to the TCP/IP socket specified in the command line. (Again, the Q-SPY output is only performed in the Spy configuration.)

(4) The system clock tick rate is adjusted to deliver `BSP_TICKS_PER_SEC` clock ticks per second. Note that the actual system clock tick rate is a multiple of the hardware tick rate for the given board.

(7) The signature of the "idle thread" function conforms to the Win32 thread signature. The idle thread spins as long as `QF_running_` flag is set (this flag is cleared in `QF_stop()`). The thread is "throttled" by the `Sleep()` Win32 API call.

(8) The idle thread uses `QF_publish()` to publish the `TERMINATE` event to the application when it detects the ESC key press.

(9) The function `BSP_displayPhilStat()` is called from the DPP application to display the status of Philosopher '`n`'. Here the function simply prints to the screen.

(10) The function `BSP_displayPhilStat()` also demonstrates how to generate an application-specific trace record. In this case, the function outputs the philosopher number as a byte and the status as a string.

(11) The assertion-failure handler `Q_onAssert()`, which every QP application must define, prints the message to the screen and stops the application by calling `QF_stop()`.

8 GUI Applications with Plain Win32 API

Perhaps the trickiest part of integrating QP with any GUI system is reconciling the event-driven multitasking models used in QP and the GUI system. Since both the GUI system and QP are in fact event-driven frameworks, it is crucial to carefully avoid potential conflicts of authority (who's controlling CPU, events, event queuing, event processing, and so on).

8.1 General Structure of the QP Application with GUI

Generally, when QP is combined with a GUI system, the control should reside in QP, because QP operates at the higher level of abstraction than the GUI system. The GUI "callbacks", "message maps" or other mechanisms should generally be used only for generating QP events, but not for actual processing of the events. Event processing should occur in the active objects. Also, the most recommended design is to encapsulate all screen updates inside a dedicated active object, which will be henceforth called generically the "GUI-Manager". All other active objects in the application should not call GUI library to manipulate the screen directly, but rather should send events to the GUI-Manager to perform the screen updates. That way, the GUI-Manager can coordinate concurrent access to the screen and resolve potential conflicts among independent active objects.

> **NOTE:** It is also possible to perform output to the GUI from several active objects, because the Win32 graphical functions are reentrant from many Win32 threads ([Petzold 96]). However, this structure of an application is not recommended because concurrent output to the screen can easily lead to inconsistencies.

This section describes how to add GUI to the Dining Philosophers Problem (DPP) example, where the status of dining philosophers is shown in a dialog box. Additionally, the GUI version of the DPP example has been extended to demonstrate interaction with the application. The GUI for DPP example provides the "TERMINATE" button, which generates the TERMINATE_SIG event, which causes shutdown of the application.

The examples of GUI applications for the plain Win32 API are located in `<qp>\examples\80x86\win32\vc2008\dpp-gui\`. The structure of the GUI application is intentionally very close to the structure of the embedded QP applications. In particular, the application has the `main()` function, which runs in a separate thread of execution. The GUI subsystem, centered around the `WinMain()` function, is loosely coupled with the QP application via events. The code structure of the GUI subsystem has been chosen to closely correspond to the code generated by the Application Wizard for MFC (see the MFC application section.)

> **NOTE:** The `<qp>\examples\80x86\win32\vc2008\dpp-gui\` directory contains the header file "`afxres.h`", which is not provided in the standard distribution of the free Express Editions of Visual C++ 2005/2008. The "`afxres.h`" header file allows in this case compilation of the GUI resource file `dpp.rc`. To properly add the GUI functionality to the Express Editions of the Visual C++ toolsets you can download and install the separate "Microsoft Platform SDK", which will provide the standard header files and the MFC libraries. You will still need a resource editor, such as ResEdit (http://www.resedit.net/).

8.2 The GUI Subsystem (WinMain() and the Dialog Procedure)

The GUI subsystem is located in the `dpp_gui.c` module of the DPP example located in `<qp>\examples\80x86\win32\vc2008\dpp-gui`. Listing 8 shows the file.

