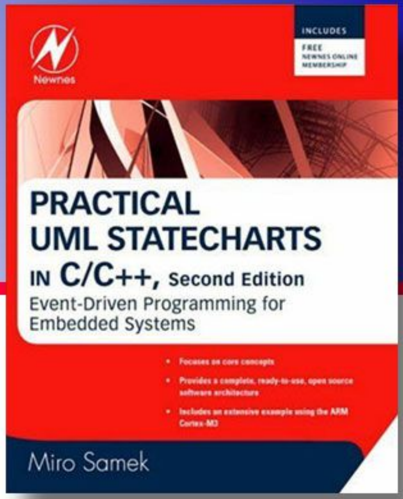
quantum™ LeaPs
innovating embedded systems


PRACTICAL
UML STATECHARTS
IN C/C++, Second Edition
Event-Driven Programming for Embedded Systems

Miro Samek

Application Note
QP™ and Win32 (Windows)

Document Revision J
August 2013

Table of Contents

1 Introduction

This Application Note describes how to use QP™ state machine frameworks version **5.x.x** or higher with the Microsoft Windows operating system including desktop Windows (e.g., **Windows 7**) 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.

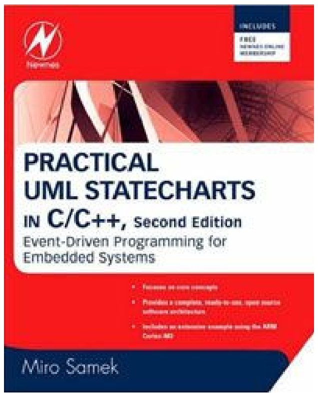
Integrating QP with the Win32 GUI API is interesting for at least two reasons. First, you might use QP to build highly modular, well structured, multithreaded Windows applications based on the concept of active objects (a.k.a. Actors) and hierarchical state machines. In this use case, QP complements Windows by providing the high-level structure, while Windows API renders the GUI and provides various services. Also, the QP port to Windows enables developers to build 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 using QP on Windows is **rapid prototyping** (virtual prototyping), simulation, and testing of deeply embedded software on the desktop, including building realistic user interfaces consisting of buttons, LEDs, and LCD displays (both segmented and graphic). Moving embedded software development from an embedded target to the desktop eliminates the target system bottleneck and dramatically shortens the development time while improving the quality of the software. The Windows-based desktop systems often make excellent platforms to **develop**, **test**, and **debug** embedded applications. This Application Note has been specifically designed to provide a **complete toolkit** with all the needed graphical components tested with the powerful **free** tools (such as the free Visual C++ Express Edition and the free ResEdit resource editor).

All these options get especially attractive if you consider using the **QM™** modeling tool for designing QP applications graphically and generating code **automatically**.

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

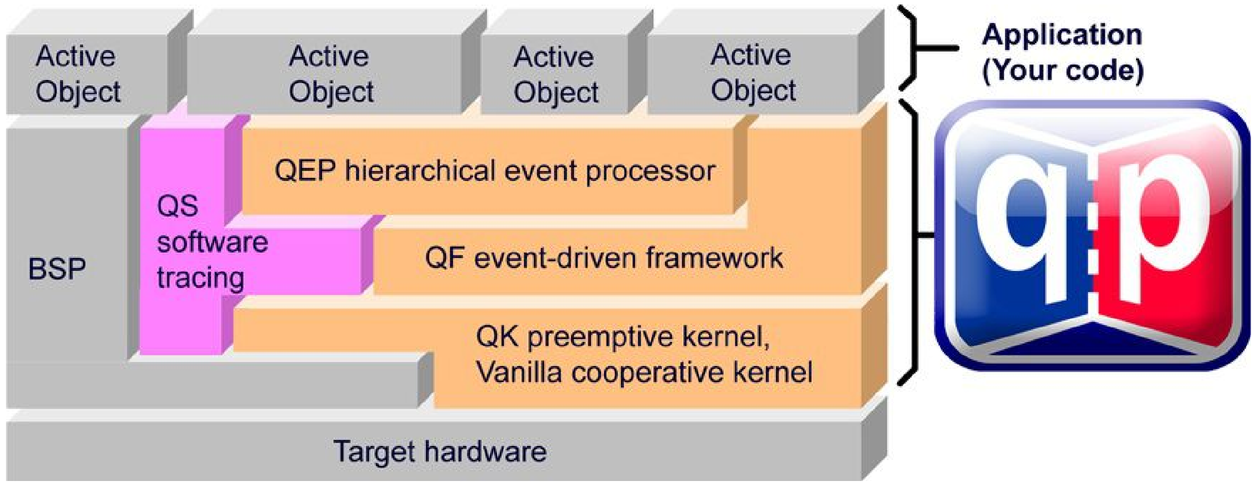
1.1 About QP™


PRACTICAL UML STATECHARTS IN C/C++, Second Edition
Event-Driven Programming for Embedded Systems
Miro Samek

QP™ is a family of very lightweight, open source, active object frameworks. QP enable software developers to build well-structured applications as systems of concurrently executing hierarchical state machines (UML statecharts) directly in C or C++, or by means of the QM™ graphical UML modeling tool. 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



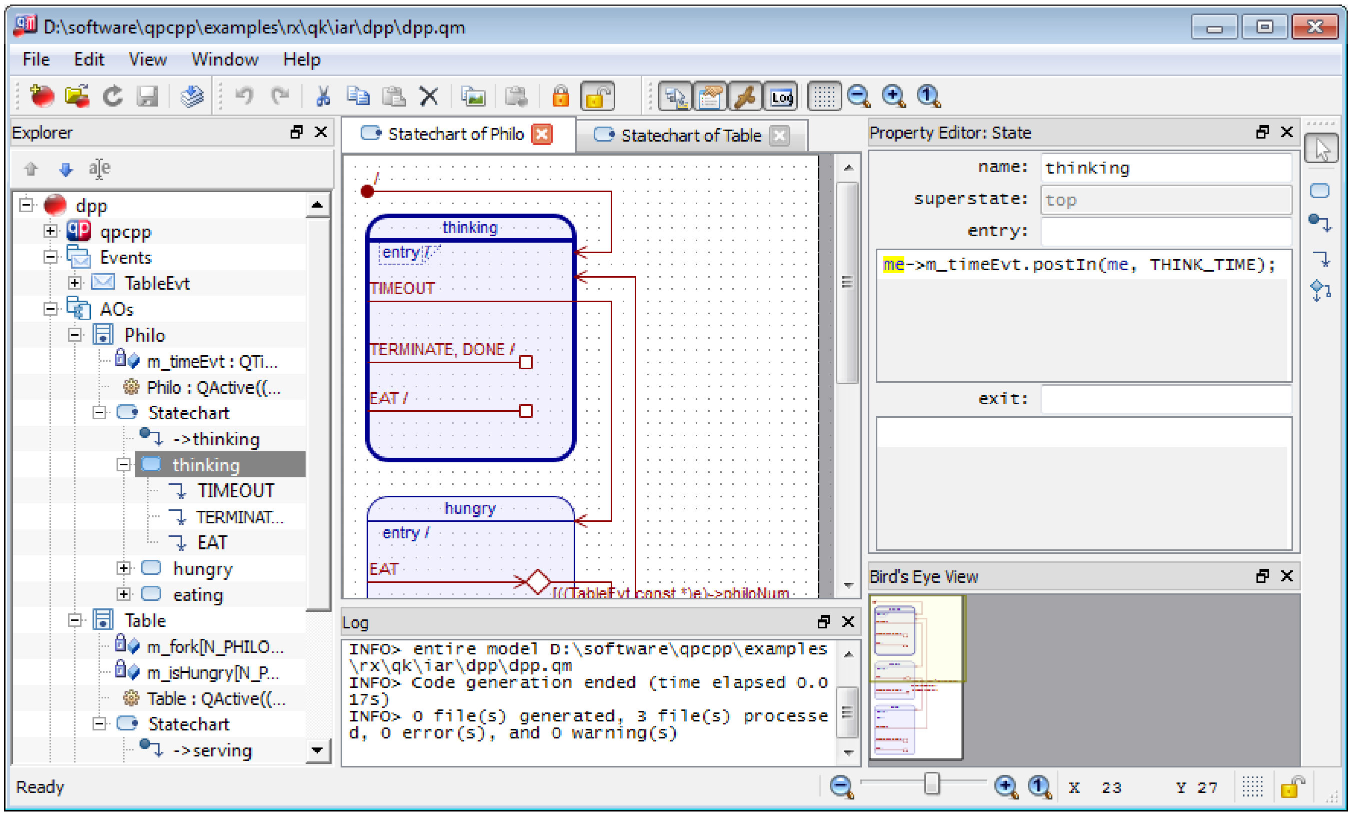
1.2 About QM™

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

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

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

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



```
name:       thinking
superstate: top
entry:
me->m_timeEvt.postIn(me, THINK_TIME);
exit:
```

```
INFO> entire model D:\software\qpcpp\examples
\rx\qk\iar\dpp\dpp.qm
INFO> Code generation ended (time elapsed 0.0
17s)
INFO> 0 file(s) generated, 3 file(s) processe
d, 0 error(s), and 0 warning(s)
```

1.3 Key Features of this QP on Windows

1. The QP™ supports 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. The QP port to Win32 does not force the application-level code to include the extensive `<windows.h>` header file. All the dependency on the Win32 API is encapsulated inside the port implementation files. (NOTE:Obviously, parts of the code, such as BSP that depend on Win32 API need to include `<windows.h>` and perhaps other Microsoft include files.)
4. The provided code contains all graphical components for efficient pixel-addressable graphical displays (LCDs, OLED, etc.), segmented displays, owner-drawn buttons, and LEDs.
5. The provided implementation is based on the raw Win32 API for widest potability and to enable the developers to use free tools, such as Microsoft Visual C++ Express Edition and the ResEdit resource editor.

1.4 About the Tools

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

- Microsoft Visual C++ Express 2012 for console and GUI applications (requires "Platform SDK")

NOTE: The Visual C++ Express editions are pre-configured for development with the .NET framework. To enable development with the raw Win32 API, you need to search the Microsoft website for "Express Edition Platform SDK" and install the software to your machine. The search of the Microsoft website will lead you also to the step-by-step installation instructions.

- Free ResEdit resource editor (www.resedit.net/) for editing GUI resources

NOTE: The Visual C++ Express editions do not include the resource editor and even the resource editor included in the Professional Visual C++ editions is inferior to ResEdit.

- MinGW (currently only for console applications).

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

1.6 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 Getting Started

This section describes how to install, execute, build, and debug QP applications on Windows.

NOTE: This application note pertains both to QP/C and QP/C++. Most of the code listings in this document refer to the C version. The QP/C++ code is shown only when the differences from C become non-trivial and significant.

2.1 Installing QP

You need to download and install the Generally Available distribution of the QP code (QP/C or QP/C++) from www.state-machine.com/downloads. QP is distributed in a ZIP archive (e.g., `qpc_5.0.0.zip` for the QP/C version, or `qpcpp_5.0.0.zip` for the QP/C++ version). You can uncompress the QP archive(s) into any location on your hard drive, but you need to define an environment variable `QPC` to point to the QP installation directory. Also, to take advantage of the QSPY software tracing, which is part of the Debug build configuration of the provided examples, you need to download and install the Qtools collection and you need to define an environment variable `QTOOLS` to point to the installation directory. The following table summarizes the components you need to install and the environment variables you need to define:

Software component	Environment Variable	Example
QP/C framework	`QPC`	`C:\qp\qpc`
QP/C++ framework	`QPCPP`	`C:\qp\qpcpp`
QP-nano framework	`QPN`	`C:\qp\qpn`
Qtools collection	`QTOOLS`	`C:\tools\qtools`

The following Listing 1 shows selected directories and files after installing the QP baseline code.

Listing 1: Directories and files pertaining to the QP-Win32 port.

```
<qp>\                      - QP-root directory for Quantum Platform (QP/C or QP/C++)
  +-include\               - QP public include files
  | +-qassert.h            - Assertions platform-independent public include
  | +-qevt.h               - QEvt declaration
  | +-. . .
  |
  +-ports\                 - QP ports
  | +-win32\               - ports to Win32
  | | +-vc\                - Visual C++ toolset
  | | | +-Debug\           - Debug build
  | | | | +-qp.lib         - QP library
  | | | +-Release\         - Release build
  | | | +-Spy\             - Spy build
  | | | +-qp.sln           - Visual Studio solution file to build the QP libraries
  | | | +-qp.vcxproj       - Visual Studio project file to build the QP libraries
  | | | +-qep_port.h       - QEP platform-dependent public include
  | | | +-qf_port.h        - QF platform-dependent public include
  | | | +-qf_port.c        - QF platform-dependent implementation
  | | | +-qs_port.h        - QS platform-dependent public include
  | | | +-win32_gui.h      - Win32 GUI facilities for building embedded front panels
  | | | +-win32_gui.c      - Win32 GUI facilities for building embedded front panels
  | | |
| | +-mingw\            - MinGW compiler (GNU on Windows)
| | | +-debug\          - Debug build
| | | | +-libqp.a       - QP library
| | | +-release\        - Release build
| | | +-spy\            - Spy build
| | | +-Makefile        - Makefile to build the QP libraries
| | | +-qep_port.h      - QEP platform-dependent public include
| | | +-qf_port.h       - QF platform-dependent public include
| | | +-qf_port.c       - QF platform-dependent implementation
| | | +-qs_port.h       - QS platform-dependent public include
|
+-examples\             - QP examples
| +-win32\              - examples for Win32
| | +-vc\               - Visual C++ toolset
| | | +-dpp\            - Dining Philosophers Problem example (console)
| | | +-dpp-gui\        - Dining Philosophers Problem example (GUI version)
| | | | +-Debug\        - directory containing the Debug build
| | | | +-Release\      - directory containing the Release build
| | | | +-Spy\          - directory containing the Spy build
| | | | +-Res\          - directory containing the GUI Resources (bitmaps, etc.)
| | | | +-qm_code\      - directory containing the code generated by QM
| | | | | +-dpp.h       - the DPP header file
| | | | | +-phio.c      - the Philosopher active objects
| | | | | +-table.c     - the Table active object
| | | | +-dpp.qm        - QM model file for the DPP example
| | | | +-dpp_gui.sln   - Visual Studio solution to build the DPP-GUI example
| | | | +-dpp_gui.vcxproj – Visual Studio project to build the DPP-GUI example
| | | | +-bsp.h         - Board Support Package header file for DPP-GUI
| | | | +-bsp.c         - Board Support Package implementation for DPP-GUI
| | | | +-main.c        - the main function for DPP-GUI
| | | | +-dpp_gui.rc    - Resource file for the DPP-GUI application
| | | | +-resource.h    - Resource header file updated by ResEdit
| | | |
| | | +-game-gui\       - "Fly 'n' Shoot" game example for the EK-LM3S811 board
| | | | +-Debug\        - directory containing the Debug build
| | | | +-Release\      - directory containing the Release build
| | | | +-Spy\          - directory containing the Spy build
| | | | +-qm_code\      - directory containing the code generated by QM
| | | | | +-game.h      - the game header file
| | | | | +-missile.c   - the Missile active objects
| | | | | +-ship.c      - the Ship active object
| | | | | +-tunnel.c    - the Tunnel active object
| | | | +-game.qm       - QM model file for the game example
| | | | +-game_gui.sln  - Visual Studio solution to build the game-GUI example
| | | | +-game_gui.vcxproj – Visual Studio project to build the game-GUI example
| | | | +-bsp.h         - Board Support Package header file for the game
| | | | +-bsp.c         - Board Support Package implementation for the game
| | | | +-gui.h         - Qt GUI header header file for the game
| | | | +-main.c        - the main function for the game
| | | | +-game-gui.rc   - Resource file for the game-GUI application
| | | | +-resource.h    - Resource header file updated by ResEdit
```

