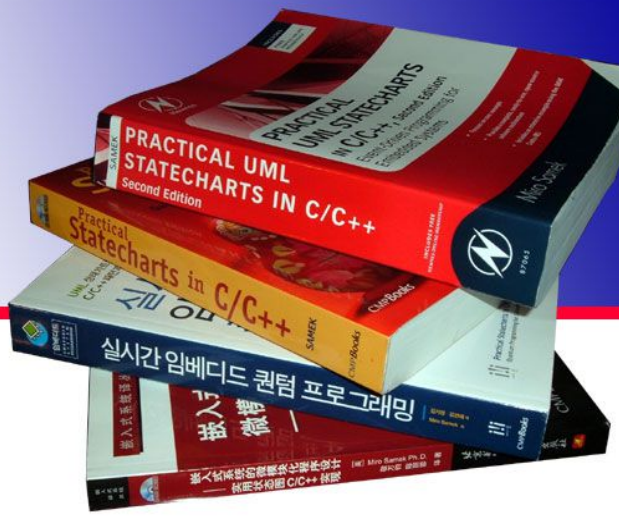
Quantum® LeaPs
innovating embedded systems




# Application Note
# QP/C++™ and the Qt™ GUI Framework


**Document Revision K**
**May 2014**


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

# Table of Contents

# 1 Introduction

This document describes how to use the **QP/C++™** active object framework version **5.3.0** (or higher) with the **Qt** cross-platform application and GUI framework version **5.1.1** (or higher), which can be interesting for at least two reasons. First, you might use QP/C++ to build highly modular, well structured, **multithreaded** desktop or mobile Qt applications based on the concept of **active objects** (a.k.a. actors). In this use case, QP/C++ complements Qt by providing the high-level structure, while Qt renders the GUI and provides various services. The active object computing model underlying QP raises the level of abstraction and provides a **more productive** architecture, which is safer to use, more efficient, and easier to understand than the low-level programming with the low-level `QThread` and `QWaitCondition` classes of Qt, which are just thin wrappers around P-thread (POSIX threads API).

The QP-Qt integration can be also useful for rapid prototyping (virtual prototyping), simulation, and testing of embedded software on the desktop, including building realistic user interfaces consisting of buttons, knobs, LEDs, dials, and LCD displays (both segmented and graphical). 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.

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

## 1.1 About Qt

**Qt** is a comprehensive C++ application development framework for creating cross-platform GUI applications using a "write once, compile everywhere" approach. Qt lets programmers use a single source tree for applications that run on Windows, Linux, and Mac OS X as well as mobile devices. The Qt libraries and tools are also part of Qt/Embedded Linux—a product that provides its own window system on top of embedded Linux.

Starting from version 4.6, Qt provides its own State Machine Framework. However, the Qt state machines are designed very traditionally, where states and transitions are mapped to objects, which need to be instantiated and then inter-connected in complicated ways using the Qt's signal/slot mechanism as well as Qt properties. The state machine code in this approach is very fragmented into state machine objects, transition objects, slot functions for every action, and a lot of "plumbing" code cross-referencing all these elements. The result is that the overall structure of the state machine is very difficult to see from the code and any change to the state machine structure has ripple effects across many objects. This defeats a lot of benefits of using state machines in the first place, as the main purpose of state machines is the clear and highly regular code structure.

## 1.2 About QP/C++™

**QP/C++™** is a lightweight, open source, active object framework for building responsive and modular real-time embedded applications as systems of cooperating, event-driven active objects (actors). 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).

In contrast to the state machine implementation of Qt, the QP/C++ framework represents hierarchical state machines as pure code. States are represented as state-handler functions with transitions as switch-case statements within these state-handler functions. The representation is very readable, concise, and 100% traceable from design, meaning that every state machine element has a single, unique representation without any repetitions. Also in contrast to Qt, the QP code can be generated automatically from state machine diagrams drawn in the free QM modeling tool.

## 1.3 About QM™

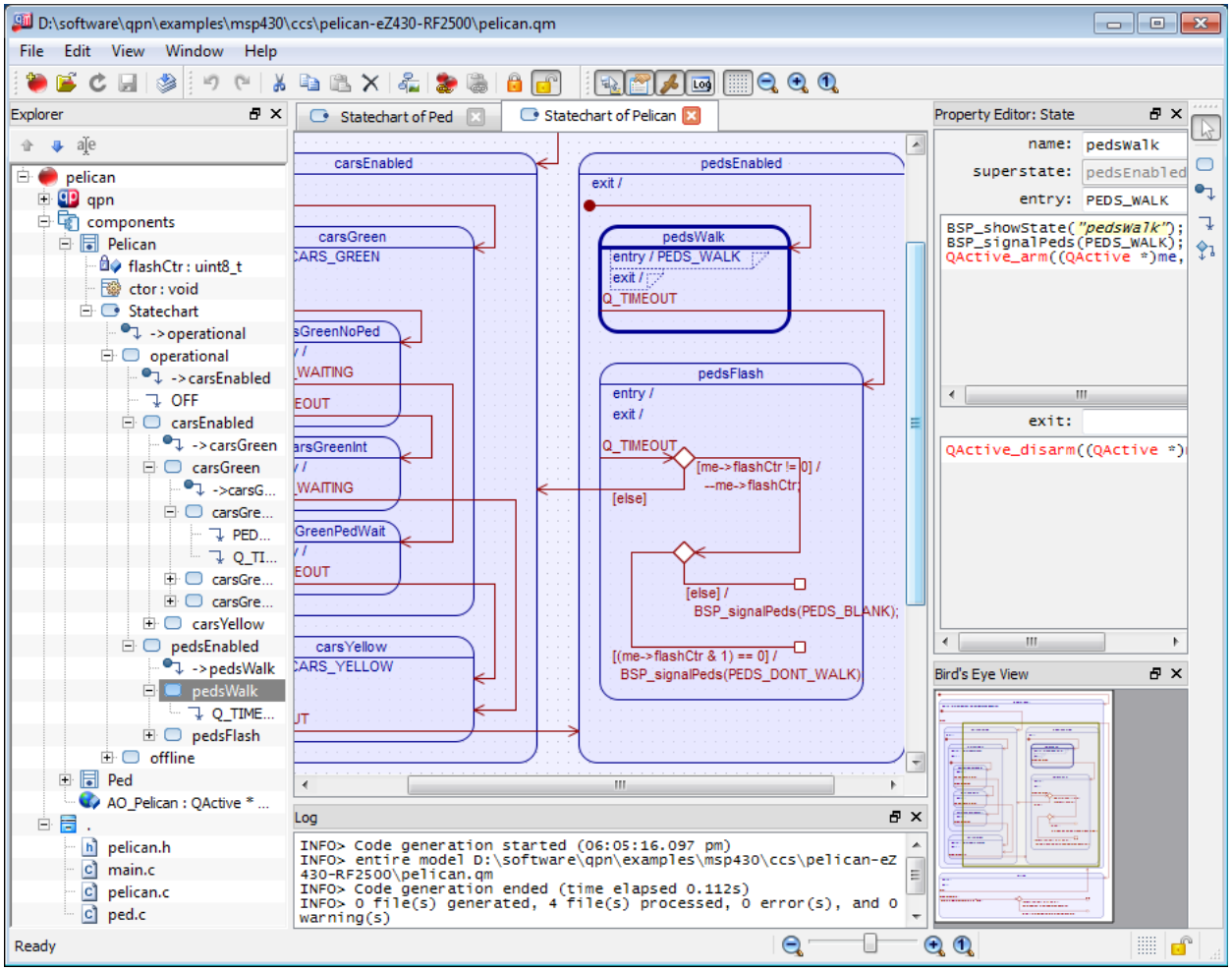
**QM™** (QP™ Modeler) is a free, cross-platform, graphical UML modeling tool for designing and implementing real-time embedded applications based on the QP™ state machine frameworks. QM™ itself is based on the Qt framework and therefore runs naively on Windows, Linux, and Mac OS X.

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

The code accompanying this App Note contains three application examples: the Dining Philosopher Problem [AN-DPP], the PEdestrian LIght CONtrolled [AN-PELICAN] crossing, and the "Fly 'n' Shoot" game simulation for the EK-LM3S811 board (see Chapter 1 in [PSiCC2] all modeled with QM.

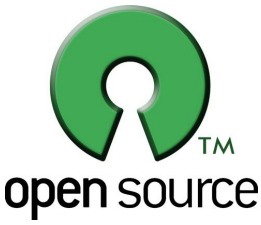
**NOTE:** The provided QM model files assume QM version **3.1.3** or higher.

**Figure 1: The PELICAN example model opened in the QM™ modeling tool**



## 1.4 Licensing 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

## 1.5 Licensing QM™

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

# 2 Getting Started

This section describes how to install, execute, build, and debug QP/Qt applications.

## 2.1 Installing Qt

This Application Note assumes that you have downloaded and installed the Qt framework. Perhaps the best way of getting started with Qt is to download the open source Qt SDK (Software Development Kit) from qt-project.org. The Qt SDK includes pre-built Qt libraries, header files, demos, examples, as well as the Qt Creator development environment and the GNU compiler (in the Windows version).

> **NOTE:** The provided examples work with any version of Qt 5.x, including open source (LGPL) versions as well as commercial versions of Qt. Also, the provided Qt project files can be used with any compiler chain or IDE supporting Qt development can be used (e.g., Visual Studio, Qt Creator, or simple command line).

## 2.2 Installing QP/C++ Baseline Code

The Generally Available distribution of the QP/C++ framework is available from SourceForge at: sourceforge.net/projects/qpc/files/QP-Cpp/. QP/C++ is distributed in a ZIP archive (e.g., `qpcpp_5.3.0.zip`). You can unzip the QP/C++ archive into any location on your hard drive, but you need to define an environment variable QPCPP to point to the QP/C++ 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 | Windows Example | Linux Example |
|---|---|---|---|
| QP/C++ framework | `QPCPP` | `C:\qp\qpcpp` | `~/qpcpp` |
| Qtools collection | `QTOOLS` | `C:\tools\qtools` | `/etc/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-Qt integration.**
**Qt project files are underlined**

```
qpcpp/                      - QP-root directory for QP/C++
  +-include/                - QP public include files
  | +-qassert.h             - Assertions platform-independent public include
  | +-qep.h                 - QEP event processor interface
  | +-. . .
  |
  +-ports/                  - QP ports
  | +-qt/                   - ports to Qt
  | | +-mingw/              - MinGW compiler (GNU on Windows)
  | | | +-debug/            - Debug build
  | | | | +-libqp.a         - QP library
  | | | +-release/          - Release build
  | | | | +-libqp.a         - QP library
  | | | +-qp.pro            - Qt project file to build the QP library
  | | | +-qep_port.h        - QEP platform-dependent public include
```