Listing 8: The GUI subsystem of the dialog-based DPP-GUI application (file `dpp_gui.c`).

```
(1) #include "qp_port.h"
(2) #include "dpp.h"
(3) #include "bsp.h"
(4) #include "resource.h"

    . . .
    /*..........................................................................*/
(5) int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hPrevInst,
                       LPSTR cmdLine, int iCmdShow)
    {
        int nResponse;
        l_cmdLine = cmdLine;                          /* save the command line */

           /* create the modal dialog box that is the GUI of the appliction ... */
(6)     nResponse = DialogBox(hInst, MAKEINTRESOURCE(IDD_DIALOG),
                              NULL, &dppDlg);
        if (nResponse == IDOK)
        {
            /* TODO: Place code here to handle when the dialog is
            *  dismissed with OK */
        }
        else if (nResponse == IDCANCEL)
        {
            /* TODO: Place code here to handle when the dialog is
            *  dismissed with Cancel */
        }

(7)     return 0;
    }
    /*..........................................................................*/
(8) static BOOL CALLBACK dppDlg(HWND hWnd, UINT iMsg,
                         WPARAM wParam, LPARAM lParam)
    {
        switch (iMsg) {
            case WM_INITDIALOG: {
                BSP_hWnd = hWnd;                  /* save the main window hanlde */

                /* --> QP: spawn the thread to run the QF application... */
(9)             Q_ALLEGE(CreateThread(NULL, 0, &QF_app, l_cmdLine, 0, NULL)
                         != (HANDLE)0);
                break;
            }
            case WM_COMMAND: {
                switch (wParam) {
                    case IDOK:
                    case IDCANCEL: {
                        /* --> QP: translate all GUI events into QP events */
(10)                    QF_publish(Q_NEW(QEvent, TERMINATE_SIG));
                        EndDialog(hWnd, 0);
                        break;
                    }
                }
                break;
            }
        }
        return FALSE;
    }
```

```
    /*..........................................................................*/
(11) static DWORD __stdcall QF_app(LPVOID lpParameter) {
        extern int main(LPSTR cmdLine);
(12)    main((LPSTR)lpParameter);                    /* run the QF application */
        return 0;
    }
```

(1-3) The module starts with including the platform-specific QF header file `qp_port.h` followed by the application header file `dpp.h`, followed by the board support package (BSP) `bsp.h`.

(4) The GUI application includes the header file `resource.h` generated by the Visual Studio resource manager.

(5) The standard `WinMain()` function is the main entry point in the case of a GUI application.

(6) This particular application is based on a single dialog box. The `WinMain()` simply launches the modal dialog box via the `DialogBox()` Win 32 API. The `DialogBox()` function contains the message loop for the dialog box and returns only after the dialog box closes.

NOTE: The structure of `WinMain()` is very standard as for any other typical Windows GUI application. In particular, the `WinMain()` function contains the standard “message pump”.

(7) The return from `WinMain()` terminates the application (the process) and all Win32 threads spawned from this process.

The DPP-GUI application is based on a dialog box, so all GUI programming occurs in the dialog procedure.

NOTE: The GUI subsystem of a QP application could as well be based on a traditional window or multiple windows, not necessarily on a dialog box. The GUI programming would in that case occur in the Windows procedures of the windows comprising the GUI.

The job of the Window Procedure (WndProc) in the QP application is only to translate the Windows messages (such as mouse clicks, button presses, or menu commands) into QP events and post/publish these events to the active object(s). The actual drawing to the screen is in this design typically not performed by the WndProc at all, but rather is left to the active objects—preferably to the single active object called “GUI-Manager”.

The main point here is that the “GUI Manager” receives and processes the GUI events just like any other events. However, the “GUI Manager” active object is independent on the Windows “message pump”, so it can also receive other events from the rest of the application. In this way, GUI encapsulated inside an active object becomes a reusable, event-driven component of the application. All other active objects do not concern themselves with rendering the screen and thus are decoupled from any particular GUI implementation (Win32 or other). In the DPP-GUI example, the Table active object is the only one performing output to the GUI (through the `BSP_displayPhilStat()` function), so it plays here the role of the “GUI Manager”.

