quantum™ LeaPs

innovating embedded systems

QP-nano™
Programmer's Manual

Document Version 1.5.01

October 2006

quantum LeaPs™, LLC

www.quantum-leaps.com

Table of Contents

1 Overview

Quantum Platform Nano™ (QP-nano) is a reusable event-driven application infrastructure for executing concurrent UML state machines (statecharts) specifically designed for low-end 8- and 16-bit embedded microcontrollers. The use of QP-nano generally simplifies the design of event-driven software by allowing the application to be divided into multiple **active objects**[1] that QP-nano manages. Active objects in QP-nanao are encapsulated tasks (each embedding a state machine and an event queue) that communicate with one another asynchronously by posting events to each other's event queues. Within an active object, events are processed sequentially in a run-to-completion (RTC) fashion, while QP-nano encapsulates all the details of *thread-safe* event exchange and queuing.

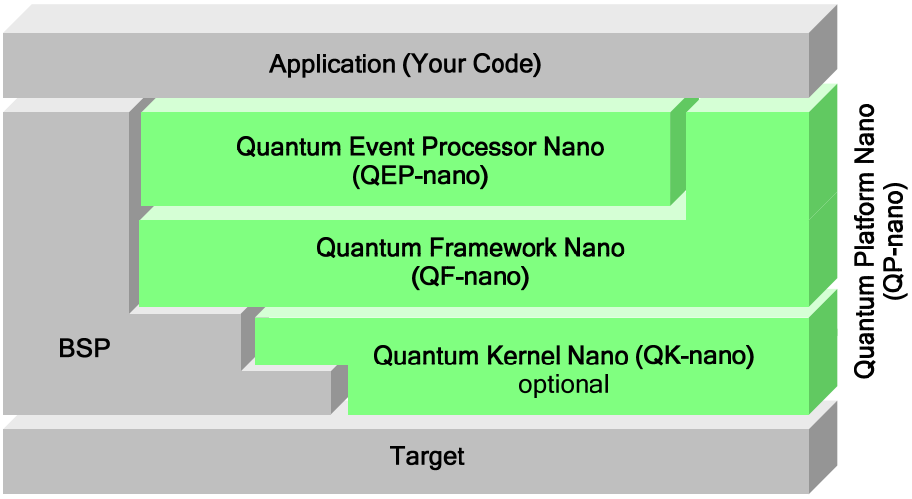

Figure 1 Components of the Quantum Platform Nano (QP-nano).

Most of QP-nano is written in portable ANSI-C, with microprocessor-specific and compiler-specific code kept to a minimum for ease of portability. QP-nano is designed to work with just a "bare metal" target microcontroller, but can also be combined with the very lightweight Quantum Kernel Nano (QK-nano) to provide preemptive multitasking capabilities (requires more stack space). QP-nano is very compact, typically taking about 0.8-1.2KB of code space (depending on the CPU code density and compiler options) and only several bytes of RAM (not including the stack). A minimal system that can benefit from QP-nano can have as little as 128 bytes of RAM and 2-4KB of ROM. If you can program your application in C, you most likely can afford QP-nano.

[1] The term "*active object*" comes from the Unified Modeling Language (UML) and denotes "*an object having its own thread of control*" [OMG 03].

1.1 Key Features of QP-nano

QP-nano does not attempt to implement fully the UML state machine specification [OMG 03]. The design philosophy of QP-nano is to provide a minimal and generic event processor (QEP-nano), and a minimal real-time framework (QF-nano) to allow run-to-completion execution of concurrent state machines in your application. The main features of QP-nano are:

- Full support for hierarchical state nesting.
- Guaranteed entry/exit action execution on arbitrary state transition topology with up to 4 levels of state nesting.
- Full support of nested initial transitions.
- Highly maintainable boilerplate state machine representation in C code that is very easy to modify and extend at any stage of the development cycle.
- Extremely small RAM/ROM footprint. A state machine instance in QEP-nano requires only one function pointer and an event in RAM (altogether typically 4 bytes). The hierarchical state machine code takes about 700-900 bytes of code space while the simpler "flat" finite state machine takes only about 60 bytes of code space (ROM). QF-nano requires typically around 500 bytes of ROM and just a few bytes of RAM. Of course, you must also budget some RAM for the C stack.
- No RAM required for representing states and transitions—number of states limited only by the code space (ROM).
- Provisions in the code for placing constants in code space (ROM).
- Support for events with one scalar parameter, configurable to 0 (no parameter), 8, 16, or 32-bits.
- Build-in deterministic, thread-safe event queue for direct event delivery mechanism with first-in-first-out (FIFO) policy.
- One single-shot time event (timer) per active object with configurable tick-counter dynamic range: 0 (no timer), 8, 16, or 32-bits
- Support for up to 8 concurrently executing state machines (active objects)
- Built-in cooperative, priority-based scheduler for Run-To-Completion (RTC) execution of active objects.
- Optionally configurable for fully preemptive multitasking kernel (QK-nano). The preemptive option requires only 50-100 bytes more of code space (ROM), but requires more stack (RAM) than the non-preemptive option.
- Idle-time callback for implementing power-saving modes.
- Very clean source code passing strict static analysis with PC-Lint™.
- Source code 98%-compliant with the Motor Industry Software Reliability Association (MISRA) Guidelines for the Use of The C Language in Vehicle Based Software [MISRA 98].
- Sample applications and exhaustive test suite included in the code distributions and explained in the manual.
- Comprehensive documentation. Application Notes and extensive examples available online from http://www.quantum-leaps.com/doc.
- Dual-licensing with open source and commercial licensing options.

1.2 QP-nano versus The Full-Version QP

QP-nano implements a proper subset of features provided in the full-version QP (see "QEP/C Programmer's Manual", "QF/C Programmer's Manual", and "QK/C Programmer's Manual"), so if you are familiar with the QP full version, you should be immediately productive with QP-nano. Moreover, large parts of the documentation from the QP full version pertain also to QP-nano.

QP-nano has been specifically designed for small 8- and 16-bit embedded microcontrollers with less than 1KB of RAM that simply cannot accommodate the full-version of QP. The main objective in designing QP-nano was to reduce the RAM footprint. The second objective was to reduce the number of parameters and stack use in function calls. In the process, several features of the full version QP had to be eliminated and replaced with simpler mechanisms. Perhaps the most important change is elimination of arbitrary event parameters and dynamic event allocation from event pools, because it simply is to RAM-intensive. Also, the time event management has been drastically simplified.

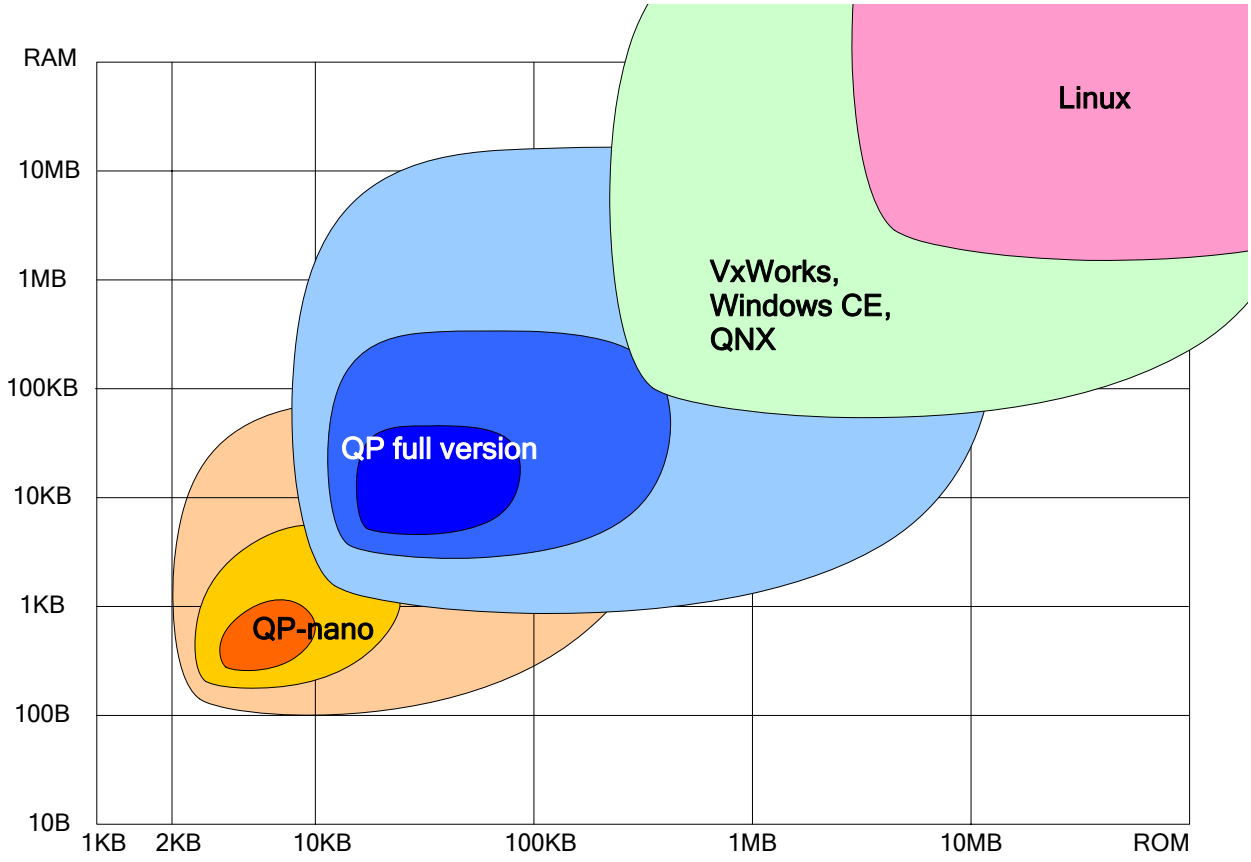

Figure 2 Applicability of QP-nano vs. QP full version depending on ROM and RAM size. QP-nano and QP full version correspond to the highest volume embedded products.