2.2 Building the QP Libraries

All QP components are deployed as libraries that you statically link to your application. The pre-built QP libraries are provided inside the `<qp>\ports\win32\` directory. Normally, you should have no need to re-build the QP 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.

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

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

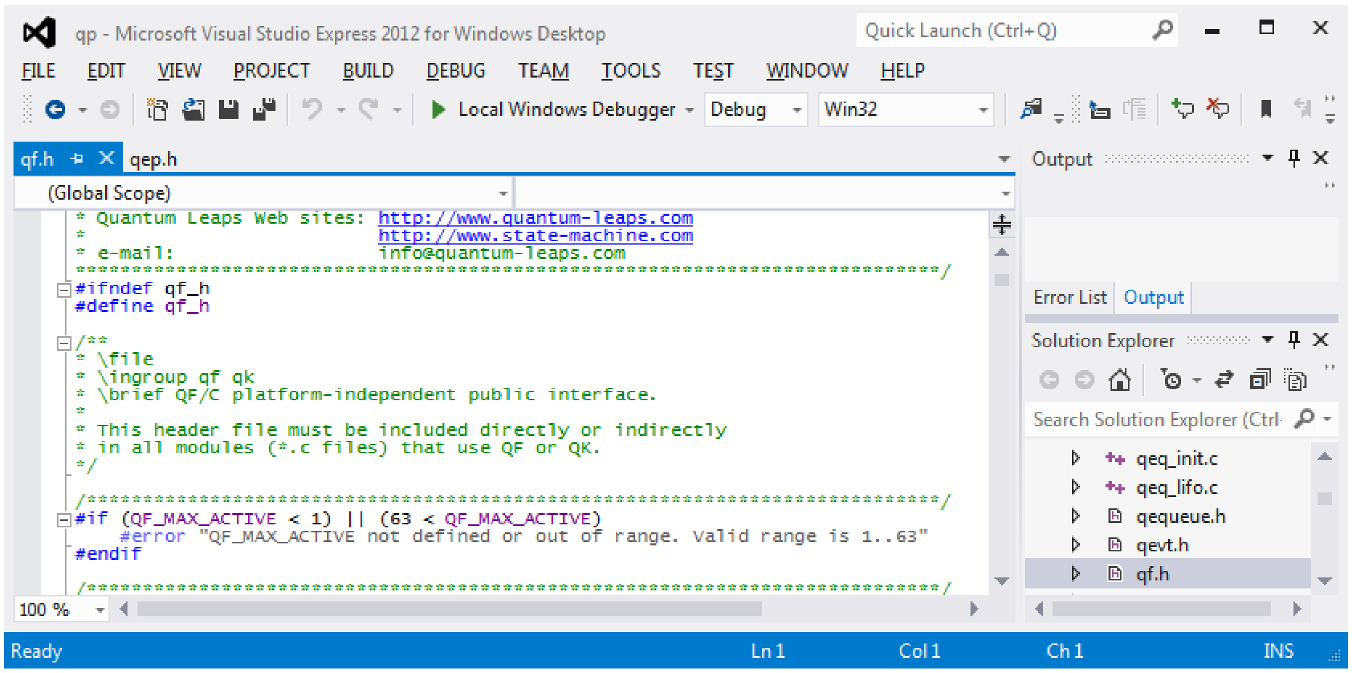
Release - this configuration is built with no debugging information and high optimization. Single-stepping and debugging is effectively impossible due to the lack of debugging information and optimized code, but the debugger can be used to download and start the executable. When the QP framework finds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

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

2.2.1 Building QP libraries with Visual Studio Express

The build process should produce the QP library in the location: `<qp>\ports\win32\vc\Debug\qp.lib`. To build the Release and Spy configurations, please select the desired configuration in the Visual Studio IDE and repeat the build for each configuration.

Figure 3: Building QP libraries in Visual C++ Express 2012



```
 * Quantum Leaps Web sites: http://www.quantum-leaps.com
 *                          http://www.state-machine.com
 * e-mail:                  info@quantum-leaps.com
 ****************************************************************************/
#ifndef qf_h
#define qf_h

/**
 * \file
 * \ingroup qf qk
 * \brief QF/C platform-independent public interface.
 *
 * This header file must be included directly or indirectly
 * in all modules (*.c files) that use QF or QK.
 */

/****************************************************************************/
#if (QF_MAX_ACTIVE < 1) || (63 < QF_MAX_ACTIVE)
    #error "QF_MAX_ACTIVE not defined or out of range. Valid range is 1..63"
#endif

/****************************************************************************/
```

2.2.2 *Building QP libraries with MinGW*

For the MinGW port, you perform a console build with the provided Makefile in `<qp>\ports\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 configurations

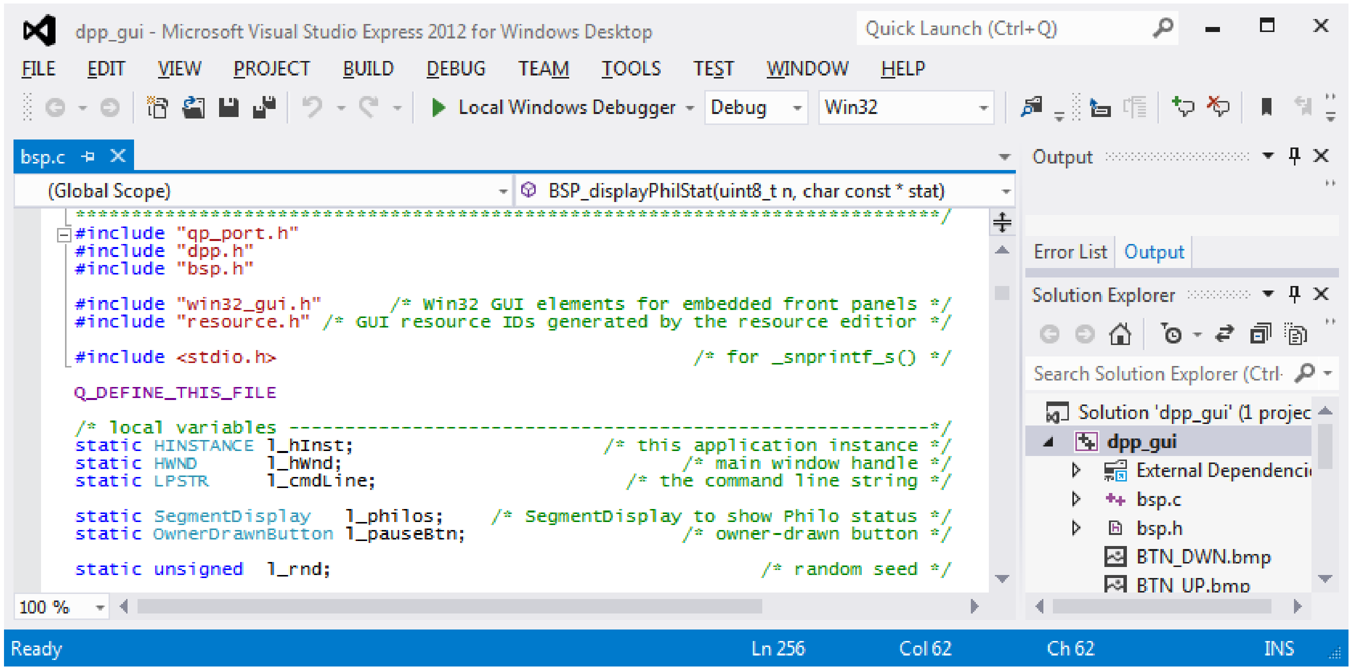
Software Version	Build command
Debug (default)	`make`
Release	`make CONF=rel`
Spy	`make CONF=spy`

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

2.3 Building the Examples

The QP examples directory (`<qp>/examples/win32/vc/`, see Listing 1) contains the Visual Studio solution files for building the examples.

Figure 4: The DPP-GUI example in the Visual C++ Express 2012


dpp_gui - Microsoft Visual Studio Express 2012 for Windows Desktop
Quick Launch (Ctrl+Q)
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST WINDOW HELP
Local Windows Debugger
Debug
Win32
bsp.c
(Global Scope)
BSP_displayPhilStat(uint8_t n, char const * stat)
Output
Error List
Output
Solution Explorer
Search Solution Explorer (Ctrl-
Solution 'dpp_gui' (1 projec
dpp_gui
External Dependenci
bsp.c
bsp.h
BTN_DWN.bmp
BTN_UP.bmp
100 %
Ready
Ln 256
Col 62
Ch 62
INS

```c
*************************************************************************/
#include "qp_port.h"
#include "dpp.h"
#include "bsp.h"

#include "win32_gui.h"      /* Win32 GUI elements for embedded front panels */
#include "resource.h" /* GUI resource IDs generated by the resource edition */

#include <stdio.h>                                      /* for _snprintf_s() */

Q_DEFINE_THIS_FILE

/* local variables ---------------------------------------------------------*/
static HINSTANCE l_hInst;                     /* this application instance */
static HWND      l_hWnd;                             /* main window handle */
static LPSTR     l_cmdLine;                     /* the command line string */

static SegmentDisplay   l_philos;    /* SegmentDisplay to show Philo status */
static OwnerDrawnButton l_pauseBtn;                    /* owner-drawn button */

static unsigned  l_rnd;                                      /* random seed */
```

For example, to build the DPP-GUI example, you load the project `<qp>/examples/win32/vc/dpp-gui/dpp_gui.sln` into the Visual Studio IDE and start the build. Similarly, to build the "Fly 'n' Shoot game example, you use the project `<qp>/examples/win32/vc/game-gui/game_gui.sln`.

NOTE: The **Spy** build configuration demonstrates using the direct QSPY software tracing output to the Visual Studio debug console and requires the **Qtools** collection to be installed and the `QTOOLS` environment variable to be defined.

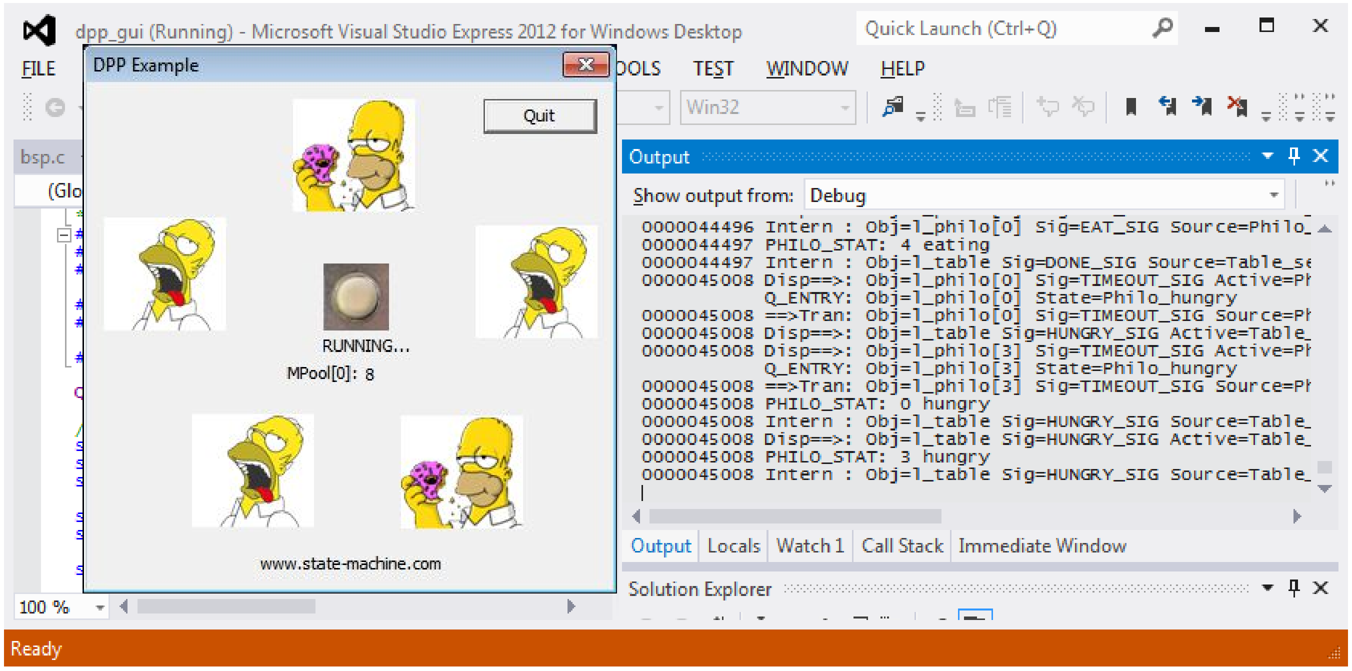
2.4 Running the DPP-GUI Example

The Dining Philosophers Problem (DPP) example demonstrates multiple active objects collaborating with each other. Each dining philosopher is represented as an active object (`Philo`) and there is additional active object `Table` for coordinating the shared resources (forks).