(8) The WndProc is in this case a dialog procedure, but could just as easily be any other window procedure.

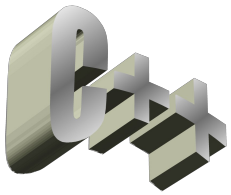
(9) The main WindProc of the application is responsible for spawning the Win32 thread that runs the QP application. The thread routine `QF_app()` calls the `main()` function, which is the same `main()` function as discussed for console-style application in Listing 6.

NOTE: Active objects can be started only after creating the main window. In particular the Table active object (the "GUI Manager" in this case) performs output to the GUI in the top-most initial transition. By this time the window handle must be ready.

(10) All GUI events are translated into QP events and are published to the active objects.

(11-12) The thread routine `QF_app()` calls the `main()` function, which is the same `main()` function as discussed for console-style application in Listing 6

9 GUI Applications with MFC



The C++ version of QP (QP/C++) can easily be integrated with Microsoft Foundation Classes (MFC) to provide a higher-level, more modern way of designing GUI applications for Windows.

The GUI example with MFC is based on the Dining Philosopher Problem (DPP) application described in Chapter 9 of PSiCC2 and Application Note [QP AN-DPP 08]. In this application, the status of dining philosophers is shown in a dialog box.

The examples of MFC application is located in `<qp>\examples\-80x86\win32\vc2008\dpp-mfc\`.

NOTE: The skeleton of the MFC application, including the function and file names, has been generated by the AppWizard from the Visual Studio IDE. The AppWizard has been configured to generate a dialog-based, statically-linked MFC application. Additionally, the Visual Studio ClassWizard has been used to generate message maps for the customized Windows messages and for the overridden methods of the GUI subclasses generated by the AppWizard.

9.1 The Application Class

Listing 9 shows the `dpp_mfc.cpp` module of the DPP example generated by the AppWizard. This file is located in `<qp>\examples\80x86\win32\vc2008\qdpp-mfc`. You don't need to modify this file at all to integrate the GUI with QP.

Listing 9: `dpp_mfc.cpp` module synthesized by the MFC AppWizard.

```
#include "stdafx.h"
#include "dpp_mfc.h"
#include "dpp_mfcDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CDpp

BEGIN_MESSAGE_MAP(CDpp, CWinApp)
    //{{AFX_MSG_MAP(CDpp)
        // NOTE - the ClassWizard will add and remove mapping macros here.
        //    DO NOT EDIT what you see in these blocks of generated code!
    //}}AFX_MSG
    ON_COMMAND(ID_HELP, CWinApp::OnHelp)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CDpp construction

CDpp::CDpp()
{
    // TODO: add construction code here,
```

```
    // Place all significant initialization in InitInstance
}

//////////////////////////////////////////////////////////////////////////
// The one and only CDpp object

CDpp theApp;

//////////////////////////////////////////////////////////////////////////
// CDpp initialization

BOOL CDpp::InitInstance()
{
    // Standard initialization
    // If you are not using these features and wish to reduce the size
    //  of your final executable, you should remove from the following
    //  the specific initialization routines you do not need.

#ifdef _AFXDLL
    Enable3dControls();            // Call this when using MFC in a shared DLL
#else
    Enable3dControlsStatic();    // Call this when linking to MFC statically
#endif

    CDppDlg dlg;
    m_pMainWnd = &dlg;
    int nResponse = dlg.DoModal();
    if (nResponse == IDOK)
    {
        // TODO: Place code here to handle when the dialog is
        //  dismissed with OK
    }
    else if (nResponse == IDCANCEL)
    {
        // TODO: Place code here to handle when the dialog is
        //  dismissed with Cancel
    }

    // Since the dialog has been closed, return FALSE so that we exit the
    //  application, rather than start the application's message pump.
    return FALSE;
}
```

9.2 The Window Class

Because the DPP-MFC application is dialog-based the main Window class is `CQdppDlg` (generated by the AppWizard).