As shown in Figure 2, QP-nano is applicable starting from about 100 bytes of RAM and 2KB of ROM, with the best fit for systems of 256-1KB of RAM and 4+KB or ROM. The minimum requirements for the QP full version are about 1KB of RAM and 8KB or ROM, with the best fit for systems with 4+KB of RAM and 10+KB of ROM.

1.3 A Framework, Not a Toolkit

The design of QP-nano is based on the real-time framework architecture (QF-nano). Reuse on this level leads to inversion of control between your application and the infrastructure on which it is based. When you use a function library (toolkit), such as a traditional real-time kernel, you write the main body of the application and call the code you want to reuse. When you use a framework, such as QF-nano, you reuse the main body and write the code *it* calls.

1.4 Run-to-Completion Event Processing

QP-nano executes concurrent state machines in the run-to-completion (RTC) fashion, which is the universally assumed semantic in state machine formalisms, including UML statecharts. All events destined for a single state machine (active object) are processed sequentially, and the response of an active object to an event is treated as an atomic, (logically) uninterruptible activity. This means that you can code your applications with purely *sequential* techniques, which are much **safer** because you don't need to fiddle with semaphores, monitors, condition variables, signals, or other such potentially dangerous mechanisms that are indispensable in every conventional real-time kernel-based design.

Run-to-completion does not mean, however, that a state machine has to monopolize the processor until the RTC step is complete. The preemption restriction only applies to the task context of the state machine that is already busy processing events. Interrupts and other state machines (not related to the task context of the busy state machine) may be running — possibly preempting the currently executing state machine. As long as other state machines do not share resources with each other, there are no concurrency hazards.

To this end, QP-nano can be configured at compile time to either support a cooperative, non-preemptive scheduling, or fully preemptive multitasking real-time kernel (QK-nano) with very deterministic, microsecond-level response times for high-priority state machines (active objects). The preemptive option, however, requires significantly more stack space and slightly more code than the non-preemptive scheduler.

1.5 Licensing QP-nano

The **Generally Available (GA)** distribution of QP-nano™ (which includes QEP-nano and QF-nano) is available for download from the www.quantum-leaps.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.

The QK-nano™ component is available only under the commercial, royalty-free licensing. The evaluation version of QK-nano™ is available under the terms of the Quantum Leaps Evaluation License that you must accept to download the code and documentation. The evaluation version does

not include the QK-nano source (only pre-compiled object files are provided). For more information, please refer to the licensing section of our website at: http://www.quantum-leaps.com/licensing/com.htm.

1.6 Related Documentation and Resources

This *Programmer's Manual* describes QP-nano and state machine implementation techniques based on QP-nano, but does not cover other related subjects, such as: state machine basics, UML notation, state machine design, real-time concepts, and others. Please refer to the documents listed in the Section 11 "References" of this for information about the related subjects.

Also, as mentioned before, large parts of Programmer's Manuals of QP full version pertain also to QP-nano™ (see "QP Programmer's Manual").

2 Getting Started with QP-nano

This section describes how to install QP-nano and what other software or hardware components you might need to get started with QP-nano.

2.1 Installation

The evaluation version of QP-nano is available for download (after online registration) from http://-www.quantum-leaps.com/products/qpn.htm.

The QP-nano code is distributed in a ZIP archive (qpn_<ver>.zip, where <ver> stands for a specific QP-nano version, such as 1.2.00). You can uncompress the archive into any directory. The installation directory you choose will be referred henceforth as QP-nano Root Directory <qpn>. The following Listing 1 shows the directory structure and selected files included in the standard QP-nano distribution. (Please note that the QP directory structure is described in detail in a separate Application Note: "QP Directory Structure")

```
<qpn>/                    - QP-nano Root Directory
  |
  +-doxygen\              - subdirectory containing the QP-nano reference manual
  | +-html\
  | | +-index.html        - QP-nano reference manual in HTML
  |
  +-examples\             - subdirectory containing the QP-nano example files
  | +-msp430\             - examples for the MSP430 (the eZ430-F2013 evaluation kit)
  | | +-iar\              - examples compiled with the IAR MSP430 compiler
  | | | +-pelican-eZ430\  - the PELICAN crossing example for the eZ430
  | | | | +-dbg\          - directory containing the debug build
  | | | | +-bsp.c         - Board Support Package for the Toolstick/IAR 8051
  | | | | +-bsp.h         - BSP header file
  | | | | +-main.c        - the main function
  | | | | +-ped.c         - the Pedestrian state machine
  | | | | +-pelican.c     - the PELICAN crossing state machine
  | | | | +-pelican.h     - the PELICAN application header file
  | | | | +-pelican.eww   - IAR Embedded Workbench workspace for the PELICAN crossing
  | | | | +-qpn_port.h    - QP-nano configuration for this application
  | | | | +-qepn.r43      - pre-compiled object code for the QEP-nano component
  | | | | +-qfn.r43       - pre-compiled object code for the QF-nano component
  | | | |
  | | | +-bomb-eZ430\     - the Time Bomb example (FSM only) for the eZ430
  | | |   +-dbg\          - directory containing the debug build
  | | |   +-bsp.c         - Board Support Package for the Toolstick/IAR 8051
  | | |   +-bsp.h         - BSP header file
  | | |   +-main.c        - the main function
  | | |   +-bomb.c        - the Time Bomb state machine (FSM)
  | | |   +-bomb.h        - the Time Bomb application header file
  | | |   +-bomb.eww      - IAR Embedded Workbench workspace to for the Time Bomb
  | | |   +-oper.c        - the Operator state machine (FSM)
  | | |   +-qpn_port.h    - QP-nano configuration for this application
  | | |   +-qepn.r43      - pre-compiled object code for the QEP-nano component
  | | |   +-qfn.r43       - pre-compiled object code for the QF-nano component
  | |
  | +-x86\                - examples for the x86 (DOS)
  |   +-tcpp101\          - examples compiled with the Turbo C++ 1.01 compiler
  |     +-pelican\        - the PELICAN crossing example (non-preemptive version)
  |     | +-dbg\          - directory containing the debug build
  |     | | +-PELICAN.EXE - executable that you can run on any Windows PC
  |     | +-bsp.c         - Board Support Package for DOS/Turbo C++ 1.01
```

```
|      | +-bsp.h         - BSP header file
|      | +-main.c        - the main function
|      | +-ped.c         - the Pedestrian state machine
|      | +-pelican.c     - the PELICAN crossing state machine
|      | +-pelican.h     - the PELICAN application header file
|      | +-PELICAN.PRJ   - Turbo C++ 1.01 project file for the PELICAN application
|      | +-STDINT.H      - exact-width integer types (C-99 standard library file)
|      | +-qpn_port.h    - QP-nano configuration for this application
|      | +-QEPN.OBJ      - pre-compiled object code for the QEP-nano component
|      | +-QFN.OBJ       - pre-compiled object code for the QF-nano component
|      |
|
+-include\              - subdirectory containing the QP-nano public interface
| +-qassert.h           - embedded-systems-friendly assertions used in QP-nano
| +-qepn.h              - The platform-independent QEP-nano header file
| +-qfn.h               - The platform-independent QF-nano header file
| +-qkn.h               - The platform-independent QK-nano header file
|                         (not available in open source version)
|
+-source/               - QP-nano source files
| +-qepn.c              - QEP-nano implementation
| +-qfn.c               - QF-nano implementation
| +-qkn.c               - QK-nano implementation (not available in open source version)
| |
| +-lint/               - Directory for linting QP-nano
| | +-qpn_port.h        - QP-nano "port" to lint
| | +-lin.bat           - A batch file to invoke PC-lint™
| | +-std.lnt           - An indirect PC-lint file with compiler-dependent options
| | +-options.lnt       - An indirect PC-lint file with compiler-independent options
```

Listing 1 Directories and files in the standard QP-nano distribution

2.2 What You Need to Use QP-nano

The QP-nano source code is written in highly portable ANSI C. To use QP-nano, you need a C compiler or a cross-compiler for your platform. Your C compiler must also provide a mechanism to disable and enable interrupts from C. Some compilers allow you to insert in-line assembly language statements into your C source code, which makes it easy to insert the proper processor instructions to enable and disable interrupts. Other compilers actually contain language extensions to enable and disable interrupts. Alternatively, interrupt enabling and disabling can be done inside C-callable assembler functions.

If your C compiler supports interrupts in C, you can use this feature with QP-nano (both with non-preemptive and preemptive option). In this case you might not need to code in assembly at all.

The QP-nano component is deployed as source files that you link with your application[2]. The basic configuration of QP-nano (e.g., choosing the size of the event parameter) is accomplished at compile-time via preprocessor macros.

2.2.1 The Texas Instruments eZ430-F2013

To demonstrate QP-nano on a real-life small microcontroller, this Manual uses the very inexpensive evaluation kit eZ430-F2013 for the MSP430F2013 microcontroller from Texas Instruments (see Figure 2). The eZ430 is available directly from Texas Instruments website http://www.ti.com/-eZ430 and costs $20.