NOTE: The design and implementation of the Dining Philosopher Problem application, including state machines, is described in the Application Note “Dining Philosopher Problem” (see [AN-DPP]).

The DPP example application outputs the QSPY software trace data directly to the Qt console (see Figure 5). To demonstrate input to the application, the basic DPP example has been extended with the ability to pause. When the central button is depressed (but not released) the application enters the “paused” state, in which the Table stops granting permissions to eat to the Philosophers. This causes Philosophers to transition to the “hungry” state. After releasing the SERVING button, the application resumes normal execution. The application is terminated either by clicking the Close button.

Figure 5: The DPP example (Spy configuration) running in Visual Studio Express. Note the human-readable QSPY output in the Application Output window.



```
0000044496 Intern : Obj=l_philo[0] Sig=EAT_SIG Source=Philo_
0000044497 PHILO_STAT: 4 eating
0000044497 Intern : Obj=l_table Sig=DONE_SIG Source=Table_se
0000045008 Disp==>: Obj=l_philo[0] Sig=TIMEOUT_SIG Active=Ph
           Q_ENTRY: Obj=l_philo[0] State=Philo_hungry
0000045008 ==>Tran: Obj=l_philo[0] Sig=TIMEOUT_SIG Source=Ph
0000045008 Disp==>: Obj=l_table Sig=HUNGRY_SIG Active=Table_
0000045008 Disp==>: Obj=l_philo[3] Sig=TIMEOUT_SIG Active=Ph
           Q_ENTRY: Obj=l_philo[3] State=Philo_hungry
0000045008 ==>Tran: Obj=l_philo[3] Sig=TIMEOUT_SIG Source=Ph
0000045008 PHILO_STAT: 0 hungry
0000045008 Intern : Obj=l_table Sig=HUNGRY_SIG Source=Table_
0000045008 Disp==>: Obj=l_table Sig=HUNGRY_SIG Active=Table_
0000045008 PHILO_STAT: 3 hungry
0000045008 Intern : Obj=l_table Sig=HUNGRY_SIG Source=Table_
```

2.5 Running the “Fly 'n' Shoot” Game Simulation

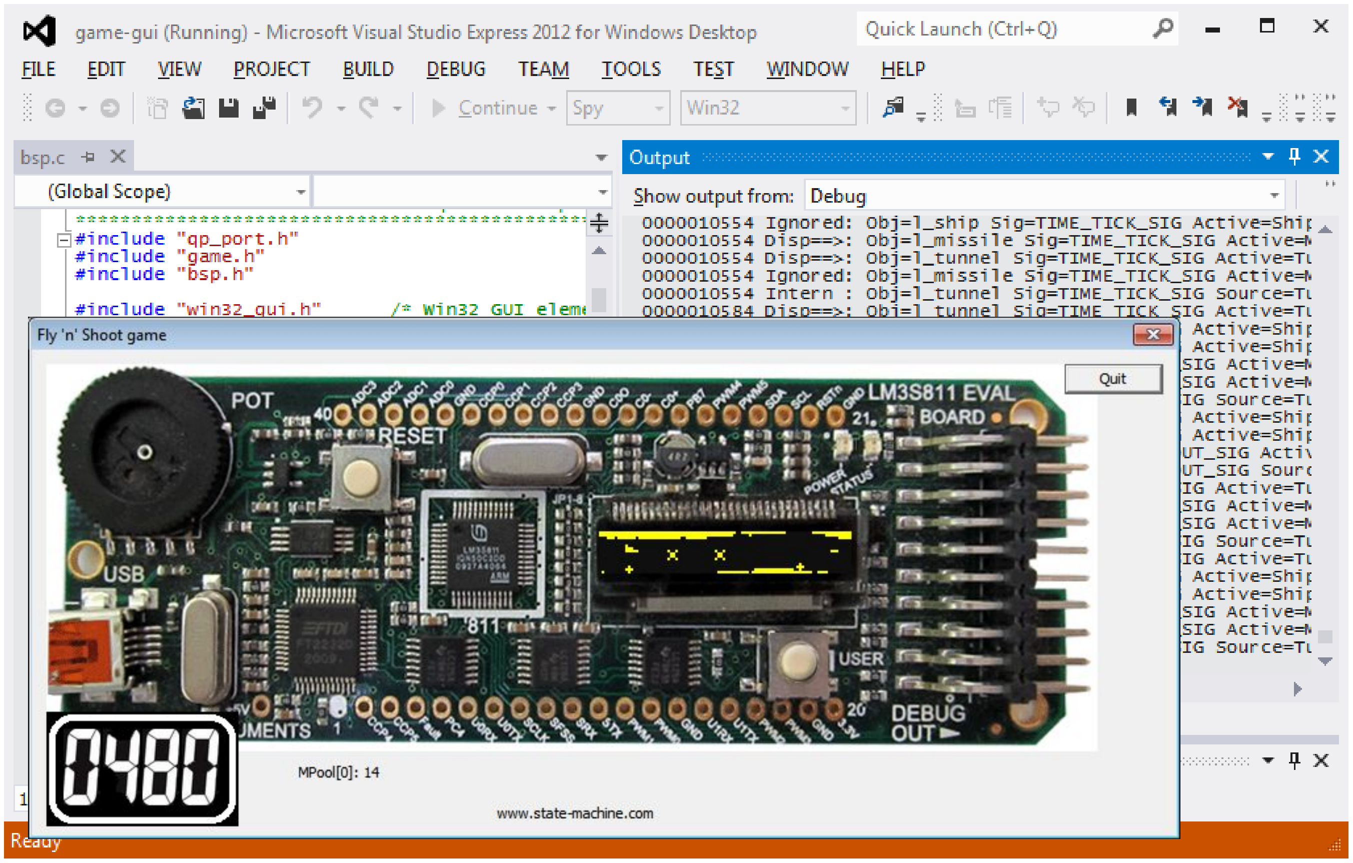
The “Fly 'n' Shoot” game example demonstrates simulating embedded target (the EK-LM3S811 board in this case) on Windows.

NOTE: The design and implementation of the “Fly 'n' Shoot” game application, including state machines, is described in the Chapter 1 of the “Practical UML Statecharts in C/C++, 2Ed” [PSiCC2].

The game simulation on the EK-LM3S811 board can be easily adapted for any other board or embedded device. The example demonstrates how to simulate a graphic display and a button for generating events as well as additional output (Score) not available in the real target (see Figure 5). The most interesting aspect of this simulation is that it runs exactly the same code as the real embedded board, except the BSP, which is implemented for the Windows GUI.

Once you launch the game simulation, it starts with flashing the “Press Button” text on the OLED display (just like on the real board). You start playing the game by clicking your mouse on the USER button. At this point the game transitions to the playing mode. You scroll the mouse wheel to move the ship icon up and down. You click on the USER button to fire the missile. You score points for surviving and shooting the mines with the missile. Just like the real thing, the game has a screen saver mode, which activates after several seconds in the demo mode. When the screen saver is active, the STATUS LED lights up.

Figure 6: The“Fly 'n' Shoot” game launched from Visual Studio 2012.



```
**************************************************
#include "qp_port.h"
#include "game.h"
#include "bsp.h"

#include "win32_gui.h"          /* Win32 GUI eleme
```

```
0000010554 Ignored: Obj=l_ship Sig=TIME_TICK_SIG Active=Shi
0000010554 Disp==>: Obj=l_missile Sig=TIME_TICK_SIG Active=M
0000010554 Disp==>: Obj=l_tunnel Sig=TIME_TICK_SIG Active=Tu
0000010554 Ignored: Obj=l_missile Sig=TIME_TICK_SIG Active=M
0000010554 Intern : Obj=l_tunnel Sig=TIME_TICK_SIG Source=Tu
0000010584 Disp==>: Obj=l_tunnel Sig=TIME_TICK_SIG Active=Tu
 Active=Shi
 Active=Shi
SIG Active=M
SIG Active=M
IG Source=Tu
 Active=Shi
 Active=Shi
UT_SIG Acti
UT_SIG Sourc
IG Active=Tu
SIG Active=M
SIG Active=M
IG Source=Tu
IG Active=Tu
 Active=Shi
 Active=Shi
SIG Active=M
SIG Active=M
IG Source=Tu
```

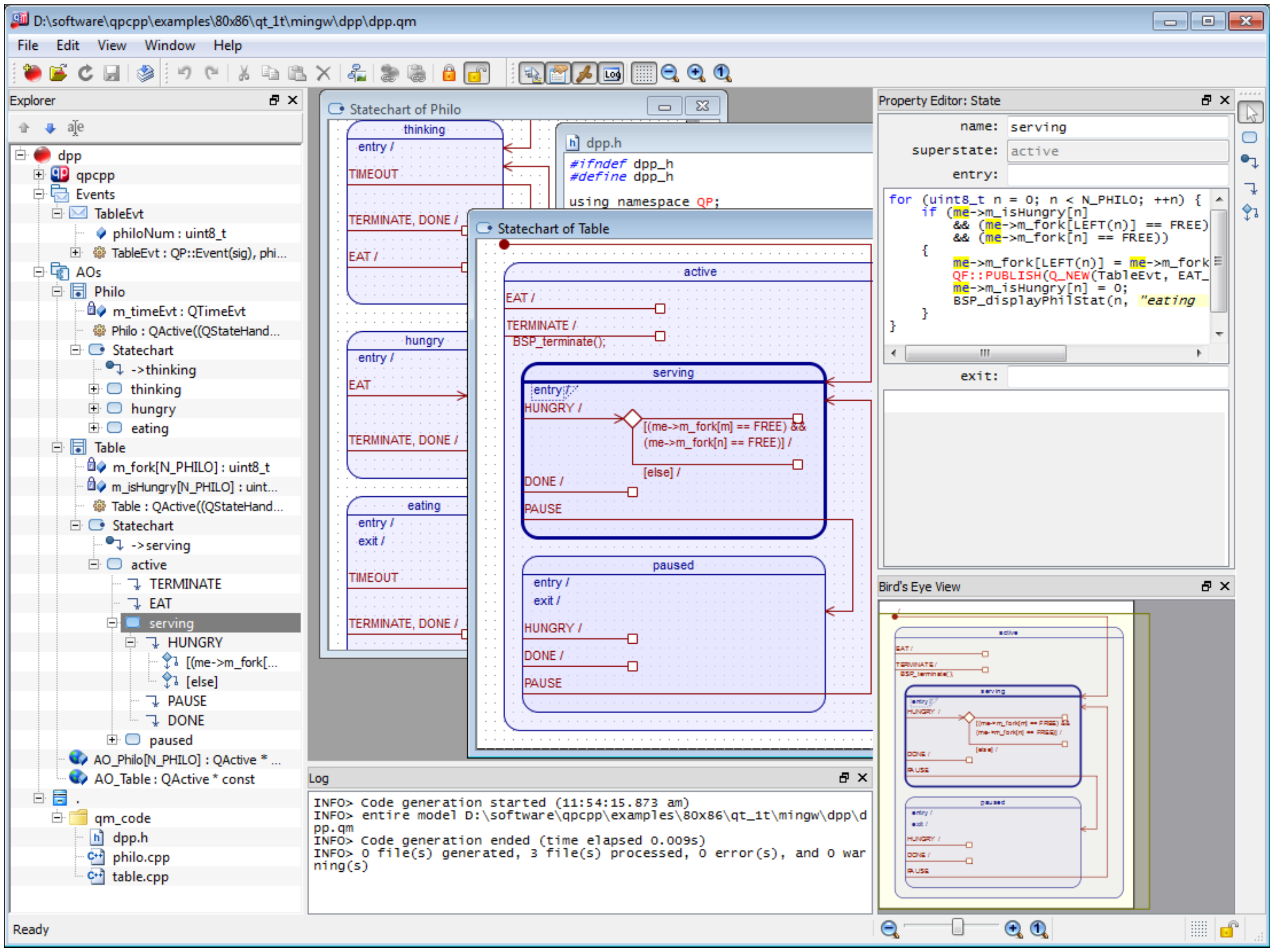
2.6 Generating the QP Application Code with the QM™ Modeling Tool

Both examples (DPP and the “Fly 'n' Shoot” game) come with the QM™ models of these applications (look for the files with extension `*.qm` in Listing 1).

NOTE: To open the models, you need to install the QM™ modeling tool from state-machine.com. QM™ is currently supported on Windows ,Linux and Mac OS X hosts. QM™ is **free** to download and **free** to use.

After you download and install QM, you open the provided models (e.g., `<qp>/examples/win32/-vc/dpp-gui/dpp.qm`) and press the “Generate Code” button. The example models are set up to generate code into the `qm_code/` sub-directory. The generated code is intentionally read-only, because it is not intended for manual editing. All changes to the generated code must be done by modifying the underlying model in the QM tool.