The `CDppDlg` class derives indirectly from the MFC `CWindow` class, which encapsulates the Window Procedure (WndProc). The job of the WndProc in the QP application is only to translate the Windows messages (such as mouse clicks or button presses) into QF events and post/publish these GUI events to the active object(s). The actual drawing to the screen is in this design typically not performed by the WndProc (the Window class) at all, but rather is left to the active objects.

NOTE: As usual in QP applications, it is strongly recommended to encapsulate all graphical screen output in one dedicated active object (the “GUI Manager”). However, it is also possible to perform output to the GUI from

several active objects, because the Win32 graphical functions are reentrant from many Win32 threads ([Petzold 96]).

The main point here is that the "GUI Manager" receives and processes the GUI events just like any other events. However, the "GUI Manager" active object is independent on the Windows "message pump", so it can also receive other events from the rest of the application. In this way, GUI encapsulated inside an active object becomes a reusable, event-driven component of the application. The other active objects do not concern themselves with rendering the screen and thus are decoupled from any particular GUI implementation (Win32 or other).

In the DPP-MFC example, the Table active object is the only one performing output to the GUI (through the `BSP_displayPhilStat()` function), so it plays here the role of the "GUI Manager".

Listing 10: Customization of the CDppDlg class synthesized by the MFC AppWizard. The code in bold indicates the customizations

```
    // dpp_mfcDlg.cpp : implementation file
    //

    #include "stdafx.h"
    #include "dpp_mfc.h"
    #include "dpp_mfcDlg.h"

    // --> QP: header files
(1) #include "qp_port.h"
    #include "dpp.h"
    #include "bsp.h"
    #include "resource.h"
    Q_DEFINE_THIS_FILE

    #ifdef _DEBUG
    #define new DEBUG_NEW
    #undef THIS_FILE
    static char THIS_FILE[] = __FILE__;
    #endif

    /////////////////////////////////////////////////////////////////////////////
    // CDppDlg dialog

(2) CDppDlg::CDppDlg(CWnd* pParent /*=NULL*/)
        : CDialog(CDppDlg::IDD, pParent)
    {
        //{{AFX_DATA_INIT(CDppDlg)
            // NOTE: the ClassWizard will add member initialization here
        //}}AFX_DATA_INIT
        // Note that LoadIcon does not require a subsequent DestroyIcon in Win32
        m_hIcon = AfxGetApp()->LoadIcon(IDI_QL);
    }

    void CDppDlg::DoDataExchange(CDataExchange* pDX)
    {
        CDialog::DoDataExchange(pDX);
        //{{AFX_DATA_MAP(CDppDlg)
            // NOTE: the ClassWizard will add DDX and DDV calls here
        //}}AFX_DATA_MAP
    }
```

```
    BEGIN_MESSAGE_MAP(CDppDlg, CDialog)
        //{{AFX_MSG_MAP(CDppDlg)
        ON_WM_PAINT()
        ON_WM_QUERYDRAGICON()
(3)     ON_BN_CLICKED(IDC_TERMINATE, OnTerminate)
(4)     ON_WM_CLOSE()
        //}}AFX_MSG_MAP
    END_MESSAGE_MAP()

    /////////////////////////////////////////////////////////////////////////////
    // CDppDlg message handlers


    static DWORD __stdcall QF_app(LPVOID lpParameter);

(5) BOOL CDppDlg::OnInitDialog()
    {
        CDialog::OnInitDialog();

        // Set the icon for this dialog.  The framework does this automatically
        //  when the application's main window is not a dialog
        SetIcon(m_hIcon, TRUE);            // Set big icon
        SetIcon(m_hIcon, FALSE);        // Set small icon

        // TODO: Add extra initialization here

        // --> QP: spawn the thread to run the QF application...
(6)     Q_ALLEGE(CreateThread(NULL, 0, &QF_app, theApp.m_lpCmdLine, 0, NULL)
                 != (HANDLE)0);

        return TRUE;  // return TRUE  unless you set the focus to a control
    }

(7) static DWORD __stdcall QF_app(LPVOID lpParameter) {
        extern int main(LPSTR cmdLine);
(8)     main((LPSTR)lpParameter);                       // run the QF application
        return 0;
    }


    // If you add a minimize button to your dialog, you will need the code below
    //  to draw the icon.  For MFC applications using the document/view model,
    //  this is automatically done for you by the framework.

    void CDppDlg::OnPaint()
    {
        if (IsIconic())
        {
            CPaintDC dc(this); // device context for painting

            SendMessage(WM_ICONERASEBKGND, (WPARAM) dc.GetSafeHdc(), 0);

            // Center icon in client rectangle
            int cxIcon = GetSystemMetrics(SM_CXICON);
            int cyIcon = GetSystemMetrics(SM_CYICON);
            CRect rect;
```

```
            GetClientRect(&rect);
            int x = (rect.Width() - cxIcon + 1) / 2;
            int y = (rect.Height() - cyIcon + 1) / 2;

            // Draw the icon
            dc.DrawIcon(x, y, m_hIcon);
        }
        else
        {
            CDialog::OnPaint();
        }
    }

    // The system calls this to obtain the cursor to display while the user drags
    //  the minimized window.
    HCURSOR CDppDlg::OnQueryDragIcon()
    {
        return (HCURSOR) m_hIcon;
    }

    void CDppDlg::OnTerminate()
    {
        // TODO: Add your control notification handler code here
(9)     QF::publish(Q_NEW(QEvent, TERMINATE_SIG));
    }

    void CDppDlg::OnClose()
    {
        // TODO: Add your control notification handler code here
(10)    QF::publish(Q_NEW(QEvent, TERMINATE_SIG));

        CDialog::OnClose();
    }
```