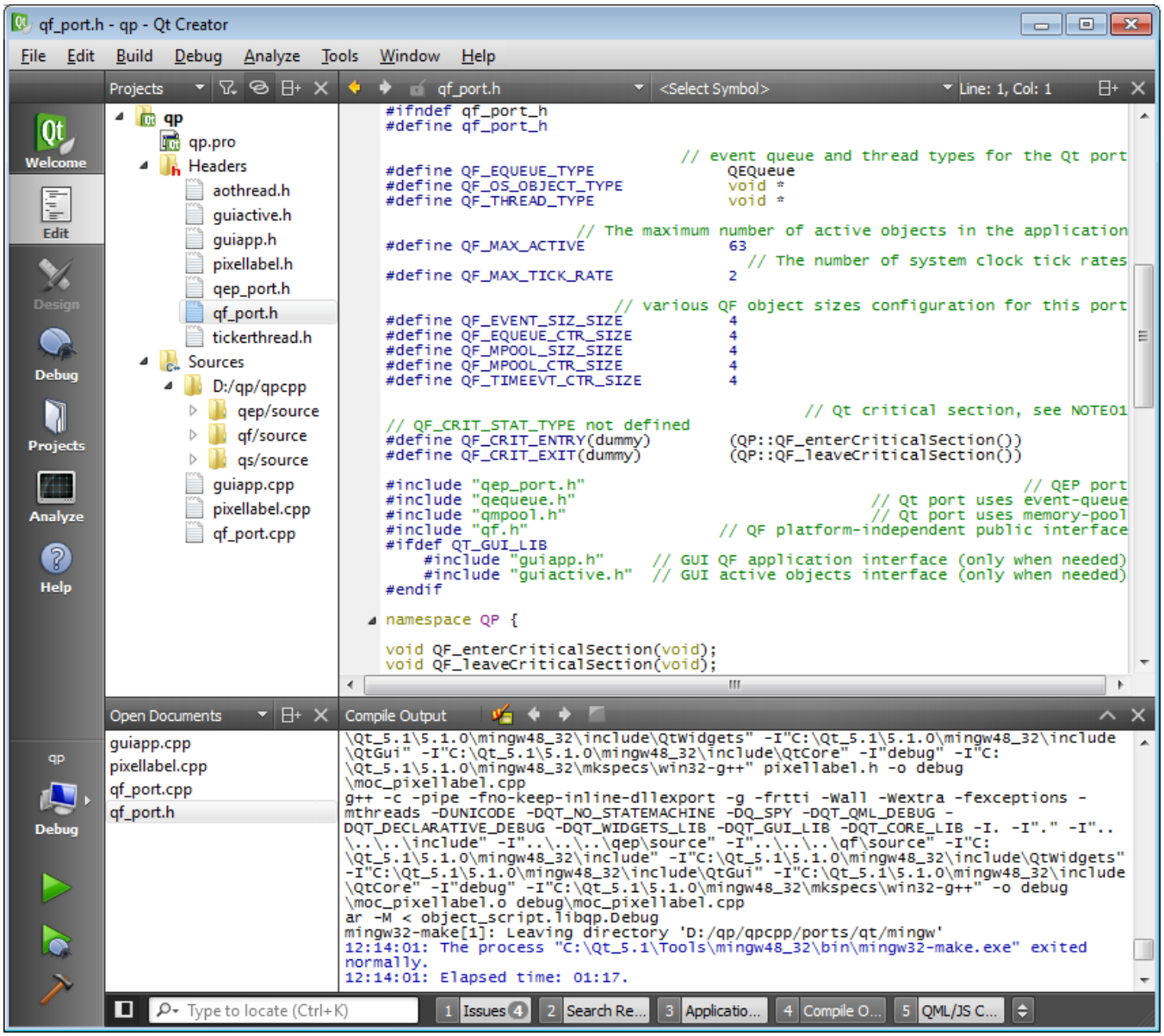
```
| | | +-qf_port.h         - QF platform-dependent public include
| | | +-qf_port.cpp       - QF platform-dependent implementation
| | | +-qp_app.h          - QPApp subclass of Qt class Qapplication to run QP
| | | +-qs_port.h         - QS platform-dependent public include
| | | +-pixellabel.h      - PixelLabel utility class for drawing pixel displays
| | | +-pixellabel.cpp    - PixelLabel utility class implementation
|
+-examples/               - subdirectory containing the QP example files
| +-qt/                   - ports to Qt
| | +-mingw/              - MinGW compiler (GNU on Windows)
| | | +-dpp/              - Dining Philosophers Problem example console application
| | | | +-bsp.h           - Board Support Package header file for DPP
| | | | +-dpp.qm          - QM model file for the DPP example
| | | | +-dpp.pro         - Qt project file for building the DPP example
| | | | +-dpp.h           - the DPP header file
| | | | +-bsp.cpp         - Board Support Package implementation for DPP
| | | | +-main.cpp        - the main function for DPP
| | | | +-phio.c          - the Philosopher active objects
| | | | +-table.c         - the Table active object
| | | |
| | | +-dpp-gui/          - Dining Philosophers Problem example GUI application
| | | | +-debug/          - directory containing the Debug build
| | | | +-release/        - directory containing the Release build
| | | | +-dpp.qm          - QM model file for the DPP example
| | | | +-dpp-gui.pro     - Qt project file for building the DPP-GUI example
| | | | +-gui.h           - Qt GUI header header file for DPP
| | | | +-gui.cpp         - Qt GUI header implementation for DPP
| | | | +-gui.ui          - Qt GUI form file file for DPP
| | | | +-ui_gui.h        - Qt GUI header file generated by Qt Designer
| | | | +-. . . .
| | | |
| | | +-game-gui/         - "Fly 'n' Shoot" game example for Qt
| | | | +-debug/          - directory containing the Debug build
| | | | +-release/        - directory containing the Release build
| | | | +-game.h          - the game header file
| | | | +-game.qm         - QM model file for the game example
| | | | +-game-gui.pro    - Qt project file for building the GAME-GUI example
| | | | +-bsp.h           - Board Support Package header file for the game
| | | | +-bsp.cpp         - Board Support Package implementation for the game
| | | | +-gui.h           - Qt GUI header header file for the game
| | | | +-gui.cpp         - Qt GUI header implementation for the game
| | | | +-gui.ui          - Qt GUI form file file for the game
| | | | +-main.cpp        - the main function for the game
| | | | +-missile.c       - the Missile active objects
| | | | +-ship.c          - the Ship active object
| | | | +-tunnel.c        - the Tunnel active object
| | | | +-ui_gui.h        - Qt GUI header file generated by Qt Designer
| | | | +-. . . .
| | | |
| | | +-pelican-gui/      - PEDestrian LIght CONtrolled crossing example with GUI
| | | | +-debug/          - directory containing the Debug build
| | | | +-release/        - directory containing the Release build
| | | | +-game.h          - the game header file
| | | | +-pelican.qm      - QM model file for the PELICAN example
| | | | +-pelican-gui.pro- Qt project file for building the PELICAN-GUI example
| | | | +-. . . .
```

## 2.3 Building the QP/C++ Library

The QP/C++ framework is deployed as a library that you statically link to your application. This section describes steps you need to take to rebuild the QP/C++ library for Qt.

The QP/C++ code distribution contains the `qp.pro` project file for building the QP/C++ library located in the `qpcpp/ports/qt/mingw/` directory. For example, to build the QP/C++ library with the Qt Creator IDE, you launch Qt Creator and open the `qp.pro` project file (see Figure 3). Next, you build the project by clicking on the "hammer" tool. To build the release configuration, click on the Projects perspective, choose the build configuration from the drop down list, and re-build the project. These builds generate the QP libraries `qibqp.a` in the sub-directories `debug/` and `release/`, respectively.

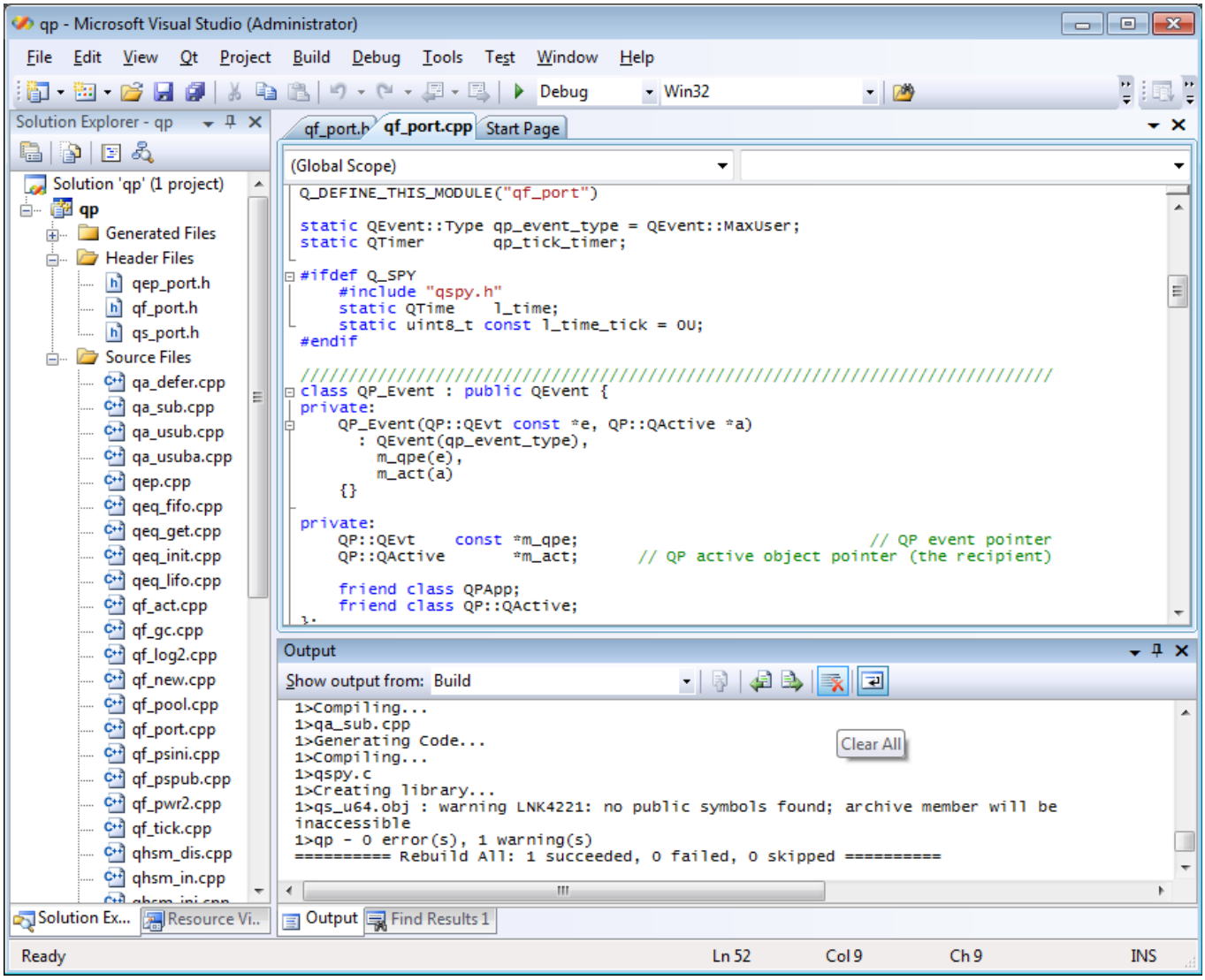
**Figure 2: The qp.pro project in the Qt Creator IDE**



The `qp.pro` project file can be also used with other tools. For example, Figure 4 shows the QP project file open in Microsoft Visual Studio augmented with the Qt Add-In.

> **NOTE:** If you decide to use Visual Studio, it is recommended to rename the sub-directory `mingw/` into `vc/` for example, to indicate that the libraries are build with Visual C++ compiler.

**Figure 3: The qp.pro project in the Visual Studio IDE**



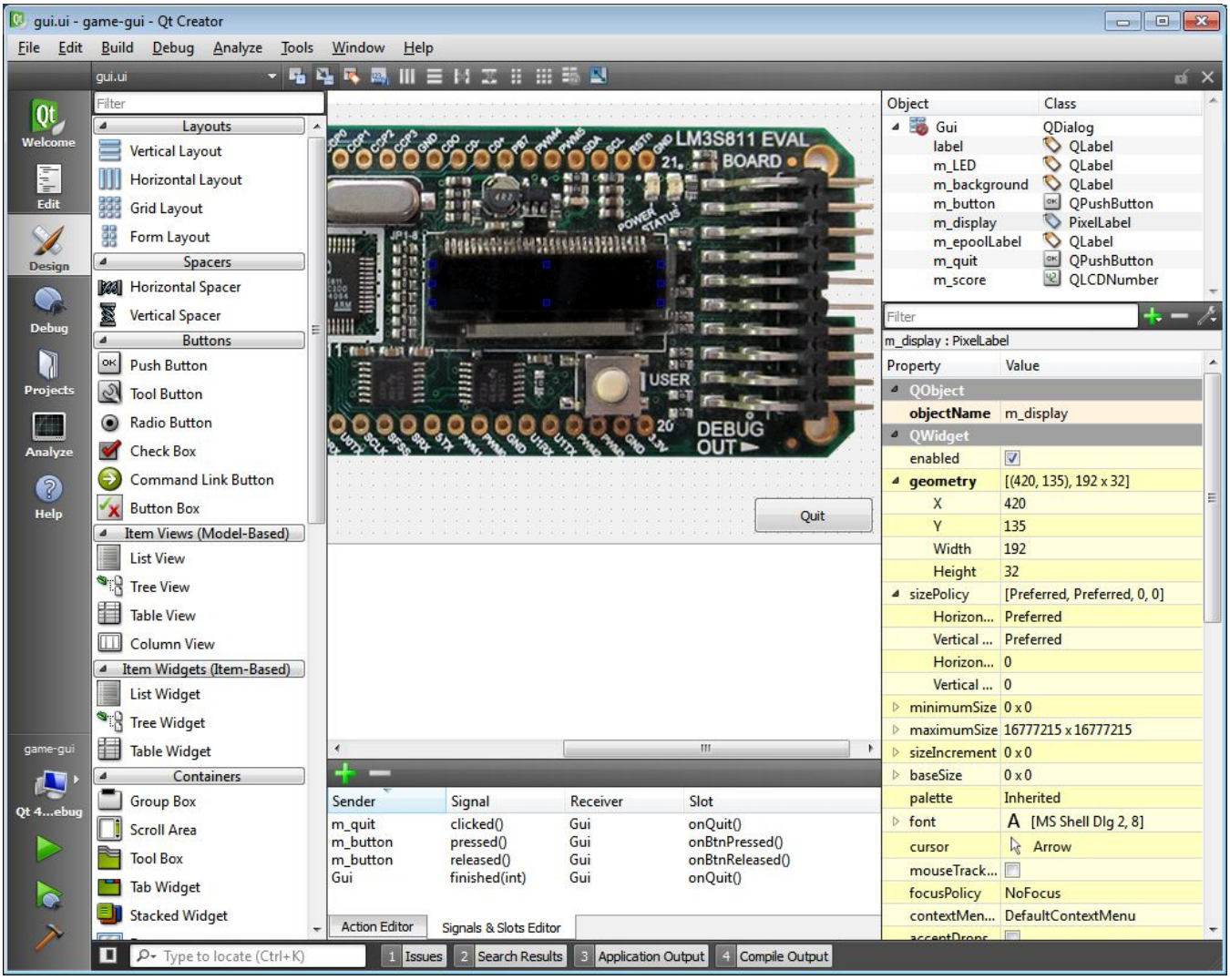
## 2.4 Building the Examples

The QP examples directory (`qcpp/examples/qt/mingw/`, see Listing 1) contains the Qt project files for building the examples.

For example, to build the DPP example, you load the project `qpcpp/examples/qt/mingw/dpp-gui/dpp-gui.pro` into Qt Creator (or Visual Studio) and start the build. Similarly, to build the PELCIAN crossing example, you use the project `qcpp/examples/qt/mingw/pelican/pelican.pro`.