Figure 7: The DPP example in QM.



```
#ifndef dpp_h
#define dpp_h

using namespace QP;
```

name: serving
superstate: active
entry:

```
for (uint8_t n = 0; n < N_PHILO; ++n) {
    if (me->m_isHungry[n]
        && (me->m_fork[LEFT(n)] == FREE)
        && (me->m_fork[n] == FREE))
    {
        me->m_fork[LEFT(n)] = me->m_fork
        QF::PUBLISH(Q_NEW(TableEvt, EAT_
        me->m_isHungry[n] = 0;
        BSP_displayPhilStat(n, "eating
    }
}
```

exit:

```
INFO> Code generation started (11:54:15.873 am)
INFO> entire model D:\software\qpcpp\examples\80x86\qt_1t\mingw\dpp\d
pp.qm
INFO> Code generation ended (time elapsed 0.009s)
INFO> 0 file(s) generated, 3 file(s) processed, 0 error(s), and 0 war
ning(s)
```

3 The QP Port to Win32

This QP Port to Windows with the raw Win32 API is structured as follows:

- A QP application under Windows is a single Win32 process.
- Ever active object executes in a separate Win32 thread.
- The real-time framework (QF) does not use interrupts and handles everything at the task level (including the QF system clock tick).
- The critical section used in QF and QS is based on the Win32 critical-section object (`CRITICAL_SECTION`)
- The active objects use the built-in QF event queue (`QEQueue`) with the Win32 event object to block on an empty queue (via the `WaitForSingleObject()` Win32 API call).
- The port uses the built-in QF memory pools (`QMPool`) to store the dynamically allocated events.
- The port provides support for Win32-GUI programming, primarily with the intent of building realistic prototypes of embedded front panels. The general structure of an embedded front panel is set to be a **dialog box** with a picture of the front panel as a background bitmap and various controls corresponding to buttons, knobs, LEDs, and displays in the foreground. The port provides facilities for dot-matrix graphic displays, segment displays, owner-drawn buttons, LEDs, and custom bitmaps. See the upcoming Sections for more information.

NOTE: The main advantage of using a dialog box as the main application window is that dialog boxes can be very conveniently **designed graphically** with a resource editor. A number of such resource editors are available for Windows, but this Application Note assumes the excellent free ResEdit (www.resedit.net), which in many ways exceeds the editors bundled with Visual Studio.

NOTE: Visual Studio Express doesn't include a resource editor, so a third-part editor , such as ResEdit, is the only option.

3.1 The qep_port.h Header File

The `qep_port.h` header file determines the configuration of the QEP event processor for hierarchical state machines. The QEP header file for the Win32 port to Visual C++ is located in `<qp>\ports\win32\-vc\qep_port.h`. The Win32 port to MinGW is identical as to VC++ and is located in `<qp>\ports\-win32\mingw\qep_port.h`. The following Listing 2 shows the QEP configuration used in the Windows ports:

Listing 2: The QEP configuration for Windows

```
            /* Exact-width types. WG14/N843 C99 Standard, Section 7.18.1.1 */
#include "stdint.h"

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

3.1.1 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 don't provide the `<stdint.h>` header file. Instead a very rudimentary file `"stdint.h"` with the six exact-width integer types actually used in QP is provided in the `<qp>\ports\win32\vc` port directory.

3.2 The qf_port.h Header File

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 `<qp>\ports\win32\vc` or `<qp>\ports\win32\mingw`.

The most important design principle applied in the `qf_port.h` header file is to completely encapsulate the Win32 API (the `<windows.h>` header file), so that QP applications don't need to include it. The following Listing 3 shows the entire `qf_port.h` header file, which consists of two parts: (1) QP customization for Win32, and (2) platform-specific macros and includes for the QF port to Win32 (not visible to the QP applications).

NOTE: Avoiding the huge `<windows.h>` header file **dramatically speeds up compilation** of the application level code, even without using the precompiled header file option. This improvement in efficiency is critical for speeding up the development cycle, which is the most important advantage of developing embedded code on the desktop.

Listing 3: qf_port.h header file for the QF port to Win32

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

                    /* The maximum number of active objects in the application */
 (4) #define QF_MAX_ACTIVE               63

                         /* 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)        QF_enterCriticalSection_()
 (6) #define QF_CRIT_EXIT(dummy)         QF_leaveCriticalSection_()

     #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 */

     void QF_enterCriticalSection_(void);
     void QF_leaveCriticalSection_(void);
 (7) void QF_setTickRate(uint32_t ticksPerSec);              /* set clock tick rate */

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

                                        /* Win32 OS object object implementation */
(10)     #define QACTIVE_EQUEUE_WAIT_(me_) \
             while ((me_)->eQueue.frontEvt == (QEvent *)0) { \
                QF_CRIT_EXIT_(); \
                (void)WaitForSingleObject((me_)->osObject, (DWORD)INFINITE); \
                QF_CRIT_ENTRY_(); \
            }

(11)     #define QACTIVE_EQUEUE_SIGNAL_(me_) \
             (void)SetEvent((me_)->osObject)

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

                                             /* native QF event pool operations */
(13)     #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_))

         #define WIN32_LEAN_AND_MEAN
(14)     #include <windows.h>                                          /* Win32 API */

     #endif                                                           /* qf_pkg_h */
```

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

(2) The `QF_OS_OBJECT_TYPE` is used to block the calling active object thread when the event queue is empty. In the Win32 port this will be a handle to the Win32 event object, but to avoid reference to Win32 handles, it is defined as an opaque `void*` pointer.

(3) The `QF_THREAD_TYPE` is used to refer to a Win32 thread. In the Win32 port this is a handle to the Win32 thread, but to avoid reference to Win32 handles, it is defined as an opaque `void*` pointer.

(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 Win32 uses a single Win32 **critical section object** `CRITICAL_SECTION`. The Win32 critical section has the ability to nest, so the QF mechanism of “saving and restoring interrupt status” is not used. To avoid direct reference to the `CRITICAL_SECTION` object, the critical section entry/exit is encapsulated in the functions `QF_enterCriticalSection_()` and `QF_leaveCriticalSection_()`, respectively.

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.

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

(8) The callback function `QF_onClockTick()` is called at every clock tick and needs to be defined in the QP application. At the very minimum the function must call `QF_TICK()`, but may perform some other work as well.

(9) The code enclosed with `#ifdef qp_pkg_h/#endif` is visible and used only in the QF port, and is not visible to the application level.

(10) 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.

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

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

3.3 The qf_port.c Source File

The QF implementation file for the Win32 port is located in `<qp>\qf\win32\vc\src\qf_port.c`. This file contains only facilities for executing QP applications and is independent on the use of GUI. In other words, the same QP port to Win32 can be used in console and GUI applications. However, the QP applications with Win32 GUI require additional support, which is provided in the files `win32_gui.h` and `win32_gui.c`, which are discussed in the upcoming Sections 3.4 and 3.5, respectively.

Listing 4: qf_port.c implementation file for the QF port to Windows.

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

    Q_DEFINE_THIS_MODULE("qf_port")

    /* Local objects -------------------------------------------------------------*/
(1) static CRITICAL_SECTION l_win32CritSect;
    static DWORD l_tickMsec = 10;  /* clock tick in msec (argument for Sleep()) */
    static uint8_t l_running;

    static DWORD WINAPI thread_function(LPVOID arg);
```

```
     /*..........................................................................*/
     void QF_init(void) {
 (2)     InitializeCriticalSection(&l_win32CritSect);
     }
     /*..........................................................................*/
     void QF_enterCriticalSection_(void) {
 (3)     EnterCriticalSection(&l_win32CritSect);
     }
     /*..........................................................................*/
     void QF_leaveCriticalSection_(void) {
 (4)     LeaveCriticalSection(&l_win32CritSect);
     }
     /*..........................................................................*/
     void QF_stop(void) {
         l_running = (uint8_t)0;
     }
     /*..........................................................................*/
 (5) int16_t QF_run(void) {
 (6)     QF_onStartup();                                     /* startup callback */

                 /* raise the priority of this (main) thread to tick more timely */
 (7)     SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_HIGHEST);
         l_running = (uint8_t)1;
         while (l_running) {
 (8)         Sleep(l_tickMsec);                    /* wait for the tick interval */
 (9)         QF_onClockTick();      /* clock tick callback (must call QF_TICK()) */
         }
(10)     QF_onCleanup();                                     /* cleanup callback */
         QS_EXIT();                               /* cleanup the QSPY connection */
         //DeleteCriticalSection(&l_win32CritSect);
         return (int16_t)0;                                    /* return success */
     }
     /*..........................................................................*/
     void QF_setTickRate(uint32_t ticksPerSec) {
         l_tickMsec = 1000UL / ticksPerSec;
     }
     /*..........................................................................*/
(11) void QActive_start(QActive * const me, uint8_t prio,
                       QEvt const *qSto[], uint32_t qLen,
                       void *stkSto, uint32_t stkSize,
                       QEvt const *ie)
     {
         DWORD threadId;
         int p;

         Q_REQUIRE((qSto != (QEvt const **)0)  /* queue storage must be provided */
            && (stkSto == (void *)0));     /* Windows allocates stack internally */

         me->prio = prio;
         QF_add_(me);                     /* make QF aware of this active object */

(12)     QEQueue_init(&me->eQueue, qSto, (QEQueueCtr)qLen);
(13)     me->osObject = CreateEvent(NULL, FALSE, FALSE, NULL);
(14)     QF_ACTIVE_INIT_(&me->super, ie);          /* execute initial transition */
```

```
            /* NOTE: if stkSize==0, Windows allocates default stack size */
(15)        me->thread = CreateThread(NULL, stkSize,
                                     &thread_function, me, 0, &threadId);
            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 * const me) {
(18)        me->thread = (HANDLE)0;     /* stop the event loop in thread_function() */
        }
        /*..................................................................*/
(19)    static DWORD WINAPI thread_function(LPVOID arg) {     /* for CreateThread() */
            do {                                      /* QActive_stop() stops the loop */
(20)            QEvt const *e = QActive_get_((QActive *)arg);     /* wait for event */
(21)            QF_ACTIVE_DISPATCH_((QHsm *)arg, e);     /* dispatch to the AO's SM */
(22)            QF_gc(e);    /* check if the event is garbage, and collect it if so */
(23)        } while (((QActive *)arg)->thread != (HANDLE)0);
(24)        QF_remove_((QActive *)arg);/* remove this object from any subscriptions */
            CloseHandle(((QActive *)arg)->osObject);        /* cleanup the OS event */
            return 0;                                           /* return success */
        }
```

(1) The static (local) variable `l_win32CritSect` is the Win32 `CRITICAL_SECTION` object for implementing the QF critical section.

(2) The initialization of the framework consists of calling `InitializeCriticalSection()` Win32 API to initialize the global critical section object `l_win32CritSect`.

(3-4) The pair of functions `QF_enterCriticalSection_()`/`QF_leaveCriticalSection_()` implement the QF critical section with the `l_win32CritSect` object.

(5) `QF_run()` provides the system clock tick thread ("ticker thread").

(6) Before entering the endless loop, `QF_run()` invokes the `QF_onStartup()` callback.

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

(8) 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).

(9) `QF_run()` invokes the `QF_onClockTick()` callback by every clock tick.

> **NOTE:** You need to implement the `QF_onClickTick()` in you application code. At the minimum, `QF_onClickTick()` needs to call `QF_TICKX()`, but might perform other work as well.

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

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

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

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

(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 static function `thread_function()` is the active object’s thread routine.

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

(21) 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.

3.4 The win32_gui.h Header File

The `win32_gui.h` header file specifies the interface for GUI programming, which includes creating the main custom dialog box for the front panel as well as various useful facilities for owner-drawn buttons, graphical displays, segment displays, and LEDs.

Listing 5: win32_gui.h header file.

```
    #ifndef win32_gui_h
    #define win32_gui_h

    #define WIN32_LEAN_AND_MEAN
(1) #include <windows.h>                                               /* Win32 API */

    /* create the custom dialog hosting the embedded front panel ................*/
(2) HWND CreateCustDialog(HINSTANCE hInst, int iDlg, HWND hParent,
                          WNDPROC lpfnWndProc, LPCTSTR lpWndClass);

    /* OwnerDrawnButton "class" ................................................*/
(3) typedef struct OwnerDrawnButtonTag {
        HBITMAP hBitmapUp;
        HBITMAP hBitmapDown;
        HCURSOR hCursor;
    } OwnerDrawnButton;

    enum OwnerDrawnButtonAction {
        BTN_NOACTION,
        BTN_PAINTED,
        BTN_DEPRESSED,
        BTN_RELEASED
    };

    void OwnerDrawnButton_init(OwnerDrawnButton * const me,
                               HBITMAP hBitmapUp, HBITMAP hBitmapDwn,
                               HCURSOR hCursor);
    void OwnerDrawnButton_xtor(OwnerDrawnButton * const me);
    enum OwnerDrawnButtonAction OwnerDrawnButton_draw(
                                   OwnerDrawnButton * const me,
                                   LPDRAWITEMSTRUCT lpdis);

    /* GraphicDisplay class for drawing graphic displays with up to 24-bit color*/
(4) typedef struct GraphicDisplay Tag {
        UINT    width;
        UINT    xScale;
        UINT    height;
        UINT    yScale;
        HBITMAP hBitmap;
        HWND    hItem;
        BYTE   *bits;
        BYTE    bgColor[3];
    } GraphicDisplay;

    void GraphicDisplay_init( GraphicDisplay * const me,
                    UINT width,  UINT xScale,
                    UINT height, UINT yScale,
                HWND hItem,  BYTE const bgColor[3]);
    void GraphicDisplay_xtor(GraphicDisplay * const me);
    void GraphicDisplay_clear(GraphicDisplay * const me);
    void GraphicDisplay_setPixel(GraphicDisplay * const me, UINT x, UINT y,
                                BYTE const color[3]);
    void GraphicDisplay_clearPixel(GraphicDisplay * const me, UINT x, UINT y);
    void GraphicDisplay_redraw(GraphicDisplay * const me);