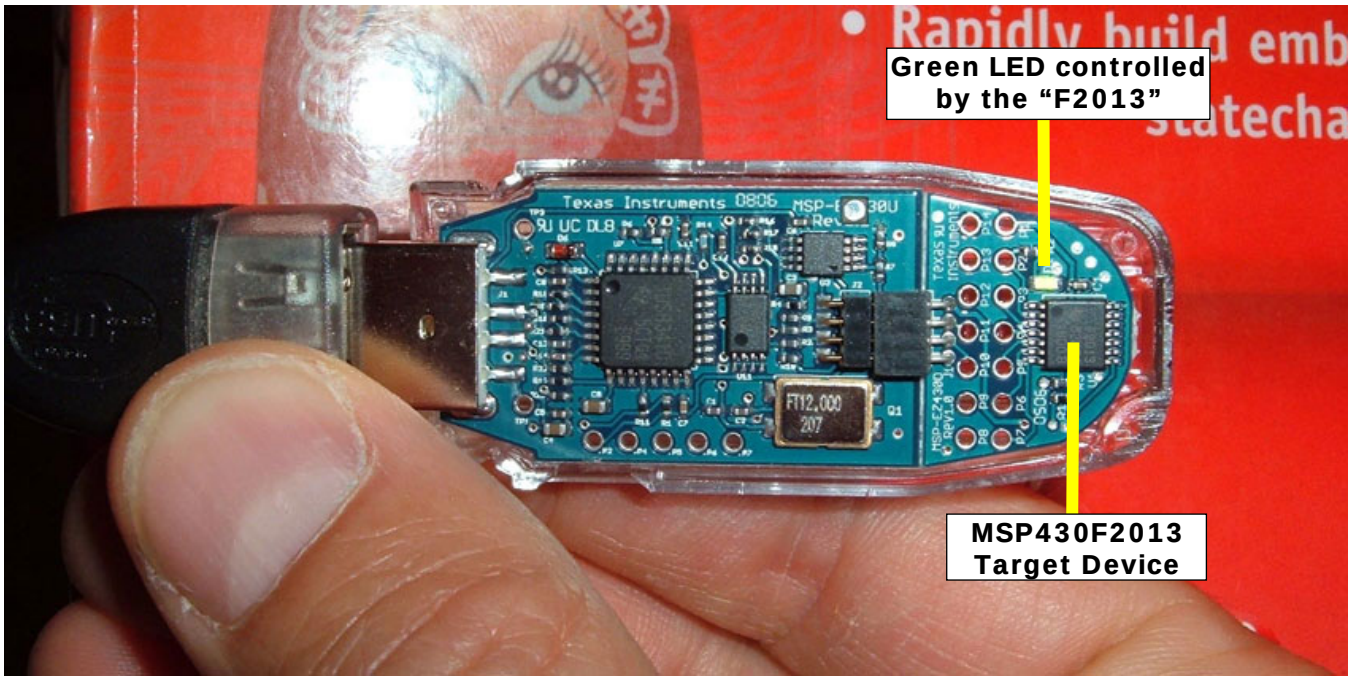

Figure 3 Texas Instruments eZ430-F2013 USB stick combines the USB debugger (left part) and the detachable target board with the MSP430F2013 device (right).

[2] The evaluation version of QP-nano contains only pre-compiled object modules that you statically link with your application.

The eZ430 USB stick contains the integrated USB J-tag pod and the target microcontroller MSP430F2013 with 128 bytes of RAM and 2KB of Flash ROM in a 14-pin package. The evaluation kit includes a CD-ROM with the KickStart™ version of the IAR MSP430 C compiler (version 3.40A as of this writing).

For compatibility with the Makefiles provided in this Manual, you should install the IAR Embedded Workbench for MSP430 into the directory **C:\tools\IAR\430_KS_3.40A**.

2.2.2 Borland Turbo C++ 1.01

To allow you evaluating the software without any embedded target board, the standard QP-nano distribution includes examples compiled with the legacy Turbo C++ 1.01 compiler, which is available for a *free* download from the Borland "Museum" at http://bdn.borland.com/article/-0,1410,21751,00.html.

NOTE: The legacy DOS platform has been chosen because it still allows programming with interrupts, directly manipulating CPU registers, and directly access I/O space. No other modern 32-bit development environment for the commodity PC allows this much so easily. The ubiquitous PC running under DOS (or a DOS console within any variant of Windows) is as close as it gets to emulate an embedded platform on a desktop machine.

To install Borland Turbo C++ 1.01, unzip the `TCPP101.ZIP` archive into a temporary directory. Run the `INSTALL.EXE` program and follow the installation instructions.

For compatibility with the provided examples, you should install the tool into the directory **C:\tools\tcpp101**. If you choose a different directory, you'll need to modify the make files and the linker response files provided in the standard QEP/C distribution.

2.2.3 GNU Make

For your convenience, the commercial QP-nano distribution contains make files for building the examples from the command line. To take advantage of these make files, you need a GNU-compatible make utility. For the Windows host, the MinGW GNU-make is available for a free download from http://www.MinGW.org/download.shtml and mirrored at http://www.quantum-leaps.com/downloads/mingw32-make-3.80.0-3.exe.

2.2.4 The <stdint.h> Header File

QP-nano uses the C99 standard file <stdint.h> for portable exact-width integer types (see http://www.open-std.org/jtc1/sc22/open/n2794/ Section 7.18). The newer C compilers should support this header file as part of the standard C library. In case this standard header file is not available (e.g., in a pre-standard compiler), you should create it and place in the compiler's include directory. At a minimum, this file should contain the typedefs for the following exact-width integer data types (C99 Standard Section 7.18.1.1) that correspond to the CPU and compiler settings you're using to build QP-nano:

exact size	signed	unsigned
8 bits	int8_t	uint8_t
16 bits	int16_t	uint16_t
32 bits	int32_t	uint32_t

For QP-nano ports to pre-C99 compilers, a minimal <stdint.h> header file will be provided as part of the QP-nano port. However, because the port cannot anticipate the installation directory of the compiler, you need to manually move the provided stdint.h header file from the QP-nano port directory to the compiler's header file directory.

NOTE: The Borland Turbo C++ 1.01 are pre-C99-standard compilers. The QP-nano distribution includes minimal stdint.h header files for this compiler in the examples directory, which you should move into the INCLUDE directory of your compiler after the installation.

3 Building and Executing the QP-nano Examples

This section explains how to build and execute the QP-nano applications based on several examples. The goal of this section is to get you started with QP-nano quickly without slowing you down with full-blown detail. This information is intentionally included early in this document, so that you could start experimenting with QP-nano as soon as possible. The main focus of this section is to walk you quickly through the main points without slowing you down with full-blown detail. The in-depth state machine implementation techniques and guidelines are described later in this Manual.

The PELICAN (PEdestrian LIght CONtrolled) crossing example showcases the QP-nano support for Hierarchical State Machines. The example has been selected specifically to require minimal target capabilities. In particular, you can verify the program even with just one LED controlled by software (or a pin connected to a scope). Also, the example assumes no input capabilities, such as pushbuttons, which is actually the case of the eZ430 USB stick.

At the end of this section, you will also find an example of a "Time Bomb" implemented as simple non-hierarchical Finite State Machine. This example shows how to use QP-nano in even smaller systems.

3.1 The PELICAN Crossing Example on the eZ430 USB Stick

The PELICAN crossing example for for the eZ430 comes in two variants: non-preemptive and preemptive. The non-preemptive PELICAN crossing example is located in the directory `<qpn>\examples\msp430\iar\pelican-eZ430\`. The preemptive variant with the QK-nano real-time kernel is located in the directory `<qpn>\examples\msp430\iar\pelican-qk-eZ430\`. Both variants use exactly the same state machine code and differ only in the QP-nano configuration and interrupt handling in the BSP. The procedures for building and executing the two PELICAN crossing examples are identical.

Here are the steps to build, load and execute the PELICAN example:

1. Install the IAR Embedded Workbench for MSP430 that comes on the eZ430 CD-ROM, Install the USB drivers for the ez430 USB stick, as described in the eZ430 User's Guide.
2. Open the IAR Embedded Workbench for MSP430
3. Open the IAR PELICAN workspace by selecting menu File->Open->Workspace… The IAR PELICAN crossing workspace file is located in `<qpn>\examples\msp430\iar\pelican-eZ430\pelican.eww`
4. Build the project by selecting menu Project->Make (F7).
5. Make sure that the eZ430 USB stick is plugged into a USB port of your PC
6. Download the code to the eZ430 by selecting menu Project->Debug.
7. After successfully downloading the code, the application should stop at `main()`. You can run/debug the code by selecting menu Debug->Go (F5). Please read the IAR Embedded Workbench User's Guide for more information on debugging the application with the IAR C-Spy debugger.
8. The following Figure 3 shows the PELICAN crossing application executing in the IAR Embedded Workbench.