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

**Figure 4: The game-gui.pro project in the Qt Creator IDE**



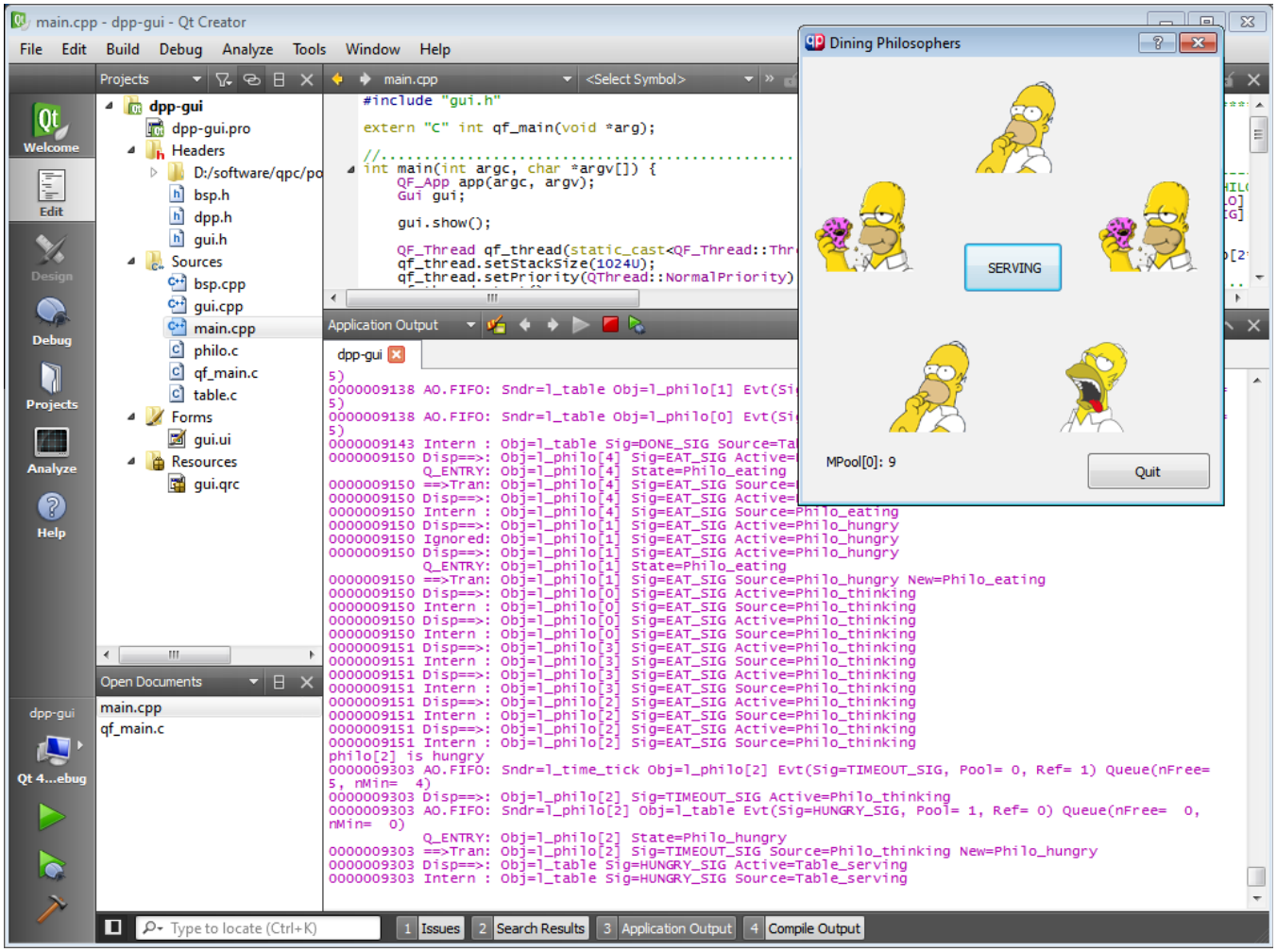
## 2.5 Running the DPP Example

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 SERVING 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 (debug configuration) running in Qt Creator.
Note the human-readable QSPY output in the Application Output window.**



## 2.6 Running the "Fly 'n' Shoot" Game Simulation

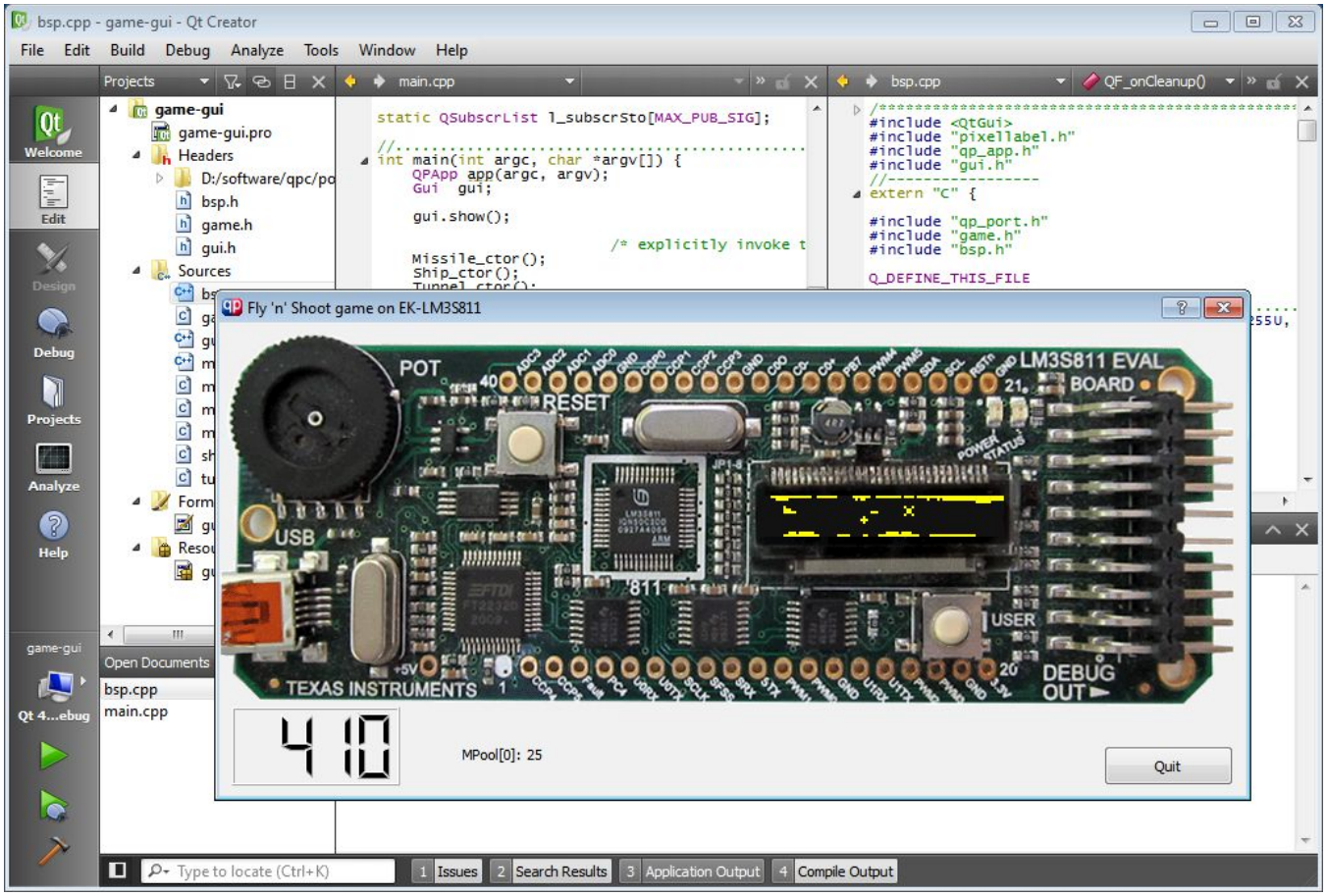
The "Fly 'n' Shoot" game example demonstrates simulating embedded target (the EK-LM3S811 board in this case) with Qt.

> **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 6). The most interesting aspect of this simulation is that it executes exactly the same code as the real embedded board, except the BSP, which is implemented with Qt.

Once you launch the game simulation, it starts with flashing the "Press Button" text on the LCD 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.

**Figure 6: The"Fly 'n' Shoot" game launched from the Qt Creator.**



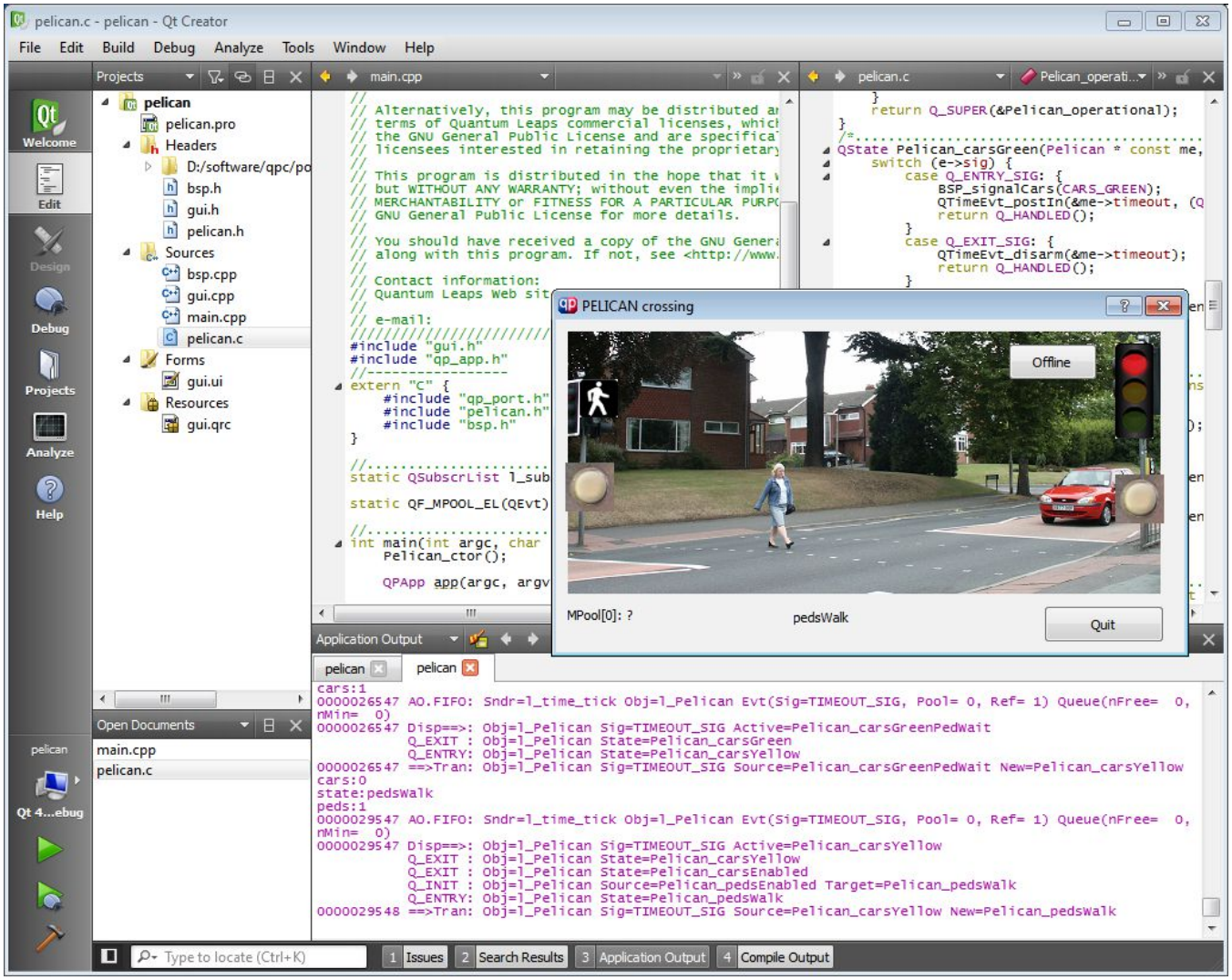
## 2.7 Running the PELICAN Crossing Example

The PEdestrian LIght CONtrolled (PELICAN) crossing example demonstrates non-trivial hierarchical state machine of a traffic light controller.

> **NOTE:** The design and implementation of the PELICAN crossing application, including state machine design, is described in the Application Note "PELICAN crossing" (see [AN-PELICAN]).

The PELICAN crossing example application outputs the QSPY software trace data directly to the Qt console (see Figure 7). When the ON button is depressed (but not released) the application enters the "offline" state, in which the traffic lights for cars flash the red signal and the pedestrian light flashes the "DON'T WALK" signal. After releasing the ON button, the application resumes normal execution. The application is terminated either by clicking the Close button.