    /* SegmentDisplay "class" for drawing segment displays, LEDs, etc..........*/
(5) typedef struct SegmentDisplayTag {
        HWND    *hSegment;                          /* array of segment controls */
        UINT     segmentNum;                               /* number of segments */
        HBITMAP *hBitmap;                             /* array of bitmap handles */
        UINT     bitmapNum;                                 /* number of bitmaps */
    } SegmentDisplay;

    void SegmentDisplay_init(SegmentDisplay * const me,
                             UINT segNum, UINT bitmapNum);
    void SegmentDisplay_xtor(SegmentDisplay * const me);
    BOOL SegmentDisplay_initSegment(SegmentDisplay * const me,
                             UINT segmentNum, HWND hSegment);
    BOOL SegmentDisplay_initBitmap(SegmentDisplay * const me,
                             UINT bitmapNum, HBITMAP hBitmap);
    BOOL SegmentDisplay_setSegment(SegmentDisplay * const me,
                             UINT segmentNum, UINT bitmapNum);

    /* generic helper functions ................................................*/
    void DrawBitmap(HDC hdc, HBITMAP hBitmap, int xStart, int yStart);

    #endif                                                       /* win32_gui_h */
```

(1) The `<windows.h>` header file provides the APIs used in the Win32-GUI programming.

(2) The `CreateCustDialog()` function creates the custom dialog box for the main application window.

(3) The `OwnerDrawnButton` struct and functions form a “class” for owner-drawn buttons.

NOTE: `OwnerDrawnButton` is a proper class in the QP/C++ port to Win32.

(4) The `GraphicDisplay struct` and functions form a “class” for graphic displays (such as LCDs, OLEDs, etc) with up to 24-bit color. The `GraphicDisplay` “class” provides efficient pixel-level interface to a 24-bit “Device Independent Bitmap”.

NOTE: `GraphicDisplay` is a proper class in the QP/C++ port to Win32.

(5) The `SegmentDisplay struct` and functions form a “class” for segment displays. The `SegmentDisplay` “class” provides generic solution for managing multiple segment groups and multiple bitmaps corresponding to the specific configurations of the segments.

NOTE: `SegmentDisplay` is a proper class in the QP/C++ port to Win32.

The actual use of the provided GUI facilities is explained in the upcoming Section 5.

3.5 The win32_gui.c Source File

The `qf_port_gui.c` source file implements the facilities for rendering various GUI elements, such as LCD displays, owner-drawn buttons, and LEDs, which have been declared in the `qf_port.h` header file (see Listing 3(16-31)).

The implementation used in the `win32_gui.c` source file is intentionally low-level, with raw Win32 API, which does not require any higher-level software layers (such as MFC, .NET/C#, Qt, or wxWidgets). The main advantages of this approach are: simplicity of use and deployment, pure C interface, and unbeatable efficiency. The following Listing 6 shows the implementation of the `CreateCustDialog()` function, which must be called from `WinMain()` in the GUI applications with QP.

Listing 6: CreateCustDialog() function (file `win32_gui.c`).

```
(1) HWND CreateCustDialog(HINSTANCE hInst, int iDlg, HWND hParent,
                          WNDPROC lpfnWndProc, LPCTSTR lpWndClass)
    {
        WNDCLASSEX wndclass;
        HWND       hWnd;

        wndclass.cbSize        = sizeof(wndclass);
        wndclass.style         = CS_HREDRAW | CS_VREDRAW;
(2)     wndclass.lpfnWndProc   = lpfnWndProc;
        wndclass.cbClsExtra    = 0;
        wndclass.cbWndExtra    = DLGWINDOWEXTRA;
        wndclass.hInstance     = hInst;
        wndclass.hIcon         = NULL;
        wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
        wndclass.hbrBackground = (HBRUSH)COLOR_WINDOW;
        wndclass.lpszMenuName  = NULL;
(3)     wndclass.lpszClassName = lpWndClass;
        wndclass.hIconSm       = NULL;

(4)     RegisterClassEx(&wndclass);

(5)     hWnd = CreateDialog(hInst, MAKEINTRESOURCE(iDlg), hParent, NULL);

        /* NOTE: WM_INITDIALOG provides stimulus for initializing dialog controls.
        * Dialog box procedures typically use this message to initialize controls
        * and carry out any other initialization tasks that affect the appearance
        * of the dialog box.
        */
(6)     SendMessage(hWnd, WM_INITDIALOG, (WPARAM)0, (LPARAM)0);

        return hWnd;
    }
```

(1) The `CreateCustDialog()` function creates a custom dialog box based on the custom Windows class and must be called from `WinMain()`. The main job of the `CreateCustDialog()` function is registering the custom Window class for QP.

(2) The window procedure (WndProc) of the custom Window class for QP is set from the parameter provided to the `CreateCustDialog()` function.

NOTE: The QP-Win32 port associates the customized dialog box with a regular Window Procedure rather than a standard Dialog Procedure. This is necessary for greater flexibility.

(3) The name of the custom Window class for QP is is set from the parameter provided to the `CreateCustDialog()` function.

NOTE: The dialog box resource must explicitly set the CLASS property and the name of this property must be passed to the `CreateCustDialog()` function.

(4) The custom Window class is registered with Windows.

(5) The main Dialog window is created from the `iDlg` resource provided by the application as parameter to the `CreateCustDialog()` function.

(6) The `WM_INITDIALOG` message is sent to the WndProc associated with the dialog box. This message indicates the end of initialization and provides a convenient stimulus for starting active objects (see also the upcoming Section 5.5).

The `win32_gui.c` source file provides also the implementation of all the interesting and hard to find GUI facilities, such as the `GraphicDisplay` “class” for efficient rendering of graphic, pixel-addressable displays, `SegmentDisplay` “class” for segment displays of all kinds as well as LEDs, and `OwnerDrawnButton` “class” for rendering buttons with owner-defined look at both the “depressed” and “released” states and generating separate “depressed” and “released” events. The upcoming Section 5.5 discusses the use of the provided GUI toolkit by means of the “Fly 'n' Shoot” game example.

4 Console Applications

An example of a Win32 console application (application without a GUI) is provided in `<qp>\examples\win32\vc\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].

Due to the careful design of the QP port to Win32, the actual structure of the application is exactly **the same** in the Win32 simulation as it is in an embedded target. Specifically, the files: `dpp.h`, `main.c`, `philo.c`, and `table.c` are exactly the same. The only difference is the Board Support Package (BSP), implemented in the `bsp.c` source file.

4.1 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.

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

     #include <conio.h>
     #include <stdio.h>

     Q_DEFINE_THIS_FILE

     /* local variables ---------------------------------------------------------*/
     static uint32_t l_rnd;                                       /* random seed */
     . . .
     /*..........................................................................*/
     void QF_onStartup(void) {
(1)      QF_setTickRate(BSP_TICKS_PER_SEC);          /* set the desired tick rate */
     }
     /*..........................................................................*/
     void QF_onCleanup(void) {
     }
     /*..........................................................................*/
     void QF_onClockTick(void) {
(2)      QF_TICKX(0U, &l_clock_tick);     /* perform the QF clock tick processing */
(3)      if (_kbhit()) {                                     /* any key pressed? */
             int ch = _getch();
             if (ch == '\33') {                     /* see if the ESC key pressed */
                 QF_PUBLISH(Q_NEW(QEvt, TERMINATE_SIG), &l_clock_tick);
             }
             else if (ch == 'p') {
                 QF_PUBLISH(Q_NEW(QEvt, PAUSE_SIG), &l_clock_tick);
             }
         }
     }
     /*..........................................................................*/
     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();
```

```
    }
    /*..................................................................*/
(4) void BSP_init(void) {
        printf("Dining Philosopher Problem example"
               "\nQEP %s\nQF  %s\n"
               "Press 'p' to pause/un-pause\n"
               "Press ESC to quit...\n",
               QEP_getVersion(),
               QF_getVersion());

        BSP_randomSeed(1234U);
        Q_ALLEGE(QS_INIT((void *)0));
        QS_OBJ_DICTIONARY(&l_clock_tick);   /* must be called *after* QF_init() */
        QS_USR_DICTIONARY(PHILO_STAT);
    }
    /*..................................................................*/
    void BSP_terminate(int16_t result) {
        (void)result;
    #ifdef Q_SPY
        l_running = (uint8_t)0;                    /* stop the QS output thread */
    #endif
(5)     QF_stop();                               /* stop the main "ticker thread" */
    }
    /*..................................................................*/
(6) void BSP_displayPhilStat(uint8_t n, char const *stat) {
        printf("Philosopher %2d is %s\n", (int)n, stat);

        QS_BEGIN(PHILO_STAT, AO_Philo[n])  /* application-specific record begin */
            QS_U8(1, n);                                  /* Philosopher number */
            QS_STR(stat);                                 /* Philosopher status */
        QS_END()
    }
    /*..................................................................*/
(7) void BSP_displayPaused(uint8_t paused) {
        printf("Paused is %s\n", paused ? "ON" : "OFF");
    }
    . . .
```

(1) The `QF_onStartup()` callback sets the desired system clock tick.

NOTE: The clock tick period can really be adjusted only with the system time slice granularity (“quantum”). Currently in desktop Windows, 3 quantums are equal to either 10 milliseconds (single processor) or 15 milliseconds (multiple-processor Pentium), see http://support.microsoft.com/-kb/259025 for more information.

(2-3) The `QF_onClockTick()` callback function invokes the `QF_TICK()` and polls the keyboard to generate keyboard events to the application.

(4) The `BSP_init()` function prints out the opening string and initializes the QS facility (only effective in the Spy build configuration).

(5) The `BSP_terminate()` function calls `QF_stop()` to terminate the “ticker thread”.

(6) In case of a console application, the display of the Philosopher status is simple printout to the console.

(7) In case of a console application, the display of the Paused status is simple printout to the console.

5 GUI Applications

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).

5.1 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 "Fly 'n' Shoot" game example, where the front panel of the EK-LM3S811 board is displayed a **dialog box**. Compared to the real embedded target, the GUI version of the "Fly 'n' Shoot" game has been extended to demonstrate a segmented LCD display (used for the score) and interaction with the application by means of the mouse and the keyboard.

The examples of GUI applications for the plain Win32 API are located in `<qp>\examples\win32\vc\game-gui\` as well as the GUI version of the DPP example at `<qp>\examples\win32\vc\dpp-gui\`. The structure of these GUI applications is intentionally very close to the structure of the embedded QP applications. In particular, the application uses the exact same `main()` function, which runs in a separate Win32-thread. The GUI subsystem, centered around the `WinMain()` function, is loosely coupled with the QP application via events.

NOTE: 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/).

5.2 Main Steps for Creating QP Applications with GUI

The recommended steps for creating QP Applications with GUI on Windows are as follows:

NOTE: The described procedure is based on copying and modifying the existing examples to preserve the compiler and linker options as well as other project settings. The described procedure assumes that you have installed Visual C++ Express Edition, Platform SDK, and ResEdit resource editor.

1. Copy the example GUI application directory (e.g., `<qp>\examples\win32\vc\dpp-gui\`) and rename the copy to your own project.

2. Remove `*.sdf`, `*.suo`, and `*.vxproj.user` files from the project directory.
3. Rename the `.sln` (solution) file to the desired name of your project (leave the `.sln` extension). Open the renamed solution file in a text editor and replace all occurrences of the original example name with the your project name. Save the solution file.
4. Rename the `.vcxproj` (project) file to the desired name of your project (leave the `.vcxproj` extension). Open the renamed project file in a text editor and automatically replace all occurrences of the original example name with the your project name. Save the project file.
5. Rename the `.rc` (resource) file to the desired name of your project (leave the `.rc` extension).
6. Open the solution file in Visual C++ Express.
7. Remove any unused files from the project and add any of your own files into the project. Don not forget the resource file.
8. Open the resource file in ResEdit and modify the front panel to your project. This step typically requires working with bitmaps, which you can either generate from digital pictures of your embedded front panel, or from your sketches. The upcoming Section ?? describes the design of the “Fly 'n' Shoot” front panel with ResEdit.
9. Adapt the `bsp.c` file to generate QP events from your GUI and to implement actions to modify the GUI in response to QP events.

NOTE: Even though this section goes into some details of the Win32 API, you don't necessarily need to know all these details to use the provided example code as a template for your own front panels. The code is thoroughly commented to help you identify sections that you need to customize.

5.3 WinMain()

In case of a GUI application, the GUI subsystem is located in the Board Support Package (the `bsp.c` module). For the “Fly 'n' Shoot” game example, the BSP is located in `<qp>\examples\win32\vc\game-gui\bsp.c`. Listing 8 shows the relevant fragment of the `bsp.c` file. This section describes the `WinMain()` function.

Listing 8: WinMain() (file `bsp.c`).