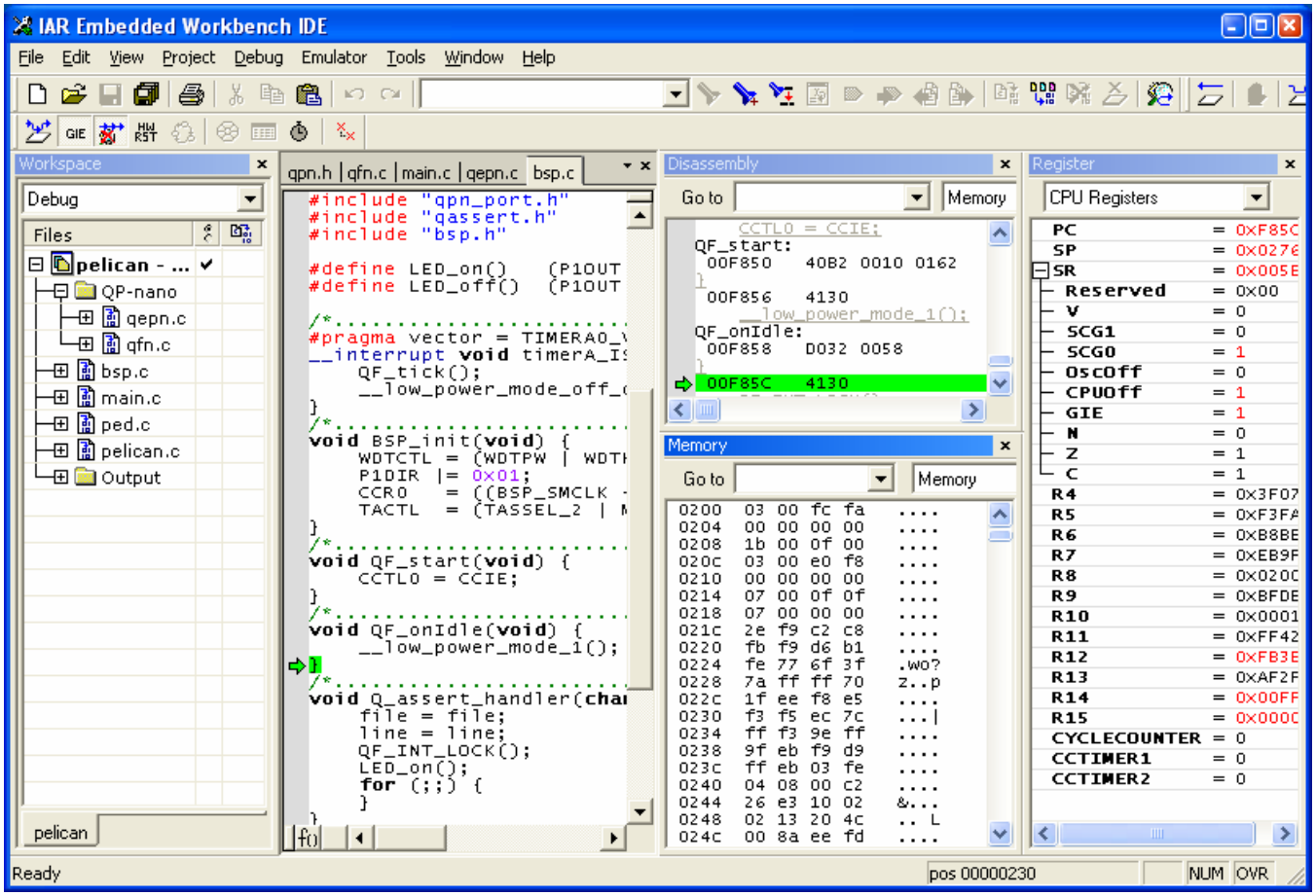

Figure 4 The PELICAN crossing example executing in IAR Embedded Workbench

Figure 3 shows the PELICAN example running in the IAR Embedded Workbench for MSP430. With the eZ430, the whole operation of the PELICAN crossing must be visualized by means of only one LED. The PELICAN crossing signals are shown as follows:

Signal for Cars:	**Green**	**Yellow**	**Red**
Not shown	Not shown	Not shown	Not shown

Signal for Pedestrians:	**WALK**	**Blank**	**DON'T WALK**
LED	LED off	LED off	LED on

3.2 The PELICAN Crossing with the IAR Simulator

You can experiment with the MSP430 code even without the actual eZ430 hardware. The IAR Embedded Workbench contains the Simulator, which can perform cycle-exact simulation of the code execution. The PELICAN crossing workspace has been specifically configured for the Simulator, and even can generate the periodic time-tick interrupt to drive the application through all its paces.

Here are the steps to build and execute the PELICAN example in the IAR simulator:

1. Install the IAR Embedded Workbench for MSP430 KickStart™ edition that is available for a fee download (after online registration) from http://www.iar.com/ew430.
2. Open the IAR Embedded Workbench for MSP430
3. Open the IAR PELICAN workspace by selecting menu File->Open->Workspace… The IAR PELICAN crossing workspace file is located in `<qpn>\examples\msp430\iar\pelican-eZ430\pelican.eww`
4. Build the project by selecting menu Project->Make (F7).
5. Select the Project->Options… menu to change the debug options of the project
6. Select the "Debugger" category on the left-hand panel.
7. Select "Simulator" driver from the drop-down box; click OK to close the dialog box.
8. Start debugging by selecting menu Project->Debug (Ctrl+D)
9. After successfully downloading the code, the application should stop at `main()`.
10. The IAR Simulator allows for simulating the interrupts, and the periodic time-tick interrupt is already configured for you by the macro `sim_setup.mac`.
11. You can run/debug the code by selecting menu Debug->Go (F5). Please read the IAR Embedded Workbench User's Guide for more information on debugging the application with the C-Spy debugger. When you select Debug->Go, the application will not display anything, but you can verify that's running by placing breakpoints in the code. For example, you can set a breakpoint at the timer-tick ISR (`timerA_ISR()`) in the bsp.c module.

3.3 The PELICAN Crossing Example in Turbo C++ 1.01

The PELICAN crossing example for Turbo C++ 1.01 comes in two variants: non-preemptive and preemptive. The non-preemptive PELICAN crossing example is located in the directory <qpn>\-examples\80x88\tcpp101\pelican\. The preemptive variant with the QK-nano real-time kernel is located in the directory <qpn>\examples\80x88\tcpp101\pelican-qk\. Both variants use exactly the same state machine code and differ only in the QP-nano configuration and interrupt handling in the BSP. The procedures for building and executing the two PELICAN crossing examples are identical.

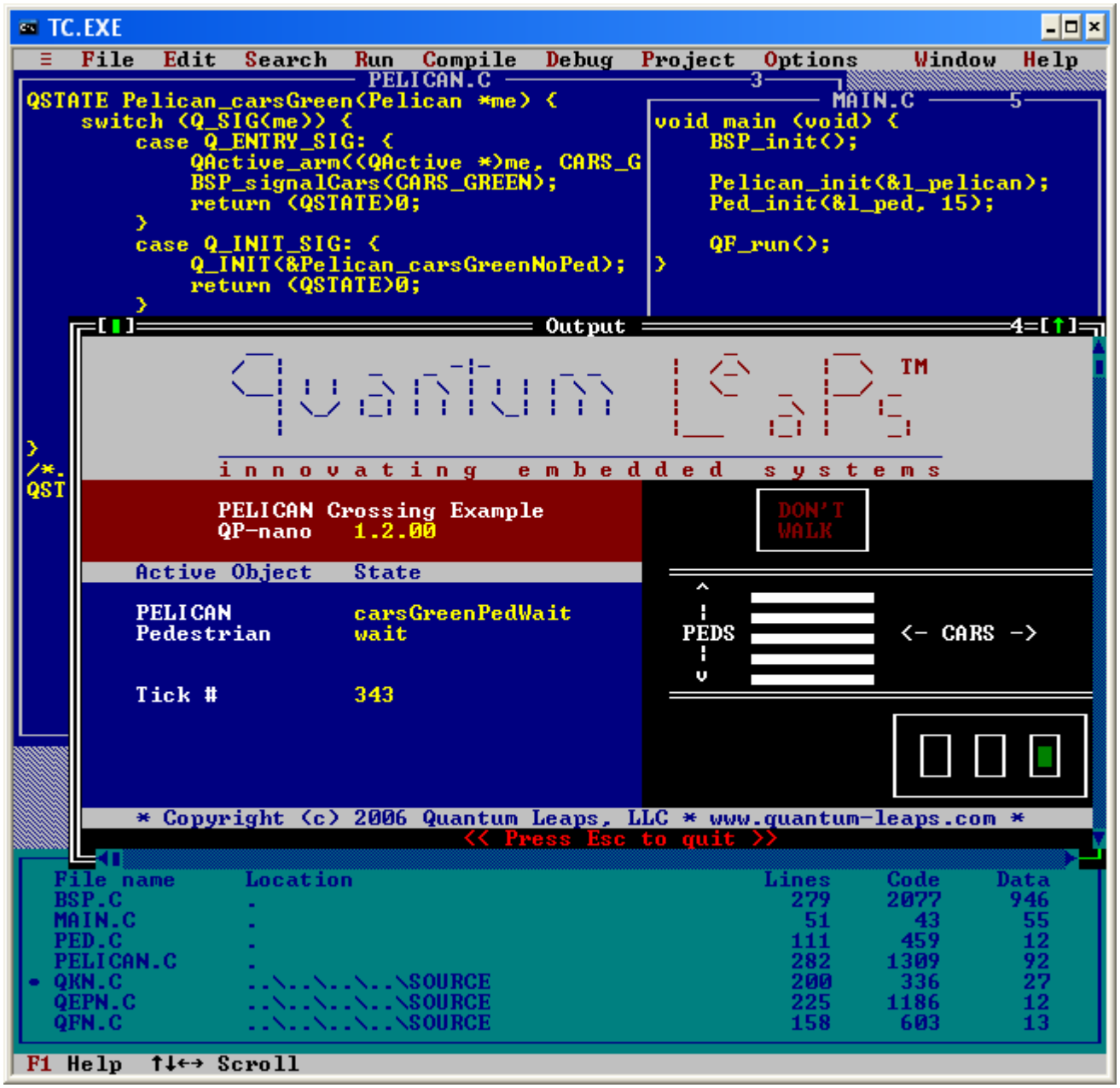

Figure 5 PELICAN-QK crossing example running in the Turbo C++ 1.01 IDE.

You can execute the PELICAN crossing examples on any Windows PC by simply double clicking on the executable located in <qpn>\examples\80x88\tcpp101\pelican\dbg\PELICAN.EXE, for the non-preemptive version, or <qpn>\examples\80x88\tcpp101\pelican-qk\dbg\PELICAN.EXE, for the preemptive version. You can also build and debug the applications using the Turbo C++ 1.01 IDE (see Figure 4).

1. Open the Turbo C++ 1.01 IDE by double-clicking on the TC.EXE executable in the Turbo C++ installation directory
2. Open the PELICAN project by selecting menu Project->Open Project…. The Turbo C++ PELICAN crossing project file is located in <qpn>\examples\80x88\tcpp101\pelican\PELICAN.PRJ (or <qpn>\-examples\80x88\tcpp101\pelican-qk\PELICAN.PRJ)
3. Build the project by selecting menu Compile->Build All.
4. Optionally set breakpoint(s) by opening the desired file(s) (double click on the file in the Project window) and toggle the breakpoint by selecting menu Debug->Toggle Breakpoint.
5. Run the example by selecting menu Run->Run. The following Figure 4 shows the PELICAN crossing application executing in the Turbo C++ IDE.

The output window of the running PELICAN application (see Figure 4) displays information in two panels. The blue panel to the left contains the current information about the state of all state machines in the application. The black panel to the right shows a schematic PELICAN crossing with signals for pedestrians (the rectangle displaying "DON'T WALK") and cars (the street light showing green in the Figure 3). While the application is running, you could correlate the state of the state machines with the signals displayed at the pedestrian crossing.

3.4 The Time Bomb Example on the eZ430 USB Stick

The time bomb example demonstrates QP-nano's support for basic, non-hierarchical Finite State Machines (FSMs). Basic FSMs are a subset of HSMs, so any FSM can be easily coded as an HSM. However, the implementation of FSM is so small and efficient that it is provided as a compile-time option in QP-nano. The support for FSM with entry and exit actions requires typically less than 50 bytes of code (ROM) and just a few bytes of RAM per state machine. (The RAM cost of a FSM is the same as an HSM.)