**Figure 7: The PELICAN crossing example (debug configuration) running in Qt Creator. Note the human-readable QSPY output in the Application Output window.**



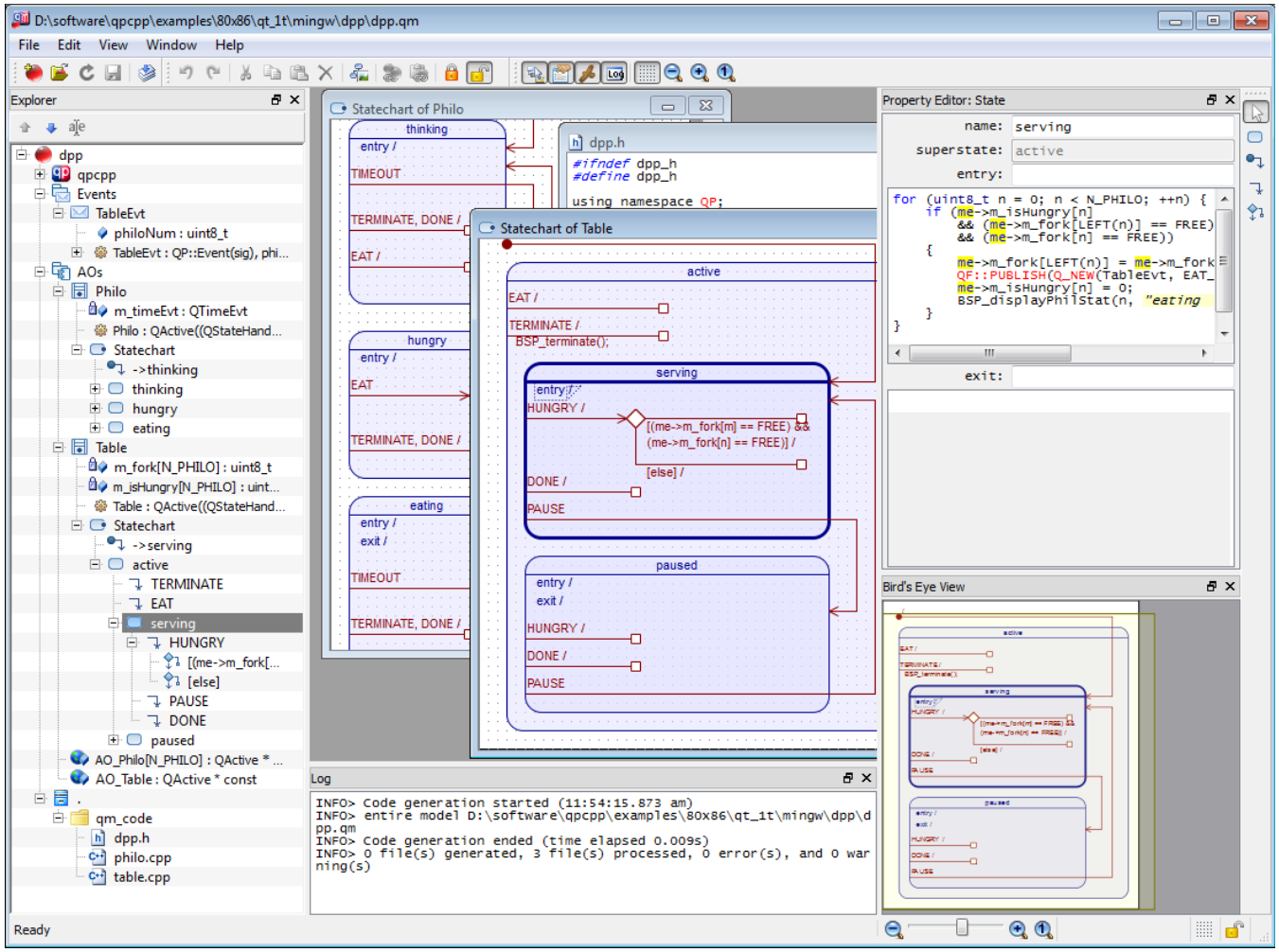
## 2.8 Generating the QP Application Code with the QM™ Modeling Tool

Both examples (DPP and the PELICAN crossing) 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., `qpcpp/examples/qt/-mingw/dpp/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 8: The DPP example in QM.**



# 3 The structure of the QP-Qt integration

Figure 9 shows how QP/C++ and Qt frameworks are integrated with each other. The dominant framework is QP/C++, which runs all active objects in the system and handles all event exchanges among them. The QP framework executes all non-GUI active objects in their own **thread** context provided by the `QThread` class of Qt. The special GUI active object encapsulates most of the Qt framework. This GUI active object runs in the context of the **Qt event loop** provided by the `QApplication` Qt class.

> **NOTE:** Active objects provide a more productive architecture, which is safer to use, more efficient, and easier to understand than directly fiddling with such low-level Qt classes as `QThread` or `QWaitCondition`.

The queuing and dispatching of QP events (subclasses of `QP::QEvt`) to non-GUI active objects is handled by the QP framework in the standard way. The **GUI active object** handles event posting in a special way, because it uses the Qt event queue. The overridden `post()` operation of the GUI active object wraps the QP event (subclass of `QP::QEvt`) in a Qt event (subclass of `Qt::QEvent`) and posts it to the Qt event loop by means of the **thread-safe** `QCoreApplication::postEvent()` call.

**Figure 9: The structure of the QP™ integration with Qt**

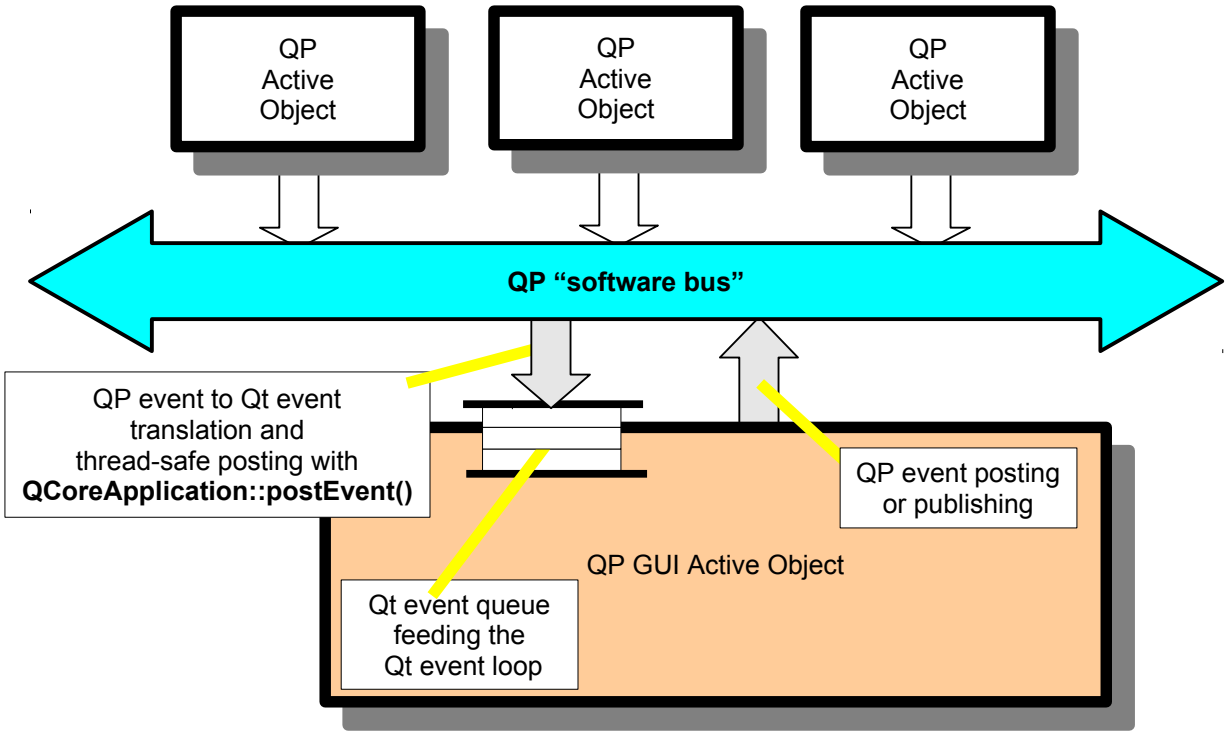
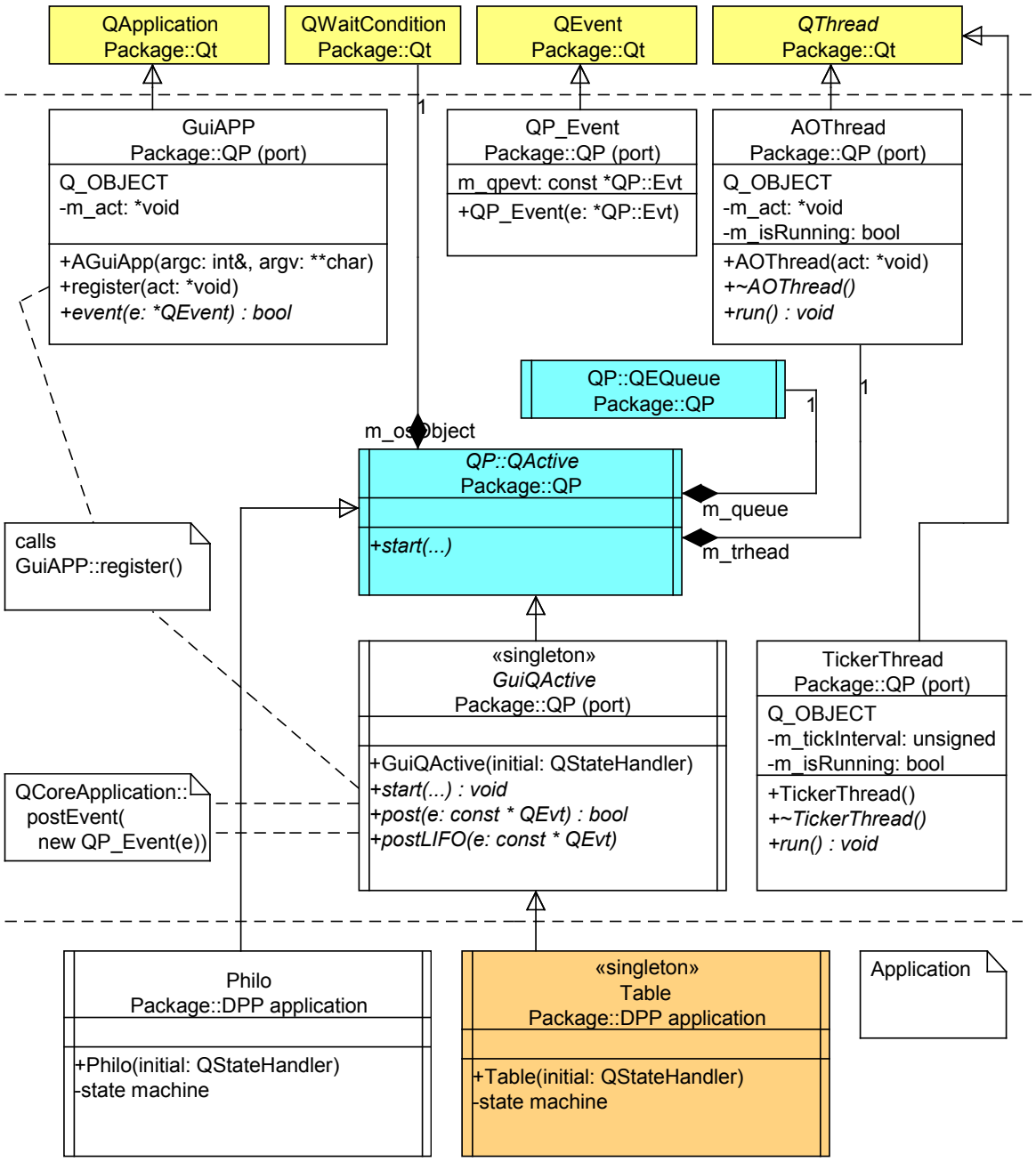



Figure 10 shows the class relationships of the QP port to Qt. The Qt classes are shown in pale-yellow. The QP base classes are shown in light-cyan. The GUI active object is shown in light-orange. All non-GUI active objects derive from `QP::QActive` base class. These active objects use the standard QP event queue (`QP::QEQueue`) and run in the thread context provided by the `QThread` class from Qt. The `GuiQActive` singleton provides the specialized version of the QP active object for the GUI. This class overrides the `post()/postLIFO()` operations to perform translation of QP events to Qt events and thread-safe posting to the Qt event queue by means of the `QCoreApplication::postEvent()` call.

**Figure 10: The class relationships of the QP integration with Qt**




**NOTE:** For clarity the class diagram shown in Figure 10 does not show the classes `QP::QMactive` and `GuiQMActive`, but these classes are provided as well to facilitate the use of faster code generation by the QM modeling tool.

## 3.1 The `qep_port.h` header file

Listing 2 shows the `qep_port.h` header file, which is located in the directory `qpcpp\ports\qt\mingw\`. The explanation section immediately following the listing highlights the main points.

**Listing 2: `qep_port.h` header file**

```
    #ifndef qep_port_h
    #define qep_port_h

    // don't define QEvent
(1) #define Q_NQEVENT         1

    // use ctors for dynamic events
(2) #define Q_EVT_CTOR        1

    // provide QEvt virtual destructor
(3) #define Q_EVT_VIRTUAL     1

(4) #include <stdint.h>  // Exact-width types. WG14/N843 C99, Section 7.18.1.1
(5) #include "qep.h"     // QEP platform-independent public interface

    #endif // qep_port_h
```

(1) The provision of the macro `Q_NQEVENT` suppresses the use of the `QEvent` class, which is only defined for backwards compatibility with the earlier versions of QP.

> **NOTE:** Prior to version **4.5.00**, the QP framework used the name `QEvent` for QP event class. This name was changed to `QEvt` to avoid the name conflict with Qt.

(2) The provision of the macro `Q_EVT_CTOR` configures QP to generate a non-default constructor of the `QP::QEvt` class as well as the virtual destructor to prepare this class for polymorphism. Also the macro `Q_NEW()` is defined differently in this case to supply the parameters to the constructor.

(3) The provision of the macro `Q_EVT_VIRTUAL` configures QP to generate the virtual destructor in the `QEvt` base class and the explicit call to the `QEvt::~QEvt()` destructor in the garbage collector. This allows events to contain members that require calling destructors to clean them up, such as `QString`, `QByteArray`, `QVariant`, and other such Qt classes.

(4) The standard header file `<stdint.h>` defines the exact-width integer types.

> **NOTE:** The Microsoft Visual C++ compilers don't provide the standard library `<stdint.h>` file. For this compiler, you can either provide a rudimentary `<stdint.h>` header file with just six types (`uint8_t, int8_t, uint16_t, int16_t, uint32_t, and int32_t`), or you can typedef these six types directly in `qep_port.h`.

(5) As usual, the platform-independent QEP interface `qep.h` must be included in every port..

## 3.2 The `qf_port.h` header file

Listing 3 shows the `qf_port.h` header file for the QP/C++ port to Qt, which is located in the directory `qpcpp\ports\qt\mingw\`. The explanation section immediately following the listing highlights the main points.