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

(4) #include "win32_gui.h"      /* Win32 GUI elements for embedded front panels */
(5) #include "resource.h" /* GUI resource IDs generated by the resource editior */

    #include <stdio.h>                                      /* for _snprintf_s() */

    Q_DEFINE_THIS_FILE

    /* local variables ---------------------------------------------------------*/
    static HINSTANCE l_hInst;                          /* this application instance */
    static HWND      l_hWnd;                                   /* main window handle */
    static LPSTR     l_cmdLine;                            /* the command line string */

(6) static GraphicDisplay   l_oled; /* the OLED display of the EK-LM3S811 board */
(7) static SegmentDisplay   l_userLED;      /* USER LED of the EK-LM3S811 board */
(8) static SegmentDisplay   l_scoreBoard;      /* segment display for the score */
```

```
 (9) static OwnerDrawnButton l_userBtn;   /* USER button of the EK-LM3S811 board */

     /* (R,G,B) colors for the OLED display */
     static BYTE const c_onColor [3] = { 255U, 255U,   0U };            /* yellow */
     static BYTE const c_offColor[3] = {  15U,  15U,  15U };   /* very dark grey */
     . . .
     /* Local functions ---------------------------------------------------------*/
     static LRESULT CALLBACK WndProc(HWND hWnd, UINT iMsg,
                                     WPARAM wParam, LPARAM lParam);

     /*..........................................................................*/
(10) int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hPrevInst,
                       LPSTR cmdLine, int iCmdShow)
     {
         HWND hWnd;
         MSG  msg;

         (void)hPrevInst;       /* avoid compiler warning about unused parameter */

(11)     l_hInst   = hInst;                        /* save the application instance */
(12)     l_cmdLine = cmdLine;                       /* save the command line string */

         /* create the main custom dialog window */
(13)     hWnd = CreateCustDialog(hInst, IDD_APPLICATION, NULL,
                                 &WndProc, "QP_APP");
(14)     ShowWindow(hWnd, iCmdShow);                        /* show the main window */

         /* enter the message loop... */
(15)     while (GetMessage(&msg, NULL, 0, 0)) {
             TranslateMessage(&msg);
             DispatchMessage(&msg);
         }
         return msg.wParam;
     }
```

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

(4-5) The GUI application includes the header file `win32_gui.h` discussed in Section 3.4 and `resource.h` generated by the resource editor (such as ResEdit).

(6-9) The GUI elements specific for your embedded panel, such as dot-matrix displays, segmented displays, LEDs, and owner-drawn buttons are allocated statically. You can have any number of instances of each component type.

(10) As in every Win32 GUI application, you need the `WinMain()` function, which is the main entry point to every GUI application on Windows.

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".

(11-12) The current instance handle and the command line are saved in static variables.

(13) The function `CreateCustDialog()` creates the main application dialog box window with the **customized Windows class** and regular window procedure (WndProc) as opposed to the dialog box procedure typical for dialog boxes.

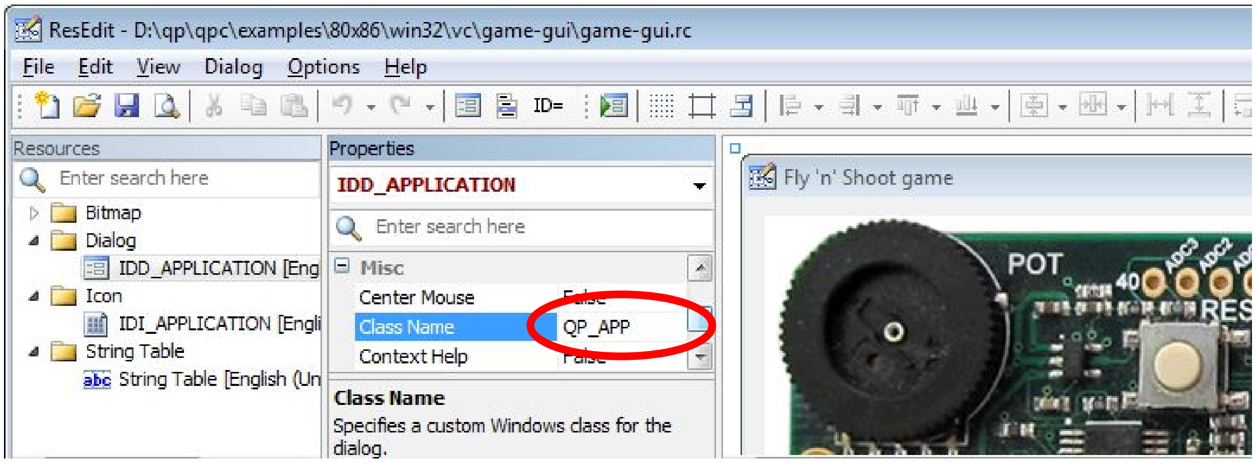
NOTE: The created dialog box is based on a customized Windows class with the name specified as the last parameter (`"QP_APP"` in this case). For this to work, the dialog box resource must be associated with the same Windows class in the "Class Name" property of the resource editor, as shown in Figure 8.

(14) The created dialog box window must be shown.

(15) The `WinMain()` function enters the standard "message pump", in which it processes all messages until the application is closed.

NOTE: The "message pump" can be extended to handle keyboard accelerators and perhaps other events.

Figure 8: Setting the Windows Class Name of the Dialog Box in ResEdit.



Center Mouse: False
Class Name: QP_APP
Context Help: False

Class Name
Specifies a custom Windows class for the dialog.

The Windows Class Name can also be visible directly in the resource file. Listing 9 shows the definition of the dialog box resource in the `game-gui.rc` resource file generated by the resource editor. Please note the highlighted definition of the `CLASS "QP_APP"`.

Listing 9: Definition of the "Fly 'n' Shoot" dialog box resource (file game-gui.rc).

```
//
// Dialog resources
//
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
IDD_APPLICATION DIALOGEX 0, 0, 580, 227
STYLE DS_ABSALIGN | DS_MODALFRAME | DS_SHELLFONT | WS_CAPTION
                   | WS_VISIBLE | WS_POPUP | WS_SYSMENU
CAPTION "Fly 'n' Shoot game"
CLASS "QP_APP"
FONT 8, "MS Shell Dlg", 0, 0, 1
{
    CONTROL IDB_EK_LM3S811, IDC_BACKGROUND, WC_STATIC, SS_BITMAP,
            7, 7, 533, 181
    CONTROL "USER", IDC_USER, WC_BUTTON, WS_TABSTOP | WS_TABSTOP
            | BS_OWNERDRAW | BS_BITMAP, 374, 132, 28, 27
    CONTROL IDB_LCD, IDC_LCD, WC_STATIC, SS_BITMAP, 287, 85, 128, 20
    CTEXT   "www.state-machine.com", IDC_STATIC, 224, 212, 103, 8,
            SS_CENTER, WS_EX_TRANSPARENT
        DEFPUSHBUTTON   "Quit", IDOK, 523, 7, 50, 14, WS_GROUP
        CONTROL IDB_SEG, IDC_STATIC, WC_STATIC, SS_BITMAP, 7, 169, 97, 54
        CONTROL IDB_SEG8, IDC_SEG3, WC_STATIC, SS_BITMAP, 16, 177, 19, 36
        CONTROL IDB_SEG8, IDC_SEG2, WC_STATIC, SS_BITMAP, 36, 177, 19, 36
        CONTROL IDB_SEG8, IDC_SEG1, WC_STATIC, SS_BITMAP, 56, 177, 19, 36
        CONTROL IDB_SEG8, IDC_SEG0, WC_STATIC, SS_BITMAP, 76, 177, 19, 36
        CONTROL IDB_LED_OFF, IDC_LED, WC_STATIC, SS_BITMAP, 417, 36, 19, 18
        LTEXT   "MPool[0]:", IDC_STATIC, 135, 193, 31, 8, SS_LEFT
        LTEXT   "?", IDC_MPOOL0, 168, 193, 44, 8, SS_LEFT
    }
    . . .
```

5.4 Window Procedure (WndProc)

The `WinMain()` function creates the main dialog box window and associates the Window Procedure (traditionally called WndProc) `WndProc` with it (see Listing 8(13)). The main job of the WndProc is processing windows messages such as mouse clicks, button presses, or menu commands.

NOTE: Even though the QP GUI application is based on a dialog box as the main window, the application uses a custom Windows class with a **regular Window Procedure** rather than a Dialog Procedure typically associated with dialog boxes. A standard Dialog Procedure is not used, because it is too specialized, which would be too limiting for the embedded front panels (e.g., the Dialog Procedure contains a specialized message loop that processes keyboard input, tab order, etc. and does not allow to override these aspects easily.)

However, in a QP application, WndProc should only translate the Windows messages 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, but rather is left to the active objects—preferably to the single active object called “GUI-Manager”.

The main benefit of this design 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).

NOTE: In the GAME-GUI example, the Tunnel active object is the only one performing output to the GUI (through the `BSP_drawBitmap()` and other similar BSP functions), so it plays here the role of the “GUI Manager”.

Listing 10: WndProc() (file `bsp.c`).

```
(1) static LRESULT CALLBACK WndProc(HWND hWnd, UINT iMsg,
                                    WPARAM wParam, LPARAM lParam)
    {
        switch (iMsg) {
            /* Perform initialization upon creation of the main dialog window
            * NOTE: Any child-windows are NOT created yet at this time, so
            * the GetDlgItem() function can't be used (it will return NULL).
            */
(2)         case WM_CREATE: {
(3)             l_hWnd = hWnd;                          /* save the window handle */
```

```
                /* initialize the owner-drawn buttons...
                * NOTE: must be done *before* the first drawing of the buttons,
                * so WM_INITDIALOG is too late.
                */
(4)             OwnerDrawnButton_init(&l_userBtn,
                           LoadBitmap(l_hInst, MAKEINTRESOURCE(IDB_BTN_UP)),
                           LoadBitmap(l_hInst, MAKEINTRESOURCE(IDB_BTN_DWN)),
                           LoadCursor(NULL, IDC_HAND));
                return 0;
            }

            /* Perform initialization after all child windows have been created */
(5)         case WM_INITDIALOG: {
(6)             GraphicDisplay_init(&l_oled,
                           BSP_SCREEN_WIDTH,  2U,     /* scale horizontaly by 2 */
                           BSP_SCREEN_HEIGHT, 2U,      /* scale vertically by 2 */
                           GetDlgItem(hWnd, IDC_LCD), c_offColor);

(7)             SegmentDisplay_init(&l_scoreBoard,
                                 4U,           /* 4 "segments" (digits 0-3) */
                                 10U);        /* 10 bitmaps (for 0-9 states) */
                SegmentDisplay_initSegment(&l_scoreBoard,
                     0U, GetDlgItem(hWnd, IDC_SEG0));
                SegmentDisplay_initSegment(&l_scoreBoard,
                     1U, GetDlgItem(hWnd, IDC_SEG1));
                . . .
                SegmentDisplay_initBitmap(&l_scoreBoard,
                     0U, LoadBitmap(l_hInst, MAKEINTRESOURCE(IDB_SEG0)));
                SegmentDisplay_initBitmap(&l_scoreBoard,
                     1U, LoadBitmap(l_hInst, MAKEINTRESOURCE(IDB_SEG1)));
                . . .
                SegmentDisplay_initBitmap(&l_scoreBoard,
                     9U, LoadBitmap(l_hInst, MAKEINTRESOURCE(IDB_SEG9)));

                BSP_updateScore(0U);

(8)             SegmentDisplay_init(&l_userLED,
                                 1U,         /* 1 "segment" (the LED itself) */
                                 2U);   /* 2 bitmaps (for LED OFF/ON states) */
                SegmentDisplay_initSegment(&l_userLED,
                     0U, GetDlgItem(hWnd, IDC_LED));
                SegmentDisplay_initBitmap(&l_userLED,
                     0U, LoadBitmap(l_hInst, MAKEINTRESOURCE(IDB_LED_OFF)));
                SegmentDisplay_initBitmap(&l_userLED,
                     1U, LoadBitmap(l_hInst, MAKEINTRESOURCE(IDB_LED_ON)));

                /* --> QP: spawn the application thread to run main() */
(9)             Q_ALLEGE(CreateThread(NULL, 0, &appThread, NULL, 0, NULL)
                         != (HANDLE)0);
                return 0;
            }

(10)        case WM_DESTROY: {
                BSP_terminate(0);
                return 0;
                }

                /* commands from regular buttons and menus... */
(11)            case WM_COMMAND: {
                    SetFocus(hWnd);
                    switch (wParam) {
                        case IDOK:
                        case IDCANCEL: {
                            BSP_terminate(0);
                            break;
                        }
                    }
                    return 0;
                }

                /* owner-drawn buttons... */
(12)            case WM_DRAWITEM: {
(13)                LPDRAWITEMSTRUCT pdis = (LPDRAWITEMSTRUCT)lParam;
(14)                switch (pdis->CtlID) {
(15)                    case IDC_USER: {                 /* USER owner-drawn button */
(16)                        switch (OwnerDrawnButton_draw(&l_userBtn, pdis)) {
(17)                            case BTN_DEPRESSED: {
                                    BSP_playerTrigger();
                                    SegmentDisplay_setSegment(&l_userLED, 0U, 1U);
                                    break;
                                }
(18)                            case BTN_RELEASED: {
                                    SegmentDisplay_setSegment(&l_userLED, 0U, 0U);
                                    break;
                                }
                            }
                            break;
                        }
                    }
                    return 0;
                }

                /* mouse input... */
(19)            case WM_MOUSEWHEEL: {
                    if ((HIWORD(wParam) & 0x8000U) == 0U) {/* wheel turned forward? */
                        BSP_moveShipUp();
                    }
                    else {                        /* the wheel was turned backwards */
                        BSP_moveShipDown();
                    }
                    return 0;
                }

                /* keyboard input... */
(20)            case WM_KEYDOWN: {
                    switch (wParam) {
                        case VK_UP:
                            BSP_moveShipUp();
                            break;
                        case VK_DOWN:
                            BSP_moveShipDown();
                        break;
                    case VK_SPACE:
                        BSP_playerTrigger();
                        break;
                }
                return 0;
            }

        }
(21)    return DefWindowProc(hWnd, iMsg, wParam, lParam) ;
    }
```

(1) The standard WndProc is made `static` here, because it is not used outside the `bsp.c` module.

(2) The `WM_CREATE` Windows message is sent to the WndProc upon the creation of the associated window.

NOTE: The dialog box controls (child-windows) are **not** created yet at this time, so the `GetDlgItem()` function can't be used in response to `WM_CREATE` (it will return `NULL`).

---

(3) The main window handle is copied to the static variable to use in other BSP functions.

(4) Any owner-drawn buttons must be initialized before they are drawn, which happens after `WM_CREATE` is sent to the WndProc. Please see the Windows message `WM_DRAWITEM` at label (12) and the upcoming Section 5.5.3 for more information about the owner-drawn buttons.

(5) The `WM_INITDIALOG` Windows message is not typically sent to regular WndProcs, but it is specifically generated in the `CreateCustDialog()` function (see Listing 8(13)) after all the child window controls have been created. The `WM_INITDIALOG` message is very useful to initialize any dialog controls, such as buttons, bitmaps, text labels, etc.

(6) The WndProc for the game example initializes the dot-matrix display `l_oled`. Please see the upcoming Section 5.5.4 for more information about dot-matrix displays.

(7) The WndProc for the game example initializes also the segment display `l_scoreBoard`. Please see the upcoming Section 5.5.5 for more information about segment displays.

(8) Finally, the WndProc for the game example initializes the LED (`l_userLED`) as a segment display. An LED is treated here as a simple 1-segment display with two states LED-off and LED-on.

(9) After all initialization is done, the WndProc creates a Win32 thread to run the QP application. The thread routine passed to the `CreateThread()` function is just a thin wrapper around the `main_gui()` function.

NOTE: The thread routine `appThread()` is defined as follows:

```
static DWORD WINAPI appThread(LPVOID par) {
    (void)par;                           /* unused parameter */
    return main_gui();              /* run the QF application */
}
```

---

(10) The `WM_DESTROY` Windows message is sent to the WndProc when the main window is closed or the application is forced to quit by the operating system. The `BSP_terminate()` calls `PostQuiteMessage()` function as well as `QF_stop()`, which causes termination of the application thread.

(11) The `WM_COMMAND` Windows message is generated by users clicking on regular buttons or activating menus. The game example contains one such button “Quit”.

(12) Owner-drawn buttons are treated differently than regular buttons such as “Quit”. They produce the `WM_DRAWITEM` Windows messages, which is sent when an owner-drawn button is pressed and again when it is released. In simulating real buttons in embedded front panels this is typically exactly what you want.

NOTE: The Win32 API for owner-drawn controls is a bit complicated, because it covers not just buttons but owner-drawn menus, combo-boxes, and many others. However, for owner-drawn buttons most of the complexities are hidden inside the `OwnerDrawnButton_draw()` function provided in the QP port to Win32.

(13) The `DRAWITEMSTRUCT` pointer is extracted from the Windows message parameter.

(14) The `switch` statement discriminates based on the ID of the child-window control. Here, you decide which owner-drawn button you want to service.

(15) For example, to service the USER button of the EK-LM3S811 board (see Figure 6), you provide a case statement labeled with `IDC_USER`, which is the ID of the owner-drawn button set in the resource editor.

(16) The function `OwnerDrawnButton_draw()`, which is provided in the QP-Win32 port, performs the “heavy lifting” of painting the button in the current state and returns the status of the button.

(17) When the button is in the `BTN_DEPRESSED` state, the game example calls `BSP_playerTrigger()`, which publishes the `PLAYER_TRIGGER` event to the active objects and also turns the USER LED on.

(18) When the button is in the `BTN_RELEASED` state, the game example turns the USER LED off.

(19) The `WM_MOUSEWHEEL` Windows message is generated by the mouse wheel. This code demonstrates how to handle the mouse-wheel input.

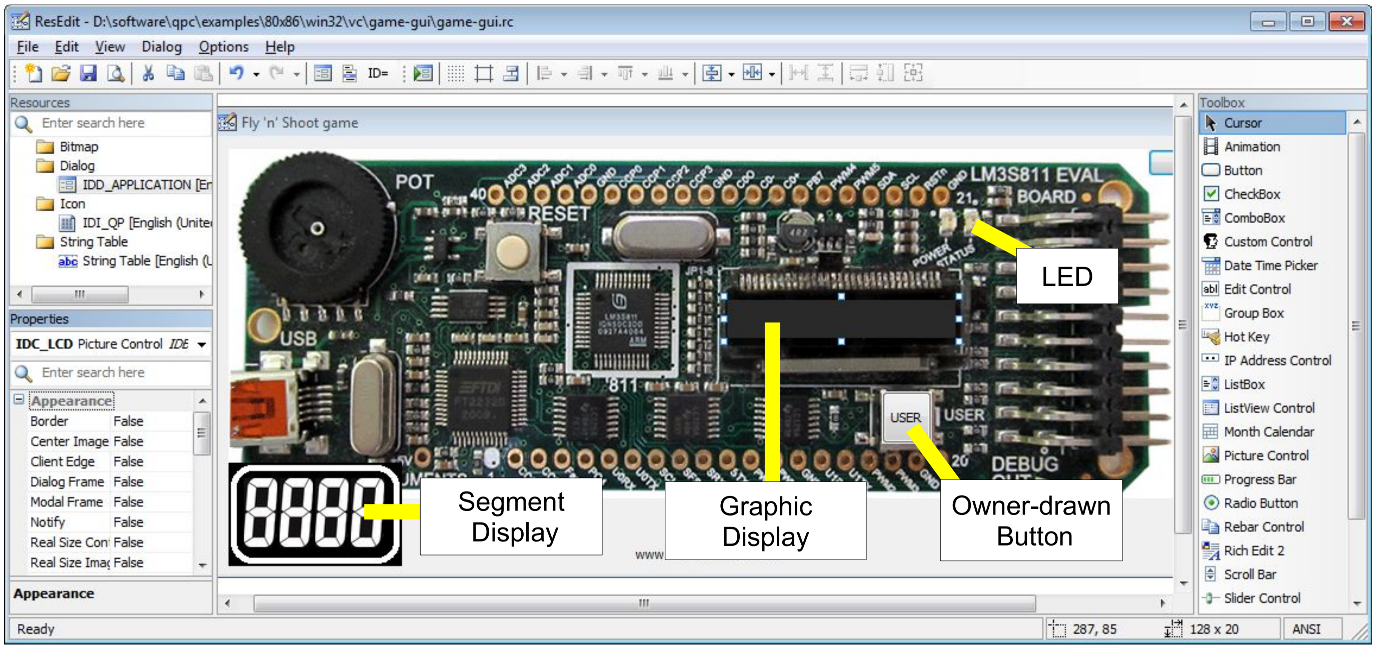
(20) The `WM_KEYDOWN` Windows message is generated by pressing keys on the keyboard. This code demonstrates how to provide keyboard input to your application.

5.5 Developing Realistic Embedded Front Panels

This Application Note has been specifically designed to provide a **complete toolkit** with all the needed graphical components and powerful **free** tools (such as the free Visual C++ Express Edition and the free ResEdit resource editor) for building realistic embedded front panels.

This section describes a typical process of building realistic embedded front panels graphically with the ResEdit resource editor and the facilities for implementing owner-drawn buttons, dot-matrix displays, and segment displays, which are provided in the `win32_gui.c` source file.

Figure 9: Prototype of the EK-LM3S811 board open in ResEdit resource editor



Properties

IDC_LCD Picture Control *IDE*

Appearance

Border	False
Center Image	False
Client Edge	False
Dialog Frame	False
Modal Frame	False
Notify	False
Real Size Con	False
Real Size Imag	False

5.5.1 *The dialog box resource*

As described in the previous sections, the structure of a GUI application with this QP-Win32 port is based on dialog box, which can be designed graphically in a resource editor, such as ResEdit. Consequently, the process of prototyping an embedded front panel starts with creating the Dialog resource. In ResEdit, you create or open an existing resource file and select menu **File | Add a resource... | Dialog**. Once the dialog resource is added, you can click on it and adjust its properties in the Property Editor.

NOTE: The most important step here is defining the **Class Name** property, because this property cannot stay undefined (see Figure 8). The Class Name you choose must match the name you pass to the `CreateCustDialog()` function.

5.5.2 *The background image*

Next, you typically want to add a background image of your front panel to the dialog box, such as the EK-LM3S811 board shown in Figure 9. The most important aspect of preparing the background image is the proper **scaling**. You should scale the image carefully such that any dot-matrix display present on the panel contains the desired number of pixels. Please note that to accommodate small pixel-count embedded displays, the provided GUI facilities for rendering dot-matrix displays allow you to **scale** the pixels independently in the horizontal and vertical directions. For example, the OLED display of the EK-

LM3S811 board has resolution of 96 x 16 pixels, which is a very small area on the modern high-resolution monitors. Therefore, the image of the board has been scaled such that the OLED display contains 192 x 32 pixels, which is exactly two times bigger than the original resolution.

The background image needs to be converted to the BMP format, which can be done by a number of programs, for example the Paint utility bundled in every version of Windows. Save the image as a bitmap in the `Res\` directory of your project.

To add the bitmap image to ResEdit, select menu **File | Add a resource... | Bitmap**. Choose the "Create from an existing file" option and choose the bitmap file you've just added from the `Res\` directory. Next, you add a "Picture Control" to the Dialog resource from the provided Toolbox. In the Property Editor of the newly added Picture Control, you change the type to "Bitmap" and select the ID corresponding to the background bitmap added in the previous step.

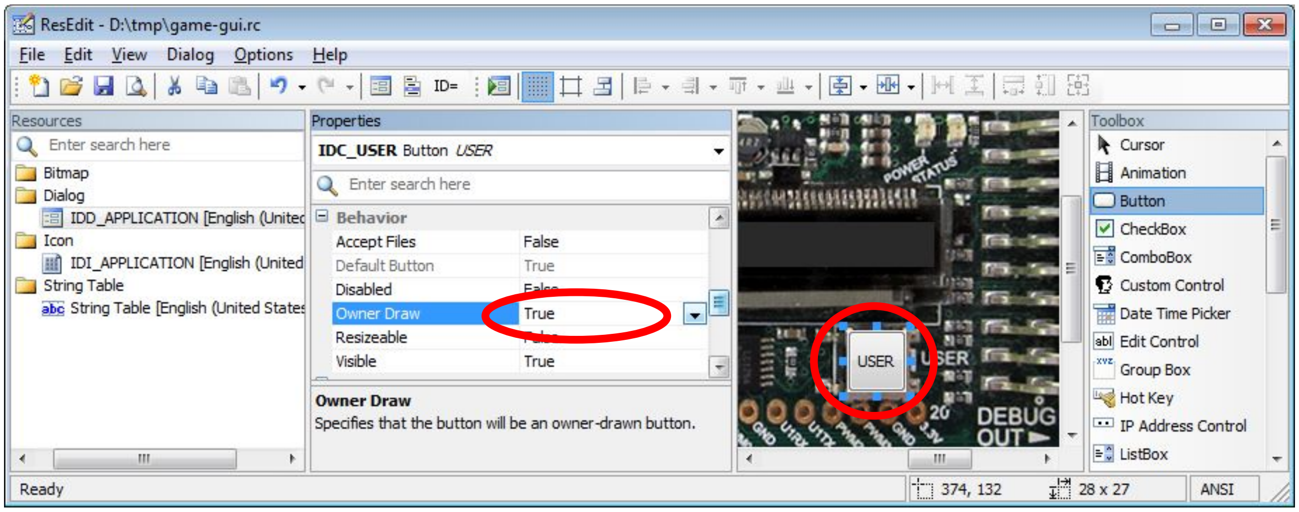
NOTE: ResEdit allows you to easily control the **stacking order** of the controls placed in the dialog box. To change the stacking order, right-click on a control and choose the option **Order | Bring to …** from the pop-up menu. The background image should obviously be always "sent to background" so that it does not come on top of any other controls.

5.5.3 *Owner-drawn buttons*

Owner-drawn buttons have two critical advantages over the regular buttons when it comes to embedded front panels. (1) they can **look** exactly as your physical buttons on your panel and, more importantly, (2) they generate events when they are **depressed** and when they are **released**. In contrast, the regular buttons are always rectangular-gray and they generate only the "clicked" event, which appears as `WM_COMMAND` in the WndProc.

To add an owner-drawn button to the dialog box, select the "Button" control from the Toolbox and place in the desired spot. Next, change the property "Owner drawn" to True.

Figure 10: Adding an owner-drawn button in ResEdit resource editor



IDC_USER Button *USER*

Behavior

Accept Files	False
Default Button	True
Disabled	False
Owner Draw	True
Resizeable	False
Visible	True

Owner Draw
Specifies that the button will be an owner-drawn button.

NOTE: The design of a front panel requires quite precise placement of the controls. To achieve the **fine control of placement in ResEdit,** you need to press **Ctrl key** and use the arrow keys to nudge the control in the desired direction. Without the Ctrl key, ResEdit places the control at coarse grid locations.

The owner-drawn button requires also some code. which is illustrated in Listing 11.

Listing 11: Snippets from the bsp.c file pertaining to owner-drawn button

```
(1) static OwnerDrawnButton l_userBtn;   /* USER button of the EK-LM3S811 board */
    . . .
    static LRESULT CALLBACK WndProc(HWND hWnd, UINT iMsg,
                                    WPARAM wParam, LPARAM lParam)
    {
        switch (iMsg) {
            case WM_CREATE: {
                . . .
(2)             OwnerDrawnButton_init(&l_userBtn,
                           LoadBitmap(l_hInst, MAKEINTRESOURCE(IDB_BTN_UP)),
                           LoadBitmap(l_hInst, MAKEINTRESOURCE(IDB_BTN_DWN)),
                           LoadCursor(NULL, IDC_HAND));
                return 0;
            }
            /* owner-drawn buttons... */
            case WM_DRAWITEM: {
                LPDRAWITEMSTRUCT pdis = (LPDRAWITEMSTRUCT)lParam;
                switch (pdis->CtlID) {
                    case IDC_USER: {                 /* USER owner-drawn button */
(3)                     switch (OwnerDrawnButton_draw(&l_userBtn, pdis)) {
(4)                        case BTN_DEPRESSED: {
                                BSP_playerTrigger();
                                SegmentDisplay_setSegment(&l_userLED, 0U, 1U);
                                break;
                            }
                            case BTN_RELEASED: {
(5)                             SegmentDisplay_setSegment(&l_userLED, 0U, 0U);
                                break;
                            }
                        }
                        break;
                    }
                }
                return 0;
            }
     . . .
     }
```

(1) You need to provide an instance of the `OwnerDrawnButton struct` for each owner-drawn button you have added to the Dialog box.

(2) You need to call `OwnerDrawnButton_init()` in the `WM_CREATE` message handler with the bitmaps for the “Released” state, “Depressed” state, and the mouse cursor when the mouse hovers above the button. You can specify the cursor shape to NULL, in which case the standard arrow cursor is used.

(3) You need to call `OwnerDrawButton_draw()` the `WM_DRAWITEM` message handler to render the button in the current state.

5.5.4 *Graphic displays*

Graphic displays of various types (LCDs, OLED, etc.) become increasingly popular in embedded systems. The QP-Win32 port provides a generic, efficient implementation of a pixel-addressable graphic displays with up to 24-bit color (RGB).

The use model of the provided dot-matrix display is to perform multiple, efficient pixel updates in-memory and then render the updated bitmap to the screen. Also, to accommodate low-resolution displays, the dot-matrix facility allows **scaling** the display independently in horizontal and vertical directions. For example, the OLED display of the EK-LM3S811 board has been scaled up by factor of 2 horizontally and 2 vertically, to the total size of 192 x 32 pixels (each original pixel corresponding to 2x2 pixels on the dot-matrix display.) Of course, the 2x2 scaling is just an example and you can use other scaling factors.

Under the hood, the implementation is based on the Device-Independent Bitmap (DIB) section of an in-memory bitmap (`CreateDIBSection()` Win32 API, see `win32.gui.c` source file in the QP-Win32 port.) The `GraphicDisplay` bitmap uses the **standard coordinate system**, in which the origin (0,0) is placed in the top-left corner, the x-coordinate extends horizontally and grows to the right, while the y-coordinate extends vertically and grows down.

To add a dot-matrix display to the dialog box in ResEdit, select the “Picture Control” from the Toolbox and place in the desired spot. In the Property Editor of the newly added Picture Control, you change the type to “Bitmap”. The dot-matrix display requires also some code. which is illustrated in Listing 12.

Listing 12: Snippets from the bsp.c file pertaining to graphic display