The Time Bomb example for the eZ430 is located in the directory `<qpn>\examples\msp430\iar\bomb-eZ430\`. You need to use IAR Embedded Workbench for MSP430 to build the application and download it to the eZ430 stick.

Here are the steps to build, load and execute the Time Bomb example:

1. Install the IAR Embedded Workbench for MSP430 that comes on the eZ430 CD-ROM, Install the USB drivers for the ez430 USB stick, as described in the eZ430 User's Guide.
2. Open the IAR Embedded Workbench for MSP430
3. Open the IAR Bomb workspace by selecting menu File->Open->Workspace... The IAR Bomb workspace file is located in `<qpn>\examples\msp430\iar\bomb-eZ430\bomb.eww`
4. Build the project by selecting menu Project->Make (F7).
5. Make sure that the eZ430 USB stick is plugged into a USB port of your PC
6. Download the code to the eZ430 by selecting menu Project->Debug.
7. After successfully downloading the code, the application should stop at `main()`. You can run/debug the code by selecting menu Debug->Go (F5). Please read the IAR Embedded Workbench User's Guide for more information on debugging the application with the C-Spy debugger.

3.5 Designing the PELICAN Crossing Application with QP-nano

This section walks you through a design of the PELICAN crossing application.

Problem specification: *The PELICAN crossing operates as follows. When the crossing is first turned on, cars are enabled (have green light) and pedestrians disabled (have "don't walk" signal). To activate the traffic light switch, a pedestrian must push a button (let's call this event PEDS_WAITING). In response, the cars get the yellow light. After a few seconds, cars get a red light and pedestrians get the "walk" signal, which shortly thereafter changes to a flashing "don't walk" signal. When the "don't walk" signal stops flashing, cars get the green light. After this cycle, the traffic lights don't respond to the PEDS_WAITING event immediately, although the button "remembers" that it has been pushed. The traffic light controller always gives the cars a minimum of several seconds of green light before repeating the cycle. Additionally, the PELICAN crossing may be not-operational, in which case the cars get a flashing yellow light, and pedestrians flashing "don't walk" signal.*

In order to remove the requirement for any user input, the PELICAN example is made self-reliant by simulating the Pedestrian. In fact, this offers an opportunity to introduce other concurrently executing state machine in the system, besides the PELICAN crossing controller.

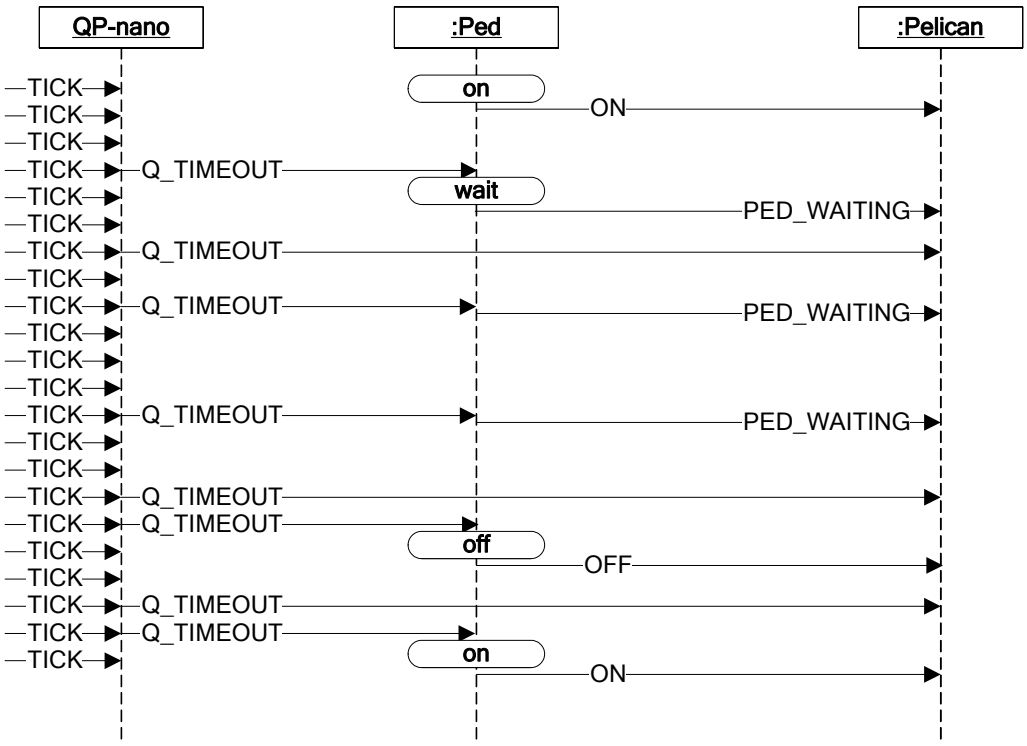

Figure 6 Sequence diagram of the PELICAN crossing example.

The sequence diagram shown in Figure 6 illustrates the operation of the PELICAN crossing application. Basically, everything is driven from the periodic TICK event delivered to the system by the clock-tick ISR (see Section ??). QP-nano maintains a private Time Event (timer) for each state machine object in the system. These timers can be programmed (armed) to generate the Q_TIMEOUT events in specified number of clock ticks. For example, the Ped (pedestrian) state machine starts in the "on" state and arms its timer to fire in specified number of clock ticks. When the timer expires,

QP-nano posts the Q_TIMEOUT event to the Ped state machine, which changes the state to "wait" and posts PED_WAITING event to the Pelican state machine. Independently, the PELICAN state machine arms its private timer to time its own operation. The PED_WAITING, ON, OFF, and private Q_TIMEOUT events drive the Pelican state machine through its own paces, which is shown with less detail in the sequence diagram in Figure 6.

3.5.1 Designing the PELICAN Statechart

The PELICAN state machine is by far the most interesting in this application because it takes advantage of sate nesting. Due to the hierarchical character of statecharts, you can approach the design from the top down or bottom up. This design walkthrough uses a combination of both approaches.

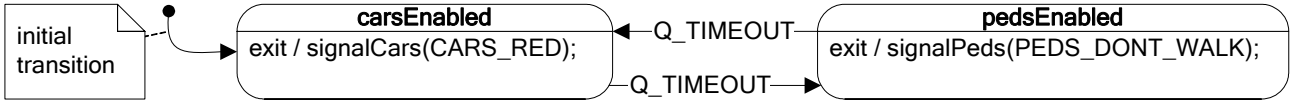

Figure 7 First step of elaborating the PELICAN statechart.

Figure 7 shows the first step in elaborating the PELICAN crossing statechart. The main function of the crossing is to alternate between the "carsEnabled" and "pedsEnabled" states. Obviously, both states need some substates to realize the details of the specification, but let's ignore this at first. At this stage, you should only make sure that the design guarantees mutually exclusive access to the crossing, which is the main safety concern here. Please note that the exit action from the "pedsEnabled" state disables pedestrians, and the exit action from "carsEnabled" disables cars. Now, due to the guarantee of cleanup, these exit actions will be executed whichever way the states happen to be exited, so I can be sure that the pedestrians have always don't-walk signal outside the "pedsEnabled" state and cars have the red light outside the "carsEnabled" state.

The signaling of the PELICAN crossing is accomplished by the functions `signalCars()` and `signalPed()`, which are provided in the Board Support Package (BSP), since they are obviously dependent on the target board.

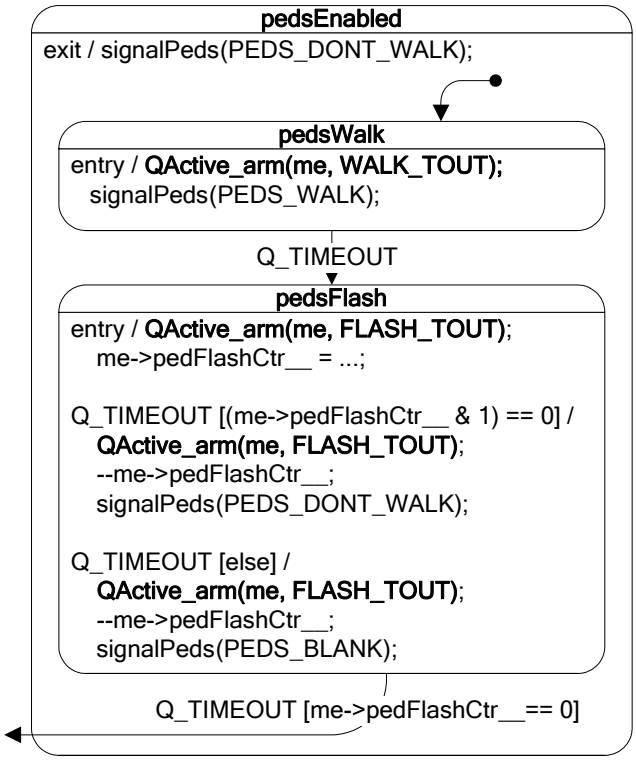

Figure 8 Elaborating the internal structure of the "pedsEnabled" state.

Figure 8 shows the internal structure of the "pedsEnabled" state. The job of the "pedsWalk" substate is to display the walk signal and time out. The job of the "pedsFlash" substate is to turn the don't-walk signal on and off. All actions are triggered by the Q_TIMEOUT events, which are generated by the timer object associated with the state machine. The QP-nano function `QActive_arm()` arms the timer for a one-shot delivery of the Q_TIMEOUT event in the specified number of clock ticks. In the substate "pedsFlash", the Q_TIMEOUT event triggers two *internal transitions* and one regular transition leading out of the state. Internal transitions in UML are different from regular transitions, because the internal transitions never cause execution of state exit or entry. Internal transitions have also a distinctive notation that is similar to the entry and exit actions. The two internal transitions in state "pedsFlash" have different guard conditions (the Boolean expressions in the square brackets), which means that they are enabled only if the conditions in the square brackets evaluate to TRUE. The guard conditions are based in this case on the internal counter `pedFlashCtr__` that controls the number of flashes of the don't-walk signal.