**Listing 3: `qf_port.h` header file**

```
     // event queue and thread types for the Qt port
 (1) #define QF_EQUEUE_TYPE              QEQueue
 (2) #define QF_OS_OBJECT_TYPE           QWaitCondition *
 (3) #define QF_THREAD_TYPE              QThread *

     // The maximum number of active objects in the application
     #define QF_MAX_ACTIVE               63

     // The number of system clock tick rates
     #define QF_MAX_TICK_RATE            2
     ~ ~ ~ ~

     // Qt critical section, see NOTE01
 (4) // QF_CRIT_STAT_TYPE not defined
 (5) #define QF_CRIT_ENTRY(dummy)        (QP::QF_enterCriticalSection())
 (6) #define QF_CRIT_EXIT(dummy)         (QP::QF_leaveCriticalSection())

     class QWaitCondition;    // forward declaration
     class QThread;           // forward declaration

     #include "qep_port.h"    // QEP port
     #include "qequeue.h"     // Qt port uses event-queue
     #include "qmpool.h"      // Qt port uses memory-pool
     #include "qf.h"          // QF platform-independent public interface

 (7) #ifdef QT_GUI_LIB
         #include "guiapp.h"    // GUI QF application interface (only when needed)
         #include "guiactive.h" // GUI active objects interface (only when needed)
     #endif

 (8) namespace QP {

     void QF_enterCriticalSection(void);
     void QF_leaveCriticalSection(void);
 (9) void QF_setTickRate(unsigned ticksPerSec);  // set clock tick rate
(10) void QF_onClockTick(void);  // clock tick callback (provided in the app)

(11) #ifdef Q_SPY
(12) void QS_onEvent(void);
     #endif
     ~ ~ ~ ~ ~
     // undefine the conflicting Q_ASSERT definition from Qt
     #ifdef Q_ASSERT
(13)    #undef Q_ASSERT
     #endif
```

(1) The macro `QF_EQUEUE_TYPE` specifies the event queue class for QP active objects. The QP-Qt integration uses the built-in QP event queue `QEQueue`.

(2) The macro `QF_OS_OBJECT_TYPE` specifies the operating system object used to block the built-in QP event queue `QEQueue`. The QP-Qt integration uses the `QWaitCondition` class from Qt, but to avoid introducing dependency on the bulky Qt header, only a pointer is used (allocated dynamically in the `QActive::start()`).

(3) The macro `QF_THREAD_TYPE` specifies the thread class for the private thread of QP active object. The QP-Qt integration uses the `QThread` class from Qt, but to avoid introducing dependency on the bulky Qt header, only a pointer is used (allocated dynamically in the `QActive::start()`).

(4) The QP-Qt integration uses the simple non-nesting critical section, so the macro `QF_CRIT_STAT_TYPE` is not used.

(5-6) The QF critical section is defined as a pair of functions, which lock and unlock a `QMutex` object.

(7) The `QT_GUI_LIB` macro must be defined for GUI applications (typically on the command line). This macro controls inclusion of the GUI-specific classes `GuiActive` and `GuiAPP` (see Figure 10).

**NOTE:** The Qt header files for GUI API are bulky and the macro `QT_GUI_LIB` avoids including them in every application module for non-GUI applications.

(8) To avoid any name conflicts, the namespace QP is used around all functions introduced in this port.

(9) The function `QF_setTickRate()` allows you to change the default system clock rate (of 10Hz).

(10) The function `QF_onClockTick()` is an-application-level callback invoked for every system clock tick. At the minimum, this function needs to call `QF::tickX()`, but can perform other processing as well.

(11-12) For the Q_SPY build configuration, the function `QS_onEvent()` is an-application-level callback invoked after every QS event to output the data in real-time.

(13) The Qt macro `Q_ASSERT` is undefined, because it conflicts with the QP macro with the same name.

## 3.3 The `guiapp.h` header file

As described in Section 3, the special GUI active object runs in the context of the Qt application. The header file `guiapp.h` declares a specialized `QApplication` subclass that overrides the `QApplication::event()` operation.

**Listing 4: `guiapp.h` header file**

```
    #ifndef guiapp_h
    #define guiapp_h

    #include <QApplication>

    namespace QP {

(1) class GuiApp : public QApplication {
(2)    Q_OBJECT

    public:
        GuiApp(int &argc, char **argv);
(3)     void registerAct(void *act);
(4)     virtual bool event(QEvent *e);

    private:
(5)     void *m_act; // GUI active object associated with this GUI application
    };

    } // namespace QP

    #endif  // guiapp_h
```

(1) The class `GuiApp` derives from the Qt class `QApplication` to specialize the event processing.

(2) The class `GuiApp` extends (indirectly) Qt base class `QObject` and requires the Qt meta-object system support to work.

(3) The function `registerAct()` registers the GUI active object with the Qt application.

(4) The event handling of the Qt base class `QApplication` is overridden to handle QP events.

(5) The member `m_act` remembers the GUI active object associated with this GUI application.

## 3.4 The `guiapp.cpp` implementation file

Listing 5 shows the most important elements of the `GuiApp` class implementation. This class specializes the Qt `QApplication` class to handle the QP events injected into the Qt event queue.

**Listing 5: `guiapp.cpp` implemenation file**

```
    namespace QP {

(1) static QEvent::Type l_qp_event_type = QEvent::MaxUser;

    //.....................................................................
(2) class QP_Event : public QEvent {  // Qt event!
    public:
(3)     QP_Event(QP::QEvt const *e)
          : QEvent(l_qp_event_type),
            m_qpevt(e)
          {}

(4)     QP::QEvt const *m_qpevt; // QP event pointer
    };

    //.....................................................................
(5) GuiApp::GuiApp(int &argc, char **argv)
    : QApplication(argc, argv),
       m_act(0)
    {
(6)     l_qp_event_type = static_cast<QEvent::Type>(QEvent::registerEventType());
    }
    //.....................................................................
    void GuiApp::registerAct(void *act) {
(7)     Q_REQUIRE(m_act == 0); // the GUI active object must not be registered
        m_act = act;
    }
    //.....................................................................
(8) bool GuiApp::event(QEvent *e) {
(9)     if (e->type() == l_qp_event_type) {
(10)        QP::QEvt const *qpevt = (static_cast<QP_Event *>(e))->m_qpevt;
(11)        static_cast<QP::QActive *>(m_act)->dispatch(qpevt);  // dispatch to AO
(12)        QP::QF::gc(qpevt);  // garbage collect the QP evt
            return true;        // event recognized and handled
        }
        else {
(13)        return QApplication::event(e); // delegate to the superclass
        }
    }

    //.....................................................................
(14) void GuiQActive::start(uint_fast8_t const prio,
                            QEvt const **qSto, uint_fast16_t /*qLen*/,
                            void * const stkSto, uint_fast16_t const /*stkSize*/,
                            QEvt const * const ie)
    {
        Q_REQUIRE((uf8_0 < prio)
```

```
                    && (prio <= static_cast<uint_fast8_t>(QF_MAX_ACTIVE))
                    && (qSto == (QEvt const **)0)/* does not need per-actor queue */
                    && (stkSto == null_void));      // does not need per-actor stack

        setPrio(prio);   // set the QF priority of this active object
        QF::add_(this);  // make QF aware of this active object
(15)    static_cast<GuiApp *>(QApplication::instance())->registerAct(this);

(16)    this->init(ie);  // execute initial transition (virtual call)
        QS_FLUSH();      // flush the trace buffer to the host
    }
    //.....................................................................
    #ifndef Q_SPY
(17) bool GuiQActive::post_(QEvt const * const e, uint_fast16_t const /*margin*/)
    #else
    bool GuiQActive::post_(QEvt const * const e, uint_fast16_t const /*margin*/,
                           void const * const sender)
    #endif
    {
        QF_CRIT_STAT_
(18)    QF_CRIT_ENTRY_();

        ~ ~ ~
(19)    if (QF_EVT_POOL_ID_(e) != u8_0) { // is it a dynamic event?
(20)        QF_EVT_REF_CTR_INC_(e); // increment the reference counter
        }
(21)    QF_CRIT_EXIT_();

        // QCoreApplication::postEvent() is thread-safe per Qt documentation
(22)    QCoreApplication::postEvent(QApplication::instance(), new QP_Event(e));
        return true;
    }
    //.....................................................................
    void GuiQActive::postLIFO(...) {
        ~ ~ ~ ~
    }
```

(1) A Qt user event type for the QP events is initialized at the top of the user range.

(2) The class `QP_Event` derives from the Qt `QEvent` and provides the "wrapper" around the QP event.

(3) The `QP_Event` constructor saves the `QP::QEvt` pointer in the `m_qpevt` attribute.

(4) The `m_qpevt` attribute stores just a pointer to the QP event (so no copying of the event data is done)

(5-6) The `GuiApp` constructor registers the user event type.

(7) The `registerAct()` operation (register GUI active object) asserts that only a single GUI active object can be registered.

(8) The virtual `event()` operation inherited from the Qt `QApplication` class is overridden to handle QP events.

(9) If the event type is the registered user event type for the QP events, the `event()` operation executes the Run-To-Completion (RTC) step of the active object processing.

(10) The QP event pointer is extracted from the Qt event "wrapper".

(11) The QP event is dispatched to the state machine of the GUI active object.

(12) The QP event is garbage-collected.

(13) In case of a standard Qt event, the event is delegated to the Qt application to be handled in a standard way.

(14) The GUI active object virtual `start()` operation is overridden for this special active object type

(15) The GUI active object is registered with the `GuiApplication`, so that the `GuiApplication` can dispatch QP events to this active object.

(16) The top-most initial transition is triggered in the GUI active object state machine.

(17) The GUI active object post operation is overridden, so that it can "wrap" QP events into Qt events and post them to the Qt application for procesing.

(18-21) A QP critical section is established to increment the reference counter of a dynamic QP event.

(22) The Qt API `QCoreApplication::postEvent()` is used to post the Qt event to the Qt application.

**NOTE:** Per the Qt documentation, `QCoreApplication::postEvent()` is **thread-safe**.

## 3.5 The `qf_port.cpp` source file

Listing 6 shows fragments of the `qf_port.cpp` source file located in the directory `qpcpp\ports\qt\`. The explanation section immediately following the listing highlights the main points.

**Listing 6: `qf_port.cpp` implementation file**

```
    namespace QP {

    //............................................................................
(1) QMutex QF_qtMutex_;

    static TickerThread l_tickerThread;

    //............................................................................
(2) void QF_enterCriticalSection() { QF_qtMutex_.lock();   }
(3) void QF_leaveCriticalSection() { QF_qtMutex_.unlock(); }

    //............................................................................
(4) void AOThread::run() {
(5)     Q_REQUIRE(m_act != static_cast<void *>(0));
(6)     QP::QF::thread_(static_cast<QP::QActive *>(m_act));
    }
    //============================================================================
(7) void TickerThread::run() {
        m_isRunning = true;
        do {
            msleep(m_tickInterval);
(8)         QP::QF_onClockTick();
    #ifdef Q_SPY
            QP::QS_onEvent();
    #endif
        } while (m_isRunning);
```

```
    }

    //............................................................................
 (9) int16_t QF::run(void) {
(10)     onStartup();  // invoke the startup callback

         l_tickerThread.setStackSize(1024U*4U);  // 4KB of stack
(11)     l_tickerThread.start();

         // run the Qt event loop (console or GUI)
(12)     return static_cast<int16_t>(QCoreApplication::instance()->exec());
    }
    //............................................................................
(13) void QF::thread_(QActive *act) {
         QThread::Priority qt_prio = QThread::IdlePriority;
(14)     switch (act->m_prio) {              // remap QF priority to Win32 priority
             case 1:
                 qt_prio = QThread::IdlePriority;
                 break;
             case 2:
                 qt_prio = QThread::LowestPriority;
                 break;
             case 3:
                 qt_prio = QThread::LowPriority;
                 break;
             case (QF_MAX_ACTIVE - 2):
                 qt_prio = QThread::HighPriority;
                 break;
             case (QF_MAX_ACTIVE - 1):
                 qt_prio = QThread::HighestPriority;
                 break;
             case QF_MAX_ACTIVE:
                 qt_prio = QThread::TimeCriticalPriority;
                 break;
             default:
                 qt_prio = QThread::NormalPriority;
                 break;
         }

         AOThread *thread = static_cast<AOThread *>(act->m_thread);
(15)     thread->setPriority(qt_prio);
(16)     thread->m_isRunning = true;
         do {                   // loop until m_thread is cleared in QActive::stop()
(17)         QEvt const *e = act->get_(); // wait for event
(18)         act->dispatch(e); // dispatch to the active object's state machine
(19)         gc(e); // check if the event is garbage, and collect it if so
         } while (thread->m_isRunning);

         QF::remove_(act);
         delete thread;
         delete act->m_osObject;
    }
    //............................................................................
(20) void QF::stop(void) {
         l_tickerThread.m_isRunning = false;
    }
```

```
    //............................................................................
(21) void QF_setTickRate(unsigned ticksPerSec) {
        l_tickerThread.m_tickInterval = 1000U/ticksPerSec;
    }
    //............................................................................
(22) void QActive::start(uint_fast8_t prio,
                         QEvt const **qSto, uint_fast16_t qLen,
                         void *stkSto, uint_fast16_t stkSize,
                         QEvt const *ie)
    {
        Q_REQUIRE(stkSto == static_cast<void *>(0)); // no per-task stack

(23)    m_thread   = new AOThread(this);
(24)    m_osObject = new QWaitCondition;
(25)    m_eQueue.init(qSto, qLen);
        m_prio = prio;

        QF::add_(this); // make QF aware of this active object
(26)    init(ie);       // execute the initial transition

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

        AOThread *thread = static_cast<AOThread *>(m_thread);
(27)    thread->setStackSize(stkSize);
(28)    thread->start();
    }
    //............................................................................
(29) void QActive::stop(void) {
        Q_REQUIRE(m_thread != 0);
(30)    static_cast<AOThread *>(m_thread)->m_isRunning = false;
    }

    } // namespace QP
```

(1) This `QMutex` object is used to implement the QP critical section.