```
(1) static GraphicDisplay   l_oled; /* the OLED display of the EK-LM3S811 board */
    /* (R,G,B) colors for the OLED display */
(2) static BYTE const c_onColor [3] = { 255U, 255U,   0U };           /* yellow */
(3) static BYTE const c_offColor[3] = {  15U,  15U,  15U };   /* very dark grey */
    . . .
    /*..........................................................................*/
    static LRESULT CALLBACK WndProc(HWND hWnd, UINT iMsg,
                                    WPARAM wParam, LPARAM lParam)
    {
        switch (iMsg) {
            /* Perform initialization after all child windows have been created */
            case WM_INITDIALOG: {
(4)             GraphicDisplay_init(&l_oled,
                             BSP_SCREEN_WIDTH,  2U,    /* scale horizontally by 2 */
                             BSP_SCREEN_HEIGHT, 2U,      /* scale vertically by 2 */
                             GetDlgItem(hWnd, IDC_LCD), c_offColor);
        . . .
    }
    /*..........................................................................*/
(5) void BSP_drawBitmap(uint8_t const *bitmap) {
        UINT x, y;
        /* map the EK-LM3S811 OLED pixels to the GraphicDisplay pixels... */
        for (y = 0; y < BSP_SCREEN_HEIGHT; ++y) {
            for (x = 0; x < BSP_SCREEN_WIDTH; ++x) {
                uint8_t bits = bitmap[x + (y/8)*BSP_SCREEN_WIDTH];
                if ((bits & (1U << (y & 0x07U))) != 0U) {
(6)                 GraphicDisplay_setPixel(&l_oled, x, y, c_onColor);
                }
                else {
(7)                 GraphicDisplay_clearPixel(&l_oled, x, y);
                }
            }
```

```
           }
 (8)       GraphicDisplay_redraw(&l_oled);  /* draw the updated GraphicDisplay on the
screen */
       }
```

(1) You need to provide an instance of the `GraphicDisplay struct` for each dot-matrix display you have added to the Dialog box.

(2-3) To simulate a monochrome display, the colors for the "on" and "off" pixel states are defined.

(4) You need to call `GraphicDisplay_init()` in the `WM_INITDIALOG` message handler with the dimensions of the display, the scaling factors, and the color of the blank pixels.

(5) The `BSP_drawBitmap()` function renders the whole 96x16 pixel OLED display of the EK-LM3S811 board. This function performs a re-mapping of the specific pixel representation for the OLED display to the pixels of the `GraphicDisplay` device-independent bitmap. In this particular case, the monochrome pixels are represented as tightly-packed bitmasks of 8-pixels per byte arranged vertically.

NOTE: The particular pixel re-mapping is just a demonstration here. You need to adapt it for your specific display. Also, you don't need to always update the whole pixel array. You might choose to update only a part of it, as is illustrated in the function `BSP_drawString()` in `bsp.c` (not shown in Listing 12).

(6) The function `GraphicDisplay_setPixel()` very efficiently sets a given pixel at (x, y) to the given color.

(7) The function `GraphicDisplay_clearPixel()` very efficiently clears a given pixel at (x, y) by setting it to the blank color specified in `GraphicDisplay_init()`.

NOTE: The pixels are set/cleared only in memory and do not appear on the screen until the call to `GraphicDisplay_redraw()`.

(8) The function `GraphicDisplay_redraw()` re-draws the current state of the in-memory bitmap on the screen.

NOTE: You can use the function **`GraphicDisplay_clear()`** to clear the entire internal device-independent bitmap by setting all the pixels of the to the blank color.

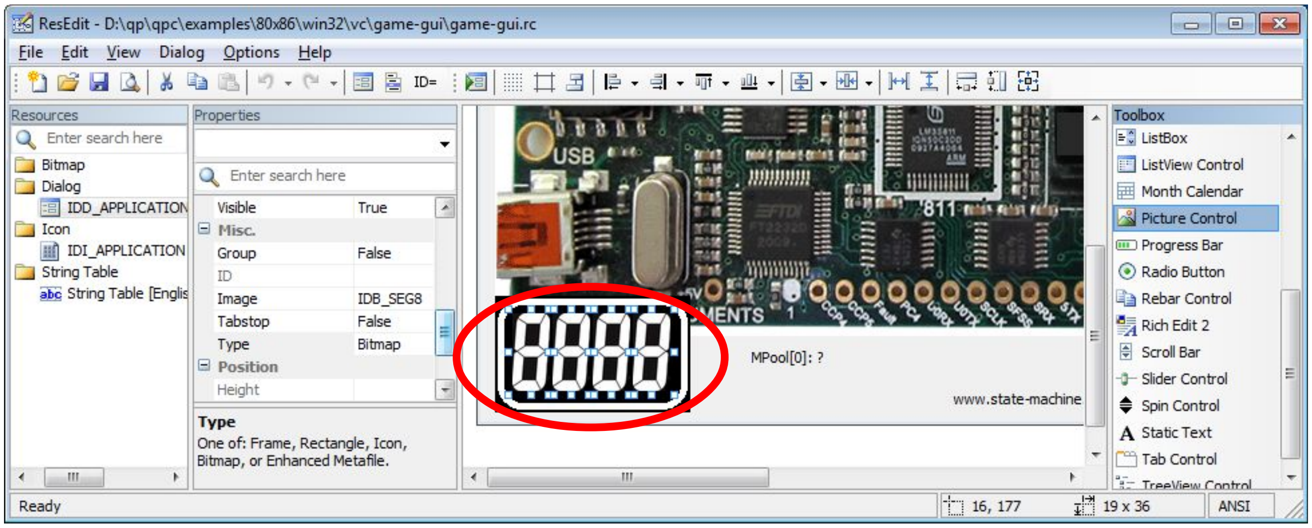
5.5.5 Segment displays

The QP-Win32 port provides also a generic facility for rendering segment displays, such as found in digital watches, calculators, thermostats, microwave ovens, washing machines, digital meters, etc.

The chosen implementation is based on selective displaying bitmaps representing the various states of the related groups of segments. In this design, you need to provide all the bitmaps for all states of segment groups that you anticipate in your application, but the advantage is that the bitmaps can be very realistic and match exactly your specific display.

To add a segment display to the dialog box in ResEdit, you add all the bitmaps for the segments and a "Picture Control" for each segment group. You also need to place the Picture Controls at the exact locations of the segments. Please note that you add one Picture Control for the whole group of segments not for every individual segment. For example, as illustrated in Figure 11, a single picture control represents a whole 7-segment digit. Also, as you keep adding the segment groups to the dialog box, you don't need to make sure that the resource IDs assigned to them are consecutive or in any specific order. The provided implementation of the Segment Display does not require any specific order.

Figure 11: Adding a segment display in ResEdit resource editor



Visible	True
Misc.	
Group	False
ID	
Image	IDB_SEG8
Tabstop	False
Type	Bitmap
Position	
Height	

Type
One of: Frame, Rectangle, Icon, Bitmap, or Enhanced Metafile.

NOTE: The design of a segment display requires quite precise placement of the bitmaps. To achieve the **fine control of placement in ResEdit,** you need to press **Ctrl key** and use the arrow keys to nudge the control in the desired direction. Without the Ctrl key, ResEdit places the control at coarse grid locations.

The segment display requires also some code. which is illustrated in Listing 13.

Listing 13: Snippets from the bsp.c file pertaining to segment display

```
(1) static SegmentDisplay   l_scoreBoard;      /* segment display for the score */
    /*..................................................................*/
    static LRESULT CALLBACK WndProc(HWND hWnd, UINT iMsg,
                                    WPARAM wParam, LPARAM lParam)
    {
        switch (iMsg) {
            /* Perform initialization after all child windows have been created */
            case WM_INITDIALOG: {
                . . .
(2)             SegmentDisplay_init(&l_scoreBoard,
                                    4U,          /* 4 "segments" (digits 0-3) */
                                    10U);       /* 10 bitmaps (for 0-9 states) */
(3)             SegmentDisplay_initSegment(&l_scoreBoard,
                     0U, GetDlgItem(hWnd, IDC_SEG0));
(4)             SegmentDisplay_initSegment(&l_scoreBoard,
                     1U, GetDlgItem(hWnd, IDC_SEG1));
(5)             SegmentDisplay_initSegment(&l_scoreBoard,
                     2U, GetDlgItem(hWnd, IDC_SEG2));
(6)             SegmentDisplay_initSegment(&l_scoreBoard,
                     3U, GetDlgItem(hWnd, IDC_SEG3));
(7)             SegmentDisplay_initBitmap(&l_scoreBoard,
                     0U, LoadBitmap(l_hInst, MAKEINTRESOURCE(IDB_SEG0)));
(8)             SegmentDisplay_initBitmap(&l_scoreBoard,
                         1U, LoadBitmap(l_hInst, MAKEINTRESOURCE(IDB_SEG1)));
                    . . .
 (9)                SegmentDisplay_initBitmap(&l_scoreBoard,
                         9U, LoadBitmap(l_hInst, MAKEINTRESOURCE(IDB_SEG9)));
      . . .
      }
      /*..........................................................................*/
      void BSP_updateScore(uint16_t score) {
          /* update the score in the l_scoreBoard SegmentDisplay */
(10)      SegmentDisplay_setSegment(&l_scoreBoard, 0U, (UINT)(score % 10U));
          score /= 10U;
(11)      SegmentDisplay_setSegment(&l_scoreBoard, 1U, (UINT)(score % 10U));
          score /= 10U;
(12)      SegmentDisplay_setSegment(&l_scoreBoard, 2U, (UINT)(score % 10U));
          score /= 10U;
(13)      SegmentDisplay_setSegment(&l_scoreBoard, 3U, (UINT)(score % 10U));
      }
```

(1) You need to provide an instance of the `SegmentDisplay struct` for each segment display you have added to the Dialog box..

(2) You need to call `SegmentDisplay_init()` in the `WM_INITDIALOG` message handler with the total number of segment groups and the total number of bitmaps associated with the display. For instance, the 4-digit `l_scoreBoard` display used in the game example uses 4 segment groups and 10 bitmaps (for digits 0..9).

(3-6) For each segment group, you need to call `SegmentDisplay_initSegment()` with the index of the segment group and the dialog control corresponding to this segment group. This way, you establish a consecutive numbering of segments (by the index number).

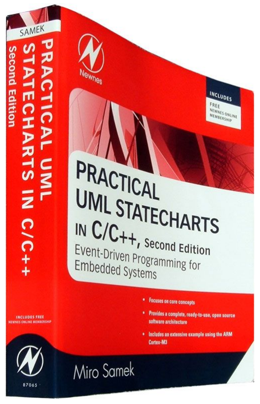
(7-9) For each bitmap, you need to call `SegmentDisplay_initBitmap()` with the index of the bitmap and the bitmap itself loaded from the resource pool. This way, you establish a consecutive numbering of bitmaps (by the index number).

(10-13) You change the bitmaps in the segments by calling `SegmentDisplay_setSegment()` with the index of the segment and index of the bitmap.

5.5.6 LEDs

LEDs can be efficiently handled as segment displays with just one "segment group" (the LED itself) and two bitmaps ("off" and "on" states of the bitmap). This way, it is also very easy to build groups of LEDs and multi-color LEDs just by increasing the number of "segment groups" and bitmaps for each color.

6 Related Documents and References

Document		Location
	"Practical UML Statecharts in C/C++, Second Edition" [PSiCC2], Miro Samek, Newnes, 2008	ISBN-13: 978-0-7506-8706-5 Available from most online book retailers, such as Amazon.com. See also: http://www.state-machine.com/psicc2.htm
[AN-DPP] "Application Note: Dining Philosopher Problem Application", Quantum Leaps, 2008		http://www.state-machine.com/resources/AN_DPP.pdf
[AN-PELICAN] "Application Note: PEDestrian LIght CONtrolled (PELICAN) Crossing Application", Quantum Leaps, 2008		http://www.state-machine.com/resources/AN_PELICAN.pdf
[Pezold] "Programming Windows", Charles Petzold and Paul Yao, Microsoft Press		Available in most bookstores
[QP-Ref]"QP/C++ Reference Manual", Quantum Leaps, LLC, 2011		http://www.state-machine.com/doxygen/qpcpp/
[Q_SPY-Ref] "Q_SPY Host Application Reference Manual", Quantum Leaps, 2011		http://www.state-machine.com/doxygen/qspy
Free QM graphical modeling and code generation tool, Quantum Leaps, 2011		http://www.state-machine.com/qm

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