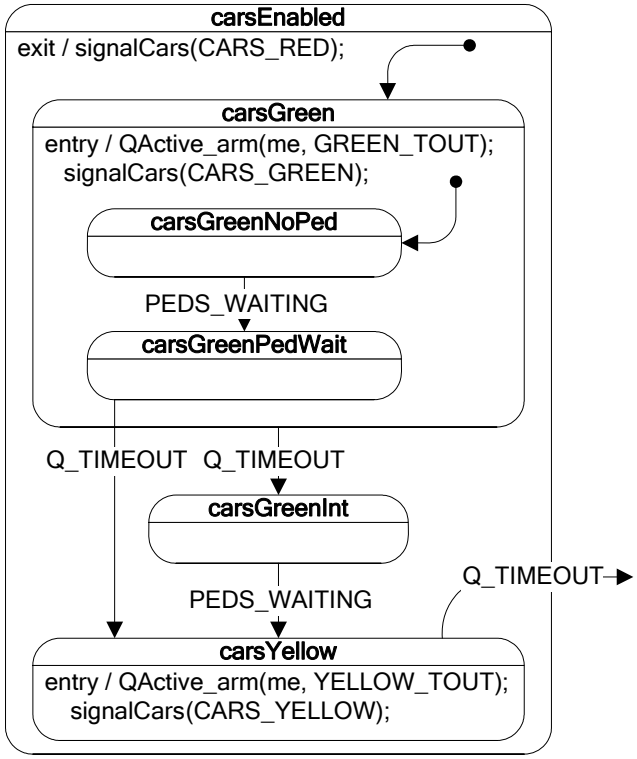

Figure 9 Elaborating the internal structure of the "carsEnabled" state.

Figure 9 shows the internal structure of the "carsEnabled" state. The most interesting problem here is to guarantee the minimum green light for cars before enabling pedestrians. Upon entry to the "carsGreen" substate, the timer is armed to expire after the minimum green light time. When the PEDS_WAITING event arrives before the expiration of this timeout, the active state is "carsGreen-NoPed", and the state machine transitions to the substate "carsGreenPedWait", which has the purpose of "remembering" that a pedestrian is waiting. When the minimum green light time expires in the "carsGreenPedWait" state, it triggers the Q_TIMEOUT transition to the "carsYellow" state, which after another timeout transitions out of "carsEnabled" state to open the crossing to pedestrians. However, if the PEDS_WAITING event does not arrive before the minimum green light timer expires the state machine will be in the "carsGreenNoPed" state that does not prescribe how to handle the Q_TIMEOUT event. Per the semantics of state nesting, the event is passed to the higher-level state, that is, to "carsGreen", which handles the Q_TIMEOUT event in the transition to "carsGreenInt" (interruptible green light).

At this point, the statechart accomplishes the main functionality of the PELICAN crossing. The design progressed top-down, by gradually elaborating the inner structure of hierarchical states. However, you can also design statecharts in the bottom-up fashion. In fact, this is the best way to add the last feature: the "offline" mode of operation.

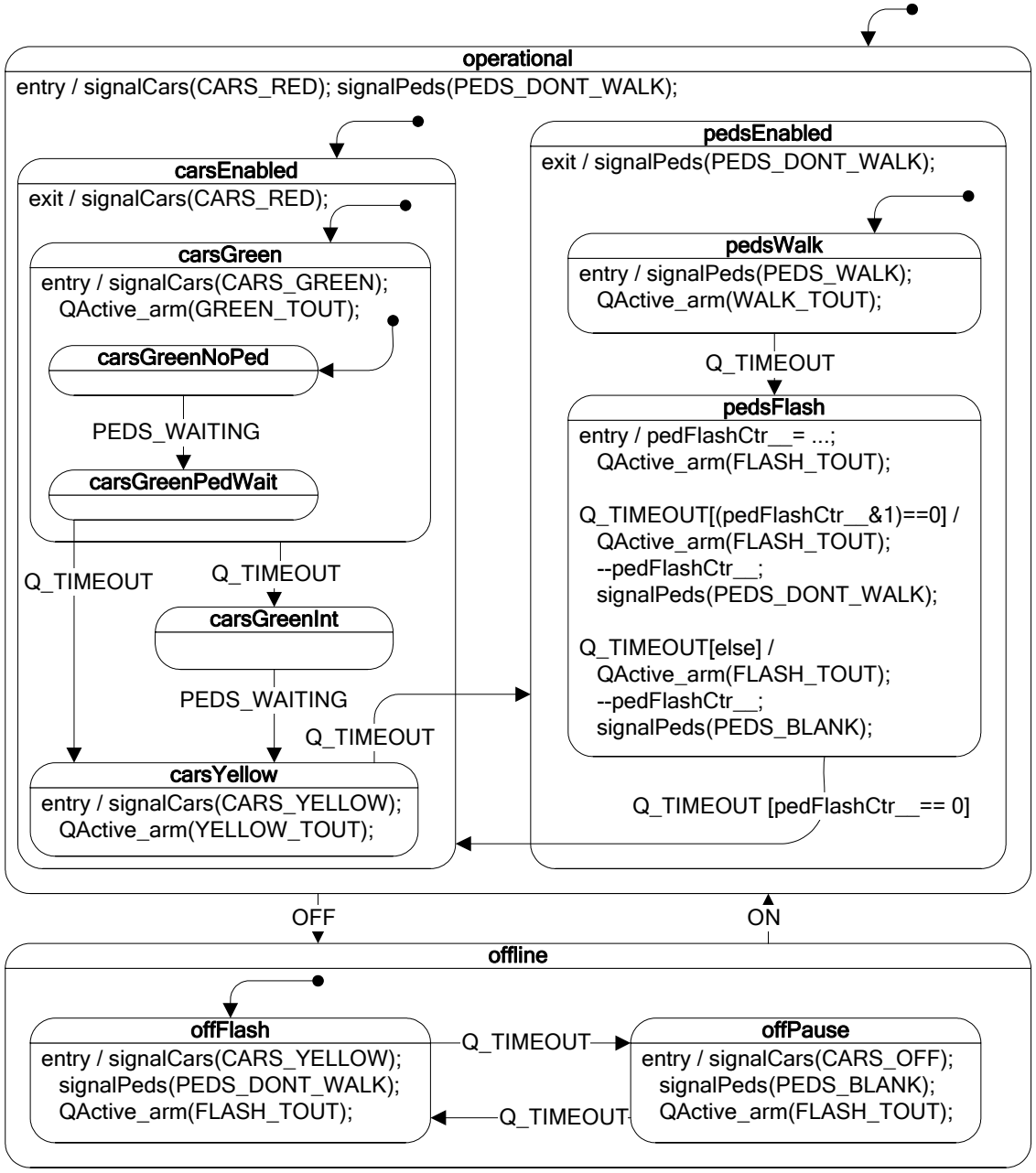

Figure 10 Complete state machine specifying the PELICAN crossing behavior

The state diagram in Figure 10 shows the final PELICAN crossing statechart. The "offline" mode of operation is added simply by enclosing the whole state machine elaborated in the previous steps inside the superstate "operational" that handles the transition OFF to the "offline" state. Please note how the state hierarchy ensures that the transition OFF is inherited by all substates of the "operational" superstate, so regardless in which substate the state machine happens to be, the OFF event always triggers transition to "offline". Now, imagine how difficult it would be to make such a last-minute change to a traditional, non-hierarchical FSM.

Another interesting aspect of the PELICAN state model is the generation of the Q_TIMEOUT events by means of only one timer (Time Event) available to the Pelican active object. (A time event is a facility that dispatches the Q_TIMEOUT event to the state machine after a preprogrammed time delay.) Note how the state machine handles the same Q_TIMEOUT event differently in different states and how entry actions are used to arm the timer.

3.5.2 Designing the Pedestrian Statechart

As indicated in the PELICAN crossing sequence diagram (Figure 6), the Ped (pedestrian) state machine goes through a very simple lifecycle starting with switching the crossing "on", then "wait" at the crossing, and finally switching the crossing "off".

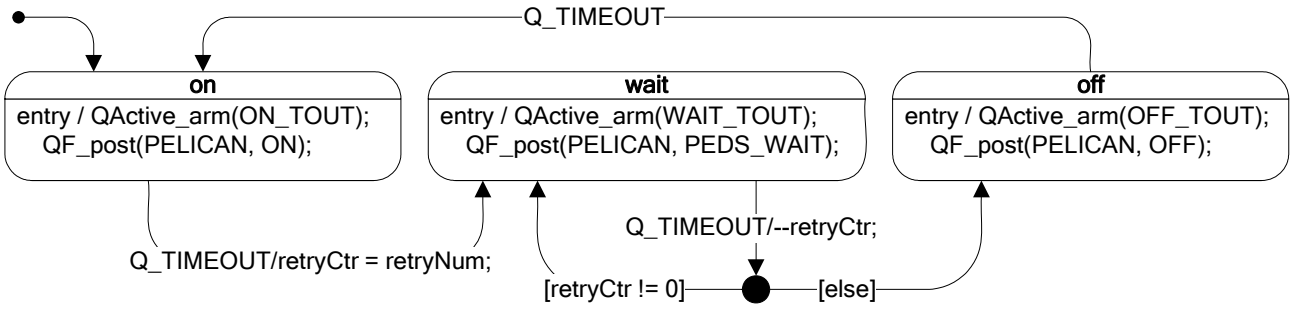

Figure 11 State machine Ped that simulates the Pedestrian in the PELICAN crossing.

As shown in Figure 11, the Ped state machine starts off in "on", where it arms the timer for ON_TOUT ticks and waits its expiration.

> **NOTE:** Each state machine in QP-nano has its own private Time Event (timer), so the Q_TIMEOUT events delivered to the Ped state machine are different than the Q_TIMEOUT events delivered to the PELICAN state machine. QP-nano handles or these timeout requests in parallel.