(2-3) The enter/exit critical section implementation function lock and unlock this mutex, respectively.

(4) The Active Object thread `run()` virtual function implements the standard active object event loop.

(5) This precondition ensures that the active object associated with the thread has been properly initialized. This initialization happens in the `QActive::start()` function.

(6) The Active Object thread function implementation delegates to the internal `QF::thread_()` member function. This is necessary to get access to the private data members of the `QP::QActive` class.

(7) The `TickerTherad` is a free-running `QThread` (not an active object) that provides the system clock tick context.

(8) The `TickerTherad` periodically calls the `QF_onClockTick()` callback, which is defined in the application.

(9) The `QF::run()` function executes the whole QP/Qt application.

(10) The `QF::onStartup()` callback is ideal for setting up the system clock tick by calling `QF_setTickRate()`. (see step (21)).

(11) The ticker thread is started.

(12) Finally, the `QF::run()` function exectues the Qt event loop by calling `QCoreApplication::instance()->exec()`. The Qt event loop does not exit until the application quits.

(13) The `QF::thread_()` function implements the event loop of regular QP active objects.

(14) The QP priority is mapped to the Qt thread priority. The policy is to assign the three lowest QP priority levels to Qt thread priorities below NormalPriority, and the three highest QP priority levels to Qt thread priorities above the NormalPriority.

(15) The active object's Qt thread priority is set to the Qt priority determined in the previous step (14).

(16) The thread loop-flag is set to true.

(17-19) The three operations of the active object's Run-to-Completion step are executed.

---

**NOTE:** The `act->get()` operation **blocks** as long as the AO's event queue is empty. This blocking is implemented with the `QWaitCondition` variable associated with each AO.

---

(20) The `QF::stop()` function clears the loop-variable of the ticker thread, which causes termination of this thread.

(21) `QF_setTickRate()` is a convenience function to adjust the system clock tick rate for QP. The ideal place for calling is is the `QF_onStartup()` callback.

---

**NOTE:** The actual ticking rate is determined by the hardware clock interrupt rate and therefore can be really adjusted in integral multiples of the fundamental hardware rate. For example, the hardware clock rate in most 80x86-based systems is set to 100Hz (a tick per 10ms).

---

(22) The `QActive::start()` function starts the QP active object, which is always strongly platform-dependent.

(23) The thread object for the active object is created.

(24) The condition variable object for the AO is created.

(25) The built-in QP event queue for the AO is initialized.

(26) The top-most initial transition of the AO's state machine is triggered.

(27) The stack size of the AO's thread is set.

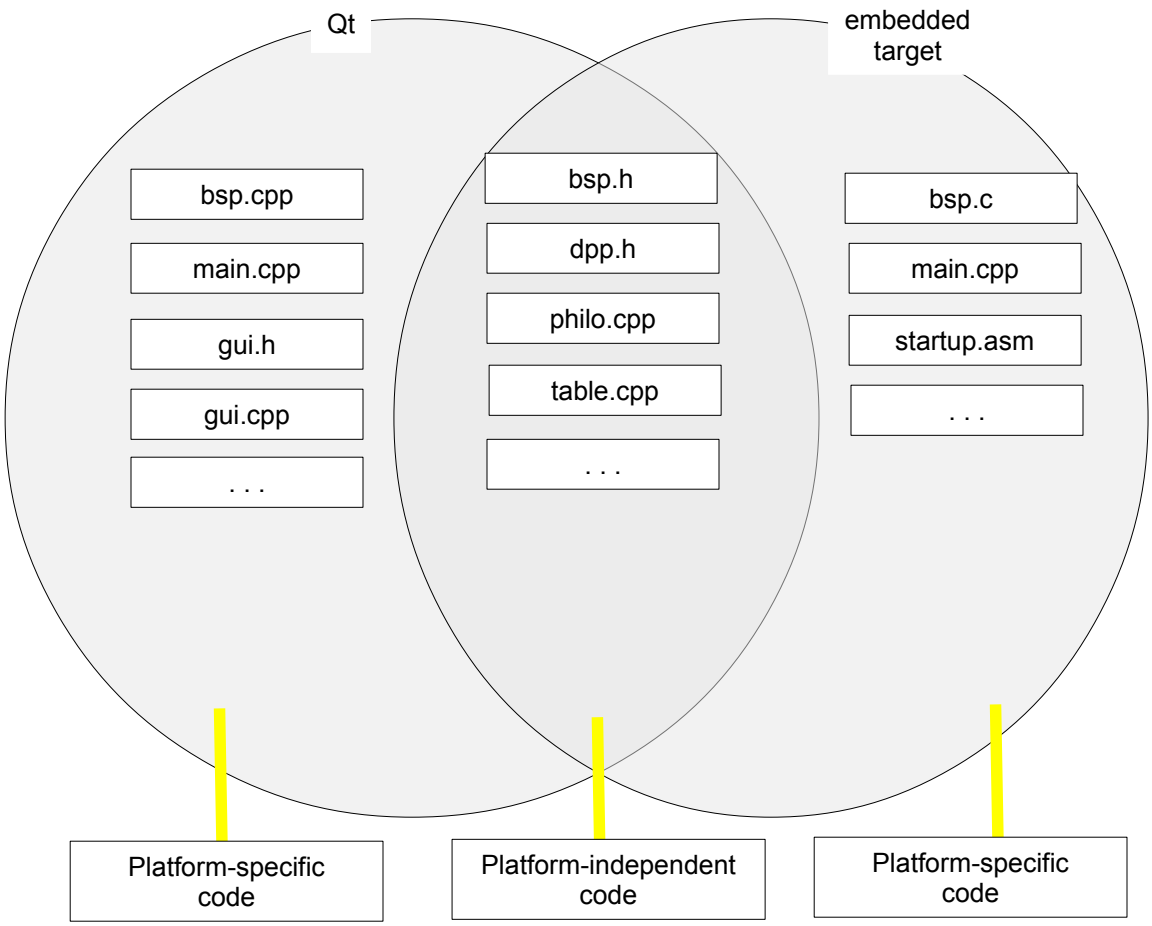
(28) The AO's thread is started to execute the AO event loop.

(29-30) The AO stop operation clears the loop-flag of the active object event loop. This causes the active object thread to terminate.

# 4 Structure of QP Applications with Qt GUI

The QP/C++ examples (DPP-GUI, GAME-GUI, and PELICAN-GUI) accompanying this Application Note demonstrate the recommended overall code structure, which is designed for dual-targeting of embedded applications. The cornerstone of this technique is abstracting all hardware dependencies into an interface, customarily called the Board Support Package (BSP), and implementing this interface differently on different "boards". Figure 11 shows an example of such code partitioning for the DPP application.

**Figure 11: Code structure for dual-targeting of the DPP application to Qt and an embedded target**



## 4.1 Board Support Package (BSP) for Qt-GUI

The BSP incorporates all "board"-related code, which includes functions that depend on the specific environment the application happens to run on. The BSP for Qt projects must be implemented in C++, because it needs to access the C++ Qt API. The BSP is typically located in the application directory (see Listing 1). Listing 7 shows the BSP interface for the DPP-GUI example, while Listing 8 shows the BSP implementation. The explanation section immediately following Listing 8 highlights the main points.

**Listing 7: BSP interface for the DPP example (file `bsp.h`)**

```
#ifndef bsp_h
#define bsp_h

#define BSP_TICKS_PER_SEC   100U

void BSP_displayPaused(uint8_t paused);
void BSP_displayPhilStat(uint8_t n, char_t const *stat);
void BSP_terminate(void);

void BSP_randomSeed(uint32_t seed); // random seed
uint32_t BSP_random(void);  // pseudo-random generator

#endif  // bsp_h
```

**Listing 8: BSP implementation for the DPP example (file `bsp.cpp`)**

```
(1) #include <QtWidgets>
(2) #include "gui.h"                   // GUI header file generated by Qt Designer
    //-----------------
(3) #include "qp_port.h"
(4) #include "dpp.h"
(5) #include "bsp.h"

    ~ ~ ~ ~
    //............................................................................
(6) void QP::QF_onClockTick(void) {
        QP::QF::TICK_X(0U, &l_time_tick);
    }
    //............................................................................
(7) void QP::QF::onStartup(void) {
(8)     QP::QF_setTickRate(BSP_TICKS_PER_SEC);
    }
    //............................................................................
(9) void QP::QF::onCleanup(void) {
    }
    //............................................................................
    void BSP_init(void) {
        ~ ~ ~ ~
    }
    //............................................................................
(10) void BSP_terminate(int) {
        qDebug("terminate");
(11)    QP::QF::stop(); // stop the QF::run() thread
(12)    qApp->quit(); // quit the Qt application *after* the QF_run() has stopped
```

```
    }
    ~ ~ ~ ~
    //............................................................................
(13) void Q_onAssert(char_t const * const file, int line) {
        QMessageBox::critical(0, "PROBLEM",
            QString("<p>Assertion failed in module <b>%1</b>,"
                    "line <b>%2</b></p>")
                .arg(file)
                .arg(line));
        QS_ASSERTION(file, line); // send the assertion info to the QS trace
(14)    qFatal("Assertion failed in module %s, line %d", file, line);
    }
```

(1) Qt5 GUI applications need to include the `<QtWidgets>` header file.

(2) The application needs to include the header file generated by the Qt Designer.

(3-5) The application needs to include the QP-port header file, the BSP interface, and the application interface.

(6) The `QP::QF_onClockTick()` callback is invoked every system clock tick. At the minimum, this callback needs to call `QP::QF::TICK_X()` to process the armed time events. You can also add other time-based actions to this callback.

(7) The `QF::onStartup()` callback is invoked once just before starting to exectue the application.

(8) The system clock rate is set to the desired `BSP_TICKS_PER_SEC` rate.

**NOTE:** The actual ticking rate is determined by the hardware clock interrupt rate and therefore can be really adjusted in integral multiples of the fundamental hardware rate. For example, the hardware clock rate in most 80x86-based systems is set to 100Hz (a tick per 10ms).

(9) The `QF::onStartup()` callback could be used to clean up just before quitting the application.

(10) The `BSP_terminate()` function performs a controlled shut-down of the application.

(11) The call `QF::stop()` stops the ticker thread.

(12) The call `qApp->quit()` stops the Qt event loop.

(13) The assertion failure handler for Qt GUI applications displays a critical message box

(14) and calls the Qt API `qFatal()` to terminate the Qt application. (The use of `qFatal()` offers opportunity to debug the application)

## 4.2 The QS (Quantum Spy) software tracing integration

The QP port to Qt provides particularly easy access to the QS (Quantum Spy) software tracing information. (See [Q_SPY-Ref] and Chapter 11 of [PSiCC2] book for more information about the QS software tracing system). In the Qt port, you can choose to have the QS data converted **on-the-fly** from the compressed binary to the human readable format for direct output to the **Qt debug console** (see Figure 5 and Figure 7). This on-the-fly formatting of the binary QS data is achieved by incorporating code normally used in the QSPY host application into the Qt port.

> **NOTE:** This QS implementation requires access to the QSPY host application code, which resides in the Qtools collection. Therefore, Qtools need to be installed in your system and the `QTOOLS` environment variable must be pointed to the Qtools directory.

For convenience, the QSPY parser code is included in the QP library for Qt, but the QS tracing output is not actually implemented in the QP port. This leaves the ultimate flexibility to the application-level code (typically the Board Support Package) to either perform a direct human-readable QSPY output to the Qt debug console, or perhaps send the binary output to a socket or a file.

### *4.2.1 The QS_onStartup() callback function*

The QS software tracing system is initialized in the `QS_onStartup()` a callback function shown in Listing 9. This particular implementation outputs the QS data in the human-readable format, so it initializes the QSPY parser by calling `QSPY_config()`.

**Listing 9: `QS_onStartup()` implementation in the `bsp.cpp` file of the DPP example**

```
    uint8_t QS_onStartup(void const *arg) {
(1)     static uint8_t qsBuf[4*1024]; // 4K buffer for Quantum Spy
(2)     QS_initBuf(qsBuf, sizeof(qsBuf));

(3)     QSPY_config(QP_VERSION,         // version
                    QS_OBJ_PTR_SIZE,    // objPtrSize
                    QS_FUN_PTR_SIZE,    // funPtrSize
                    QS_TIME_SIZE,       // tstampSize
                    Q_SIGNAL_SIZE,      // sigSize,
                    QF_EVENT_SIZ_SIZE,  // evtSize
                    QF_EQUEUE_CTR_SIZE, // queueCtrSize
                    QF_MPOOL_CTR_SIZE,  // poolCtrSize
                    QF_MPOOL_SIZ_SIZE,  // poolBlkSize
                    QF_TIMEEVT_CTR_SIZE,// tevtCtrSize
                    (void *)0,          // MATLAB File,
                    (void *)0,          // MSC file
(4)                 &custParserFun);    // customized parser function
        . . .
    }
```

(1) The work buffer for the QS buffer is allocated.

(2) The `QS_initBuf()` function passes the buffer to the QS.

(3) The `QSPY_config()` function configures the human-readable QSPY parser.

(4) The last argument to `QSPY_config()` is a callback function to invoke for customized processing of every trace record. This function gives you an opportunity to peek into the QS data and provide a custom (GUI-based) visual representation of any interesting aspects of the trace (see the next sub-section).

### 4.2.2 The Custom Parser callback function

The last argument to `QSPY_config()` is a callback function to invoke for customized processing of every trace record (Listing 9(4)). This function gives you an opportunity to peek into the QS data and provide a custom (GUI-based) visual representation of any interesting aspects of the trace. The following Listing 11 shows an example of how to parse the trace data and how you can visualize the trace data on your GUI.