(1) You need to include the QP header files as shown.

(2) The `CDppDlg` class is generated by the AppWizard. Here, in the constructor of this class the application loads the custom icon (`IDI_QL` in this case).

(3) The message map entry `ON_BN_CLICKED(IDC_TERMINATE, OnTerminate)` is generated by the ClassWizard to customize the handling of the "Terminate" button click.

(4) The message map entry `ON_WM_CLOSE()` is generated by the ClassWizard to customize the handling of the "X" button click.

(5) The `onInitDialog()` method is generated by the ClassWizard to customize the creation of the dialog box.

(6) The `onInitDialog()` method spawns a separate Win32 thread to execute the QP application. The thread routine `QF_app()` calls the `main()` function, which is the same `main()` function as discussed for console-style application in Listing 6

NOTE: Active objects can be started only after creating the main window. In particular the Table active object (the "GUI Manager" in this case) performs output to the GUI in the top-most initial transition. By this time the window handle must be ready.

(7-8) The thread routine `QF_app()` calls the `main()` function, which is the same `main()` function as discussed for console-style application in Listing 6.

(9-10) The `OnTerminate()` and `onClose()` methods are created by the ClassWizard. These functions produce the events for the QP application, but do not process the events directly.

10 Windows CE-Specific Considerations

The WinCE port of QP is exactly identical to the desktop Windows port. The only differences between WinCE show up in the applications, because WinCE uses only Unicode strings.

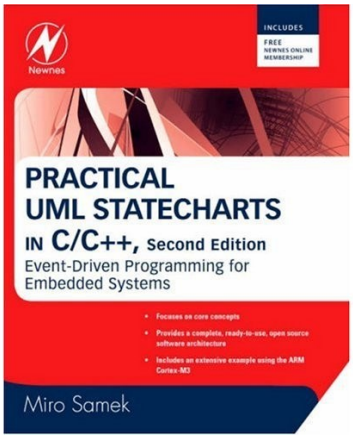
11 References

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

12 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