Upon receiving the Q_TIMEOUT event, the state machine transitions to "wait", where it generates the PED_WAITING event to the PELICAN state machine using the QP-nano facility `QF_post()`. In the "wait" state the timer is armed to fire in WAIT_TOUT clock ticks. Upon the expiration of this timeout, the internal counter of the Ped state machine retryCtr__ is decremented. If this counter is not zero (see the guard condition after the junction point), the state machine takes a self-transition back to the "wait" state. A self-transition causes execution of exit and entry actions, which means that the PED_WAITING event is posted again to the PELICAN state machine, and the timer is armed again. On the other hand, if the counter drops to zero, the [else] guard evaluates TRUE in the Q_TIMEOUT transition, and the state machine transitions to "off".

3.6 The Time Bomb Example

The time bomb example demonstrates QP-naon's support for basic, non-hierarchical Finite State Machines (FSMs). Basic FSMs are a subset of HSMs, so any FSM can be easily coded as an HSM. However, the implementation of FSM is so small and efficient that it is provided as an optimization in the QEP. You can use this optimization for performance-critical portions of your designs, such as inside interrupts or device drivers. Also, you can use FSMs in very resource-constraint embedded systems that simply cannot fit the full-featured HSMs. The support for FSM requires typically less than 50 bytes of code (ROM) and just a few bytes of RAM per state machine.

3.6.1 Designing the Time Bomb Application with QP-nano

In order to focus the discussion, this section walks you through a design of a simple "Time Bomb" application.

Problem specification: *The Time Bomb is created with the pre-programmed timeout, but is initially inactive. To activate the Bomb, a user must arm it by providing the ARM event. After arming, the Bomb starts to count down, blinking a built-in LED once every second. When the timeout expires, the Bomb "explodes".*

In order to remove the requirement for any user input, the Time Bomb example is made self-reliant by simulating an Operator. In fact, this offers an opportunity to introduce other concurrently executing state machine in the system, besides the Bomb state machine. In this case, both the Bomb and the Operator are implemented as non-hierarchical FSMs.

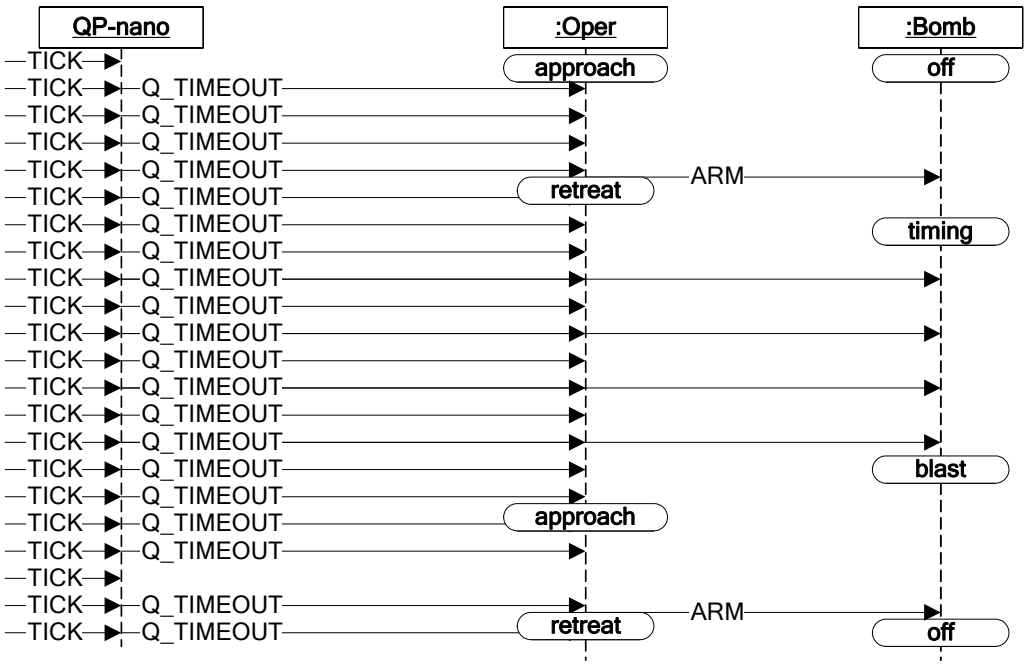

Figure 12 Sequence diagram of the Time Bomb example

The sequence diagram shown in Figure 13 illustrates the operation of the Time Bomb application. Basically, everything is driven from the periodic TICK event delivered to the system by the clock-tick ISR (see Section ??). QP-nano maintains a private timer (Time Event) for each state machine object in the system. These timers can be programmed (armed) to generate the Q_TIMEOUT

events in specified number of clock ticks. For example, upon receiving the ARM event the Bomb state machine transitions to the "timing" state, where it arms its timer to fire in specified number of clock ticks. When the timer expires, QP-nano posts the Q_TIMEOUT event to the Bomb state machine, which downcounts its internal counter and blinks the LED. Independently, the Oper (operator) state machine uses its own timer to post the ARM events to the Bomb.

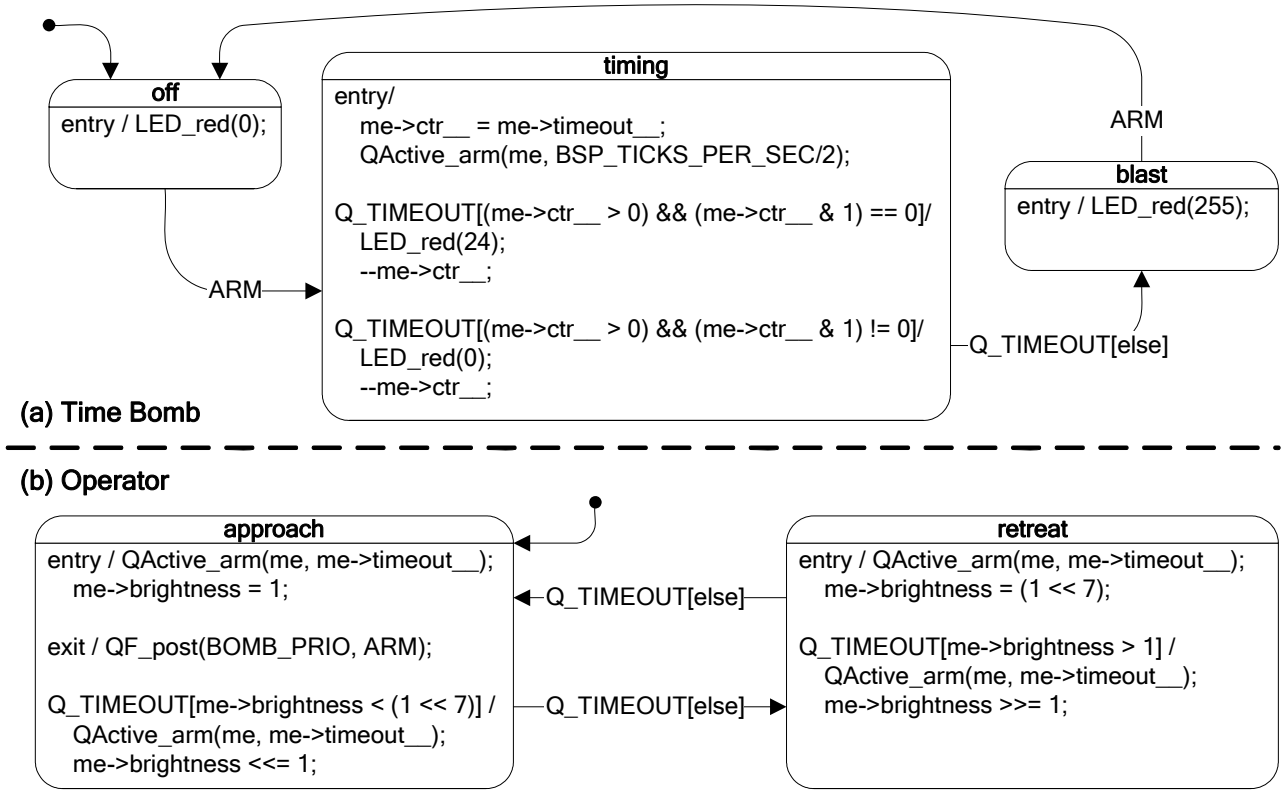

Figure 13 The Time Bomb FSM (a) and the Oper (operator) FSM (b)

Figure 14 shows the FSM that models the Bomb and the Operator. The Bomb FSM starts in state "`off`" after the top-most initial transition. The entry action to this state turns off the Red LED (`LED_red(0)`). The ARM signal causes a state transition to "timing", where upon entry the down-counter `me->ctr__` is initialized to the full timeout value and the time event is armed to deliver the Q_TIMEOUT event in half a second (the number of ticks per second divided by 2). The Q_TIMEOUT signal triggers different actions depending on the value of the downcounter `me->ctr__` (note the guard conditions in the state "timing"). When the counter is not zero and is even, the Red LED is set to low brightness. When the counter is not zero and is odd, the Red LED is extinguished. In both cases the counter is decremented. Otherwise, the counter is zero and the Time Bomb transitions to the "blast" state, where upon the entry the Red LED is set to full brightness (`LED_red(255)`).

The Operator state machine is shown in panel (b) of Figure 14. Upon entry, the time event is armed to expire in me->timeout__ clock ticks and the LED brightness is set to lowest value of 1. Upon the Q_TIMEOUT event the brightness is incremented, if it is not saturated and the time event is re-armed. Otherwise the Operator state machine transitions to "retreat" state. Upon the exit from "approach" the Operator posts the ARM event to the Bomb state machine. The "retreat" state is basically a mirror image of "approach", except the Red LED brightness is incremented in logarithmic steps.

4 Implementing PELICAN Example with QP-nano

Contrary to widespread misconceptions, you can use concurrently executing UML state machines even in very small microcontrollers (such as the "F2013" in the eZ430) and you don't need sophisticated design automation tools to translate UML state machines to efficient and highly *maintainable* C. This section explains step-by-step how to develop the PELICAN crossing application with QP-nano.