**Listing 10: `QS_onEvent()` implementation in the `bsp.cpp` file of the DPP example**

```
static int custParserFun(QSpyRecord * const qrec) {
    int ret = 1;  /* perform standard QSPY parsing */
    switch (qrec->rec) {
        case QS_QF_MPOOL_GET: { /* example record to parse */
            int nFree;
            (void)QSpyRecord_getUint32(qrec, QS_TIME_SIZE);
            (void)QSpyRecord_getUint64(qrec, QS_OBJ_PTR_SIZE);
            nFree = (int)QSpyRecord_getUint32(qrec, QF_MPOOL_CTR_SIZE);
            (void)QSpyRecord_getUint32(qrec, QF_MPOOL_CTR_SIZE); /* nMin */
            if (QSpyRecord_OK(qrec)) {
                Gui::instance()->m_epoolLabel->setText(QString::number(nFree));
                ret = 0; /* don't perform standard QSPY parsing */
            }
            break;
        }
    }
    return ret;
}
```

This specific example of a custom parser callback parses only the standard trace record `QS_QF_MPOOL_GET`, but of course you can parse all standard and user-define QS records. Perhaps the best way of knowing how to parse a given trace record is the original `qspy.c` implementation, from which you can simply copy and paste a given trace record and easily decipher what the data elements mean. After the data is de-serialized from the binary stream, you can output it to the GUI, as it is shown with the `Gui::instance()->m_epoolLabel` GUI element.

### 4.2.3 The QS::onEvent() callback function

The QP-Qt port provides a callback function `QS::onEvent()`, which is called after processing of each QP event. This function is called only in the `Q_SPY` configuration and is designed to give you an opportunity to perform the QS output. The following Listing 11 shows how to define to output the QS trace date in the human-readable format.

**Listing 11: `QS::onEvent()` implementation in the `bsp.cpp` file of the DPP example**

```
    void QS::onEvent(void) {
        uint16_t nBytes = 1024;
        QF_CRIT_ENTRY(dummy);
(1)     uint8_t const *block = QS::getBlock(&nBytes);
        QF_CRIT_EXIT(dummy);
        if (block != (uint8_t *)0) {
(2)         QSPY_parse(block, nBytes);
        }
    }
```

(1) The `QS::getBlock()` function obtains a block of data from the QS trace buffer.

(2) The raw trace data is passed to the `QSPY_parse()` function for parsing and output in the human-readable format.

### 4.2.4 *The QSPY_onPrintLn() callback function*

To output the human-readable data with `QSPY_parse()`, you need to also define the callback function `QSPY_onPrintLn()`, which performs the actual low-level output. As shown in Listing 12, in case of a Qt application, you can use `qDebug()` function to print the human-readable output to the Qt debug console.

**Listing 12: `QS_onPrintLn()` implementation in the `bsp.cpp` file of the DPP example**

```
void void QSPY_onPrintLn(void) {
    qDebug(QSPY_line);
}
```

## 4.3 The main function (`main.cpp`)

The `main()` function in the QP-Qt integration is similar to the "bare metal" QP applications, but contains some important additional elements, which have mostly to do with instantiation of the Qt application object. Listing 13 shows the `main()` function for the DPP application. The explanation section immediately following the listing highlights the main points.

**Listing 13: The main() function for the DPP example (file `main.cpp`)**

```
(1) #include "gui.h"
    //-----------------
 (2) #include "qp_port.h"
 (3) #include "dpp.h"
 (4) #include "bsp.h"

    //............................................................................
 (5) static QP::QEvt const *l_philoQueueSto[N_PHILO][N_PHILO];
 (6) static QP::QSubscrList l_subscrSto[DPP::MAX_PUB_SIG];

    // storage for event pools...
 (7) static QF_MPOOL_EL(DPP::TableEvt) l_smlPoolSto[2*N_PHILO]; // small pool

    //............................................................................
 (8) int main(int argc, char *argv[]) {
 (9)     QP::GuiApp app(argc, argv); // GUI application
(10)     Gui gui;                    // GUI object

(11)     gui.show();

         QP::QF::init();  // initialize the framework
         BSP_init();      // initialize the BSP

         // object dictionaries...
         QS_OBJ_DICTIONARY(l_smlPoolSto);

         // initialize publish-subscribe
         QP::QF::psInit(l_subscrSto, Q_DIM(l_subscrSto));
```

```
        // initialize event pools...
        QP::QF::poolInit(l_smlPoolSto,
                         sizeof(l_smlPoolSto), sizeof(l_smlPoolSto[0]));

        // start the active objects...
        for (uint_fast8_t n = 0U; n < N_PHILO; ++n) {
(12)        DPP::AO_Philo[n]->start((n + 1),
                                    l_philoQueueSto[n], Q_DIM(l_philoQueueSto[n]),
                                    (void *)0, 1024U*4U);
        }
(13)    DPP::AO_Table->start((N_PHILO + 1),
                             (QP::QEvt const **)0, 0U, // no event queue
                             (void *)0, 0U);

(14)    return QP::QF::run(); // calls qApp->exec()
    }
```

(1) The `main()` function needs the `gui.h` header file to instantiate and show the GUI.

(2-4) The `main()` function needs the usual QP include files for the QP application.

(5-7) As in all QP applications, you need to provide storage for publish-subscribe and for all event pool that you intend to use in the application.

(8) The `main()` function has the standard signature.

(9) The Qt application object of class (or subclass) of `QPApp` is instantiated on the main stack.

(10) The application-specific Qt main GUI widget is also instantiated on the main stack.

(11) The main GUI widget is displayed.

(12) QP active objects are started.

(13) The QP GUI active object is started.

---

**NOTE:** The GUI active object does **not** need event queue storage.

---

(14) As usual, the `main()` function ends with the call to `QP::QF::run()`, which runs the application and does not return as long as the application is running. In Qt, `QP::QF::run` calls `qApp->exec()` and returns the exit code from this call.

## 4.4 The Qt GUI implementation

The Qt-based GUI is completely decoupled from the application (through the BSP interface) and can be defined any way supported by Qt (such as Qt Designer or Qt Quick). In the examples accompanying this Application Note, the GUI was created graphically with the Qt Designer tool. In all the provided examples, the Qt Designer has been used in a standard way described in the Qt documentation (e.g., see [Qt-GUI]). The following sections explain briefly the main points of GUI implementation.

**NOTE:** The Qt GUI API is generally **not** thread-safe, so it should not be accessed from the non-GUI Active Objects. All the access to GUI elements must be localized in the dedicated GUI Active Object (GUI manager).

### *4.4.1 Rapid GUI design with Qt Designer*

The provided examples come with the `gui.ui` "form file" (see Listing 1), which contains the GUI layout specification for the Qt Designer tool. From the Qt Creator IDE, you can conveniently launch the Qt Designer as a plugin by double-clicking on the `gui.ui` file in the Qt Designer's Forms folder. The designing process is very intuitive (). You drag the various elements from the palette to the canvas and you edit the properties of the elements with the property editor. You can also control the stacking order.

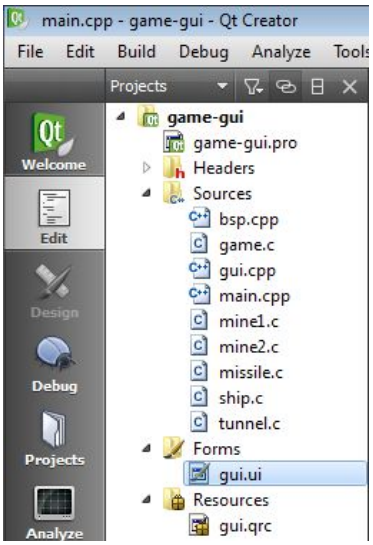
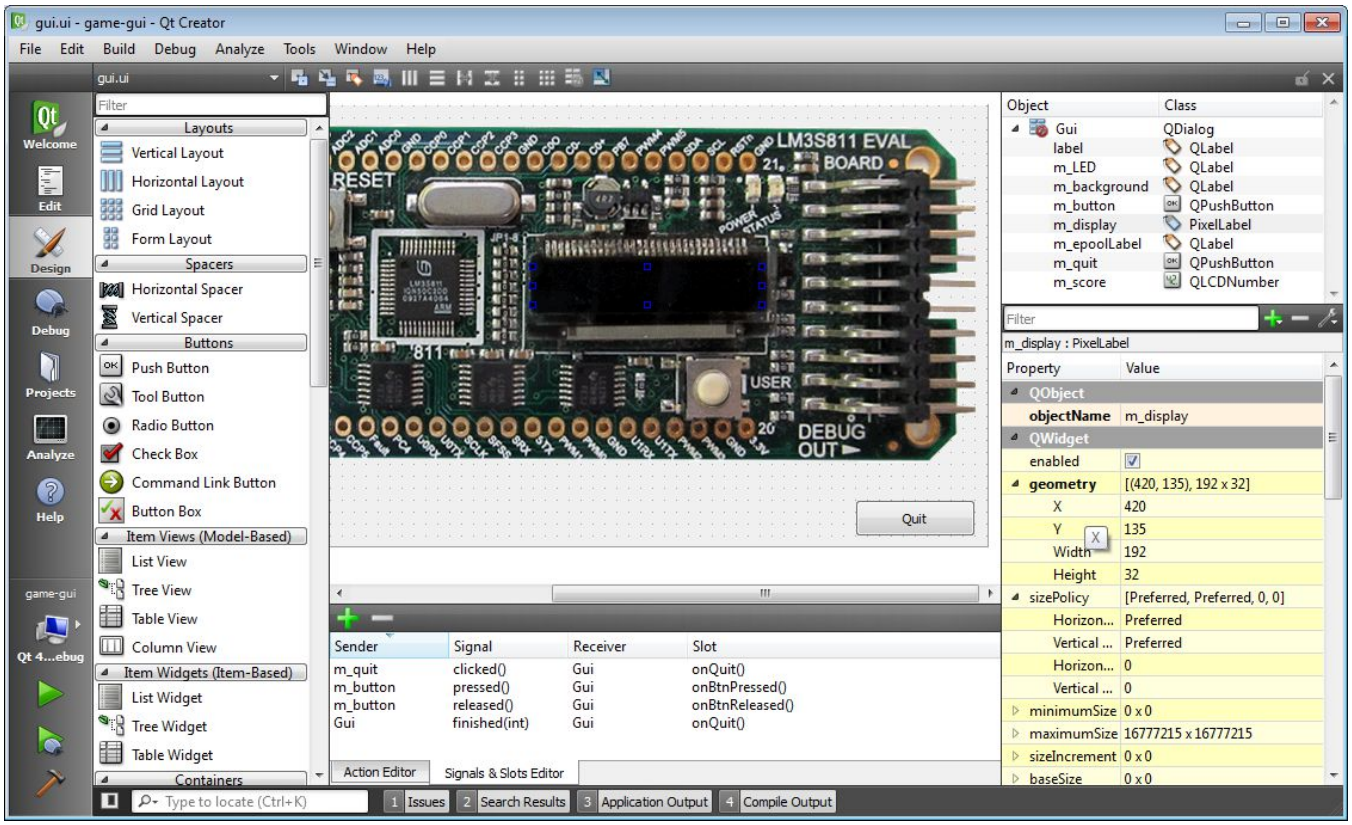



**Figure 12: Qt Designer plugin open in the Qt Creator IDE**

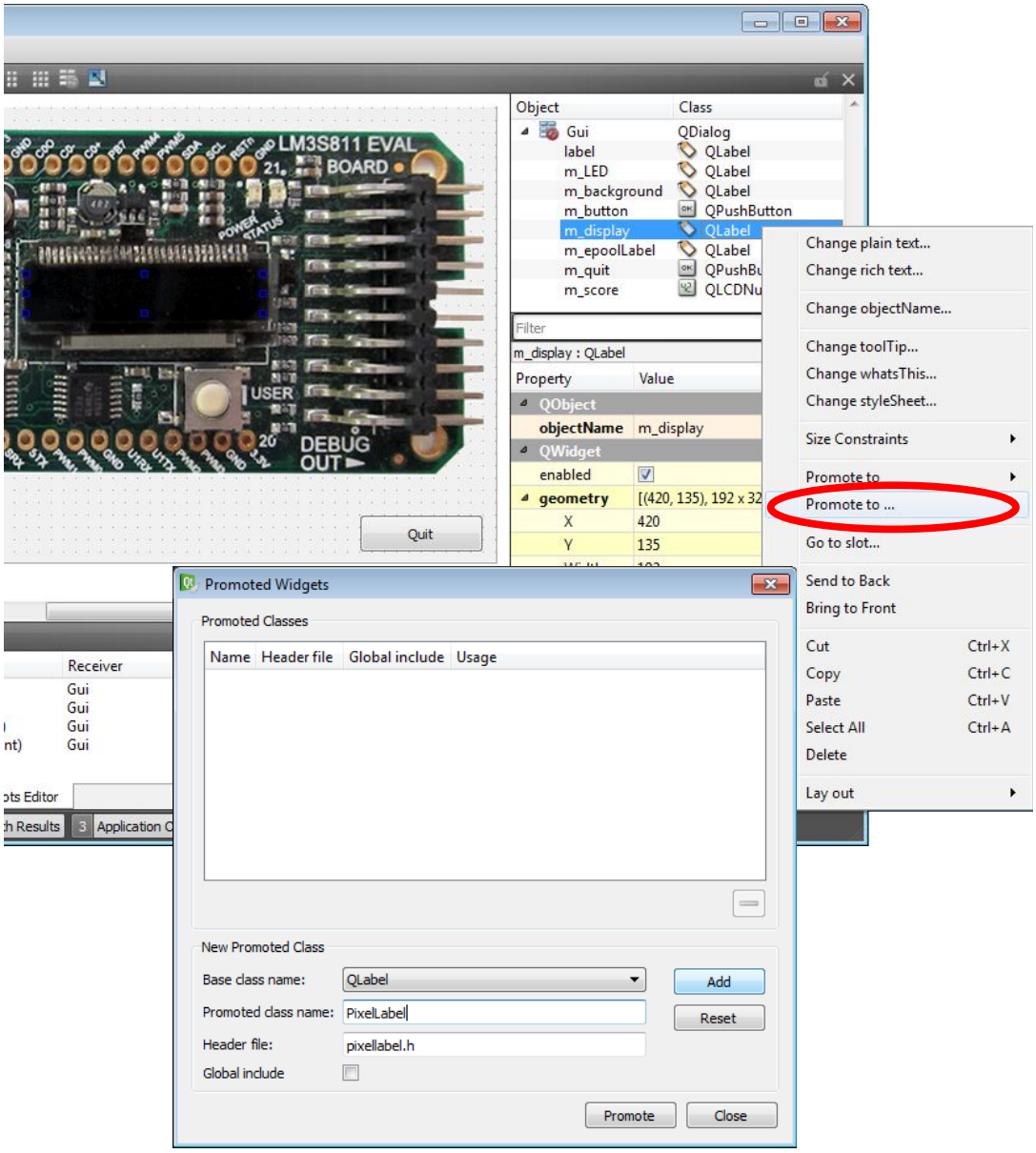


### 4.4.2 *Using custom widgets (e.g., PixelLabel) in the Qt Designer*

While adding standard widgets to the design is very easy in Qt Designer, adding a custom widget requires an extra step.