The source code for the PELICAN application is located in two directories: `<qpn>\examples\80x86\tcpp101\pelican` contains code for DOS compiled with Turbo C++ 1.01, while the directory `<qpn>\examples\msp430\iar\pelican-eZ430\` contains the code for the eZ430 USB stick compiled with IAR Embedded Workbench for MSP430 v3.40A. The only difference between these two code sets is in the Board Support Package (BSP). In particular, the state machine definitions, and even the `main()` functions in these two code sets are exactly identical. Here is the list of the source files comprising the PELICAN crossing application:

1. `qpn_port.h` contains the platform-specific customization of QP-nano (the QP-port)
2. `pelican.h` contains declaration of signals, events, as well as the `PELICAN` and `Ped` (Pedestrian) active objects.
3. `bsp.h` contains the Board Support Package interface (BSP)
4. `bsp.c` contains the implementation of the BSP, which includes all platform-specific QF-nano callbacks.
5. `pelican.c` contains the implementation of the `Pelican` state machine.
6. `ped.c` contains the implementation of the `Ped` state machine.
7. `main.c` contains the initialization of the QF-nano and active objects followed by the call to `QF_run()`.

4.1 Step 1: Configuring and Customizing QP-nano

You configure and customize QP-nano through the header file `qpn_port.h`, which is included in the QP-nano source files (`qepn.c`, `qfn.c`, and `qkn.c`) as well as in all your application C modules.

```
(1) #define Q_ROM
(2) #define Q_PARAM_SIZE            1
(3) #define QF_TIMEEVT_CTR_SIZE     2

(4) #define Q_NFSM
(5) /* #define QF_FSM_ACTIVE */
(6) #define QF_MAX_ACTIVE           4

                                        /* interrupt locking policy for MSP430 */
(7) #define QF_INT_LOCK(key_)       __disable_interrupt()
(8) #define QF_INT_UNLOCK(key_)     __enable_interrupt()

(9) #include <intrinsics.h>  /* contains prototypes for the intrinsic functions */
(10)#include "qepn.h"         /* QEP-nano platform-independent public interface */
(11)#include "qfn.h"          /* QF-nano platform-independent public interface */
```

Listing 2 qpn_port.h header file for the USB Toolstick and IAR MSP430 compiler

Listing 2(1) The `Q_ROM` macro is used inside the QP-nano source code to denote all constant objects that can be allocated in ROM. On CPUs with the Harvard architecture (e.g., the 8051), the code and data spaces are separate and are accessed through different instructions. The compilers often provide specific extended keywords to designate code or data space, such as the `__code` extended keyword in the IAR 8051 compiler. To use QP-nano with a different MCU/compiler, you need to check how to allocate constants to ROM. On Von Neumann CPUs (such as the MSP430), you can always define the `Q_ROM` macro as empty.

Listing 2(2) The `Q_PARAM_SIZE` macro defines the size in bytes of the scalar event parameter. The valid values are 0, 1, 2, and 4, which correspond to no event parameter, `uint8_t`, `uint16_t`, and `uint32_t` parameter. You might also not define this macro at all, which is equivalent to specifying no parameter.

NOTE: QP-nano has been specifically designed for small systems with very limited RAM. In this minimal version events are represented as structures containing the byte-wide enumerated type of the event, such as Q_TIMEOUT or PED_WAITING. In UML, the type of the event is called the *signal*. Optionally, every event can have in QP-nano a single scalar event parameter. Event parameters are very useful to convey the quantitative information associated with the event. For example, an ADC conversion might generate an event with the signal ADC_READY and the parameter containing the value produced by the ADC.

Listing 2(3) The `QF_TIMEEVT_CTR_SIZE` macro defines the size in bytes of the time event down-counter (timer counter). The size of this counter determines the dynamic range of timeouts (in clock tick units). The valid values are 0, 1, 2, and 4, which correspond to no time event counter, `uint8_t`, `uint16_t`, and `uint32_t` counter. You might also not define this macro at all, which is equivalent to specifying no time event counter.

Listing 2(4) The macro `Q_NFSM` eliminates the FSM code. You cannot use the simple FSMs, only the HSMs. For symmetry, you might also define the macro `Q_NHSM` that eliminates the HSM code. If you define `Q_NHSM`, you cannot use the HSMs, only the simple FSMs in your application.

Listing 2(5) The macro `QF_FSM_ACTIVE` specifies that active objects are the simple FSMs. The default is that active objects are HSMs. Obviously, `QF_FSM_ACTIVE` is not compatible with `Q_NFSM`.

Listing 2(6) The macro `QF_MAX_ACTIVE` defines the maximum number of active objects. In QP-nano, this number cannot exceed 8. However, defining `QF_MAX_ACTIVE` as 4 or less slightly improves performance of the QF-nano/QK-nano schedulers.

Listing 2(7-8) The macros `QF_INT_LOCK()` / `QF_INT_UNLOCK()` define the CPU and compiler-specific interrupt locking mechanism. For the IAR compiler, the intrinsic functions `__disable_interrupt()` and `__enable_interrupt()` are used.

Listing 2(9) The `<intrinsics.h>` header file is included for the definition of the intrinsic functions.

Listing 2(10) The `qpn_port.h` header file must always include the platform-independent QEP-nano header file `qep.h`.

Listing 2(10) The `qpn_port.h` header file includes the platform-independent QF-nano header file `qfp.h`, because QF-nano™ is used in this port.

4.2 Step 2: Declaring Signals and Events

This step of the implementation consists of enumerating all signals used in the application, such as the signals PEDS_WAITING, ON, and OFF identified in the sequence diagram in Figure 6.

NOTE: The relationship between an event and a signal is as follows. A *signal* in UML is the specification of an asynchronous stimulus that triggers reactions [OMG 03], and as such is an essential part of an event. (The signal conveys the type of the occurrence—what happened?) However, an event can also contain additional quantitative information about the occurrence in form of *event parameters*. Please refer to the document *"Brief Introduction to UML State Machines"* (http://www.quantum-leaps.com/devzone/Recipe_IntroHSM.pdf) for more information about state machine concepts.

The following Listing 3 shows the enumeration of all signals used in the PELICAN application. (The generic Q_TIMEOUT signal is already defined in the QP-nano.) Note that the user-level signals do not start from zero but rather are offset by the constant Q_USER_SIG. This is because QEP reserves the lowest few signals for the internal use and defines the constant Q_USER_SIG as an offset from which user-level signals can start. Please also note that by QP-nano convention, all signals have the suffix _SIG to easily distinguish signals from other constants. The suffix _SIG is omitted in the state diagram to reduce the clutter.

```
enum PelicanSignals {
    PEDS_WAITING_SIG = Q_USER_SIG,
    OFF_SIG,
    ON_SIG
};
```

Listing 3 Enumeration of all signals recognized by the PELICAN application.

4.3 Step 3: Declaring Active Object Structures

From the application developer's perspective, the most important step in constructing a QP-nano application is conceiving the specific active objects. As in all other active object–based frameworks, QP-nano provides a base structure for deriving concrete active objects. In the QP-nano, this base structure is QActive. This structure combines the following three essential elements: (1) it is a state machine (derives from the QHsm or QFsm structure), (2) it has an event queue, and (3) it owns a private Time Event (timer).

At the application level, however, you can mostly ignore all the other aspects of an active object and view it only as a state machine. This abstraction is possible because other elements of an active object, like the event queue or timer, work transparently behind the scenes to provide the embedded statechart with events and CPU cycles. One of the biggest advantages of QP-nano is the support for rapid construction of *executable* models. The framework helps you to put together active objects that make up the application quickly and then to elaborate gradually their internal details (mostly their state machines), keeping the application executable at all times.

4.3.1 Declaring the PELICAN Active Object

The following Listing 4 shows the derivation of the Pelican active object from the QActive base structure.

```
(1) typedef struct PelicanTag Pelican;
    struct PelicanTag {
(2)     QActive super_;                                  /* derived from QActive */
(3)     uint8_t pedFlashCtr__;                        /* pedestrian flash counter */
    };
(4) void Pelican_ctor(Pelican *me);
```

Listing 4 Derivation of the Pelican structure from the QActive base structure.

Listing 4(1) For convenience, `struct PelicanTag` is typedefed to `Pelican`. Subsequently only the name `Pelican` is used.

Listing 4(2) In this line of code you derive the Pelican structure from the `QActive` structure. You achieve it very simply by literally embedding the `QActive` object as the first data member of the derived structure (i.e., `Pelican` in this case). Per the QP-nano coding convention, the base struct member is named `super_`.

NOTE: You should convince yourself that the memory alignment resulting from placing the `super_` attribute as the first member of the `Pelican` structure is such that you can always safely cast any pointer pointing to `Pelican` on pointer to `QActive`. In other words, you can think of the `Pelican` structure as derived from, or inheriting from the `QActive` structure, because every function designed to work with a pointer to `QActive` will continue to work correctly when you pass to it a pointer to `Pelican` instead.

The derivation of structures is not limited to only one level. In fact, the `QActive` structure itself is derived from the `QHsm` (Hierarchical State Machine) structure inside QP-nano. This means that all functions designed to work with `QHsm` pointers will continue to work correctly when pointers to the `Pelican` structure are used instead.

Listing 4(3) You can add any number of arbitrary parameters after the `super_` member. In this case, a byte-size counter for counting the number of don't-walk signal flashes is added.

Listing 4(4) You declare the constructor function for the `Pelican` objects. Please note that the function takes a parameter of the type `Pelican *`. Per the QP-nano coding convention, all functions designed to work with the Pelican struct have the "`Pelican_`" prefix, and all take the `Pelican*` pointer as the first parameter. This first parameter is always named "`me`".

4.3.2 Declaring Other Active Objects

All other active objects in the application are declared similarly to the `Pelican` active object. All of them are declared in the main application include file `pelican.h`.

```
(1) typedef struct PedTag Ped;
(2) struct PedTag {
        QActive super_;                                  /* derived from QActive */
        uint8_t retry__;
        uint8_t retryCtr__;
    };
    void Ped_ctor(Ped *me, uint8_t retry);
```

Listing 5 Derivation of the Ped (pedestrian) active object.

4.4 Step 4: Declaring PELICAN States