For example, the simulation of the "Fly 'n' Shoot" game on the EK-LM3S811 needs a small graphic OLED display. Since a graphic display is a quite often used component in embedded user interfaces, the QP-Qt integration provides a custom widget called **`PixelLabel`**, which can efficiently render graphic display of up to 24-bit color (Red, Green, Blue) and provides an efficient pixel-level interface. The `PixelLabel` custom widget has been specifically designed for use with the Qt Designer. `PixelLabel` inherits the standard `QLabel` and relies on the ability of the Qt Designer to promote a standard component to the desired subclass. Figure 13 illustrates the details of such promotion.

**Figure 13: Promoting a custom widget in the Qt Designer**



To add the custom `PixelLabel` widget to the design, you need to:

1. Drag the `QLabel` standard widget to the design (because `PixelLabel` inherits `QLabel`) and position it at the desired place

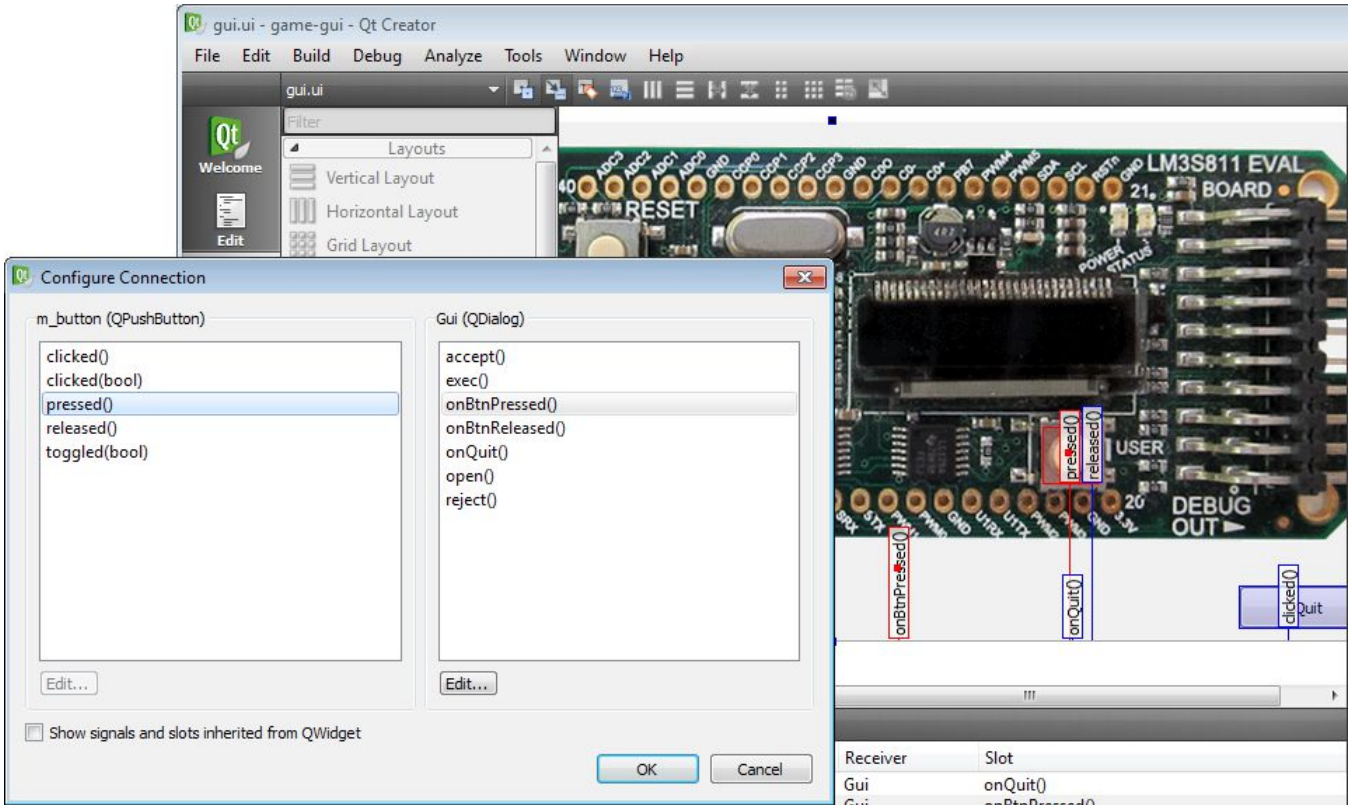
> **NOTE:** The Qt Designer lets you drag the component only to a grid position (typically with 10 pixel granularity). To position a widget precisely, you need to press **Ctrl key** and nudge the widget one pixel at a time with the **arrow keys**.

2. Edit the name of the widget in the Property Editor (e.g., `m_display`)
3. Right-click on the widget in the widget summary panel (see Figure 13), and choose the "Promote To..." menu option.
4. This opens the "Promoted Widgets" dialog box, in which you specify `PixelLabel` as the New Promoted Class (see Figure 13), and accept the suggested name of the header file (pixellabel.h).
5. Click "Promote" and then "Close".

### *4.4.3 Editing the Signal-Slot connections*

To edit the Signal-Slot connections in the Qt Designer you select the **Edit | Edit Signals/Slots** menu item (or clicking on the Edit Signals/Slots toolbar button), which switches to the Signals/Slots editing mode. For example, Figure 14 shows configuring the USER push-button signals.

**Figure 14: Editing Signals/Slots in the Qt Designer**



### 4.4.4 Integrating the generated UI with the hand-written code

The Qt Designer produces the "ui_gui.h" header file, which needs to be added to the hand-written code. As shown in Listing 14, you need to include this header file in the "gui.h" header file, and you need to add the generated Ui_Gui class to the list of base classes for the Gui class. Also, the examples come with just one *.ui form file, but you can use as many form files as you need in your applications (each form file specifies one main window or a dialog box.)

**Listing 14: gui.h header file for the "Fly 'n' Shoot" game example**

```
    #include <QDialog>
(1) #include "ui_gui.h" // generated by QtDesigner

(2) class Gui : public QDialog, public Ui_Gui {
(3)     Q_OBJECT

    public:
        explicit Gui(QWidget *parent = 0);
        static Gui *instance;

(4) private slots:
        void onBtnPressed();
        void onBtnReleased();
        void onQuit();

    protected:
(5)     virtual void wheelEvent(QWheelEvent *e);
        //virtual void keyPressEvent(QKeyEvent *e);
    };
```

(1) The generated ui_gui.h header file is included in the hand-written gui.h header file

(2) The custom Gui class is derived from QDialog and the generated Ui_Gui class.

> **NOTE:** All Gui classes in the examples are dialog-based (derive from the QDialog base class), but this is not mandatory. You can choose to derive your GUIs from QMainWindow or other Qt base classes.

(3) The Q_OBJECT macro adds the meta-object facilities to the Gui class and is required in all subclasses of QObject.

(4) The custom Gui class needs to declare all slots that you've used in the custom Signals/Slots connections to the UI.

(5) Overriding the standard virtual event handler methods can provide customized input to the Gui.

The implementation file "gui.cpp" is simple typically very simple, as most GUI setup is performed in the generated UI superclass. Listing 15 shows the complete code while the following explanation section highlights the main points.

**Listing 15: `gui.cpp` implementation file for the "Fly 'n' Shoot" game example**

```
    #include <QtWidgets>
(1) #include "gui.h"
    //-----------------
    #include "qp_port.h"
    #include "dpp.h"
    #include "bsp.h"

    Q_DEFINE_THIS_FILE

    //.....................................................................
(2) static Gui *l_instance;

    //.....................................................................
(3) Gui::Gui(QWidget *parent)
      : QDialog(parent)
{
        instance = this;  // only one instance for the Gui

(4)     setupUi(this);

        static quint8 const c_offColor[] = {  15U,  15U,  15U };
(5)     m_display->init(BSP_SCREEN_WIDTH, 2U,
                        BSP_SCREEN_HEIGHT, 2U,
                        c_offColor);
    }
    //.....................................................................
(6) void Gui::onBtnPressed() {  // slot
        m_button->setIcon(QPixmap(":/res/EK-BTN_DWN.png"));
        static QP::QEvt const fireEvt(GAME::PLAYER_TRIGGER_SIG);
        QP::QF::PUBLISH(&fireEvt, (void*)0);
    }
    //.....................................................................
(7) void Gui::onBtnReleased() { // slot
        m_button->setIcon(QPixmap(":/res/EK-BTN_UP.png"));
    }
    //.....................................................................
(8) void Gui::onQuit() {        // slot
        BSP_terminate(0);
    }
    //.....................................................................
(9) void Gui::wheelEvent(QWheelEvent *e) {
        if (e->delta() >= 0) {
            BSP_moveShipUp();
        }
        else {
            BSP_moveShipDown();
        }
    }
```

(1) Include the generated `gui.h` header file, which in turn includes all GUI include files used in the UI.

(2) The Singleton GUI instance is stored in the local static variable.

(3) The GUI class constructor performs the setup.

(4) The most important part of the setup is calling `setupUi()`, which initializes all UI components and sets their properties per the specification in Qt Designer. This step also initializes all signal-slot connections established in Qt Designer.

(5-7) Custom slots that are connected to various GUI widgets.

(8) The quit slot of the application calls the platform-dependent `BSP_terminate()` implementation.

(9) Custom event handler that adds mouse-wheel input to the application.

### *4.4.5 Using the PixelLabel class*

As mentioned above, the QP-Qt utility class `PixelLabel` has been specifically designed to add efficient, pixel-addressable Qt widget for rendering graphical displays (LCD, OLED, etc.), which are very common in embedded devices. Listing 15(5) shows the initialization of the `PixelLabel`, in which you can specify the screen horizontal and vertical resolutions as well as "**scaling factors**". These scaling factors alloy you to magnify the original display by mapping one original pixels to a cluster of pixels in the `PixelLabel`. For example, the OLED display of the EK-LM3S811 board is scaled by factor 2 from the original, meaning that every physical pixel of the original OLED display is mapped to 2x2 pixels of the simulated `PixelLabel` display. Listing 16 Shows the mapping of the OLED display pixels to the `PixelLabel` pixels. The `PixelLabel` pixels are numbered from left to right horizontallyand from top to bottom vertically, starting at (0,0) in the top-left corner. The pixel-level interface is easy to adapt for any other pixel mapping.

**Listing 16: Mapping pixels of the OLED display of the EK-LM3S811 board to the PixelLabel**

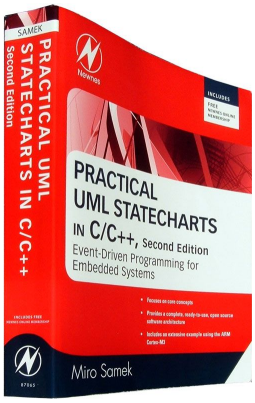
```
    void BSP_drawBitmap(uint8_t const *bitmap) {
        PixelLabel *display = Gui::instance->m_display;
        for (unsigned y = 0U; y < BSP_SCREEN_HEIGHT; ++y) {
            for (unsigned x = 0U; x < BSP_SCREEN_WIDTH; ++x) {
                uint8_t bits = bitmap[x + (y/8)*BSP_SCREEN_WIDTH];
                if ((bits & (1U << (y & 0x07U))) != 0U) {
(1)                 display->setPixel(x, y, c_onColor);
                }
                else {
(2)                 display->clearPixel(x, y);
                }
            }
        }

(3)     display->redraw();
    }
```

(1-2) The `PixelLabel` pixel-level interface is used set or clear pixels in the pixel array.

(3) After setting up the pixels, the `PixelLabel::redraw()` method is called to render the pixels to the screen.

# 5 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<br>Available from most online book retailers, such as Amazon.com.<br>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 |
| [Qt-GUI] "C++ GUI Programming with Qt 4, Second Edition", Jasmin Blanchette and Mark Summerfield. | Prentice Hall Open Source Development Series, 2008, ISBN-13: 978-0-13-235416-5 |
| [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 |

# 6 Contact Information

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

Phone: +1 919-360-5668
Fax: +1 919 869-2998

e-mail: info@quantum-leaps.com
WEB : http://www.quantum-leaps.com
http://www.state-machine.com

**Qt project homepage:**
**http://qt-project.org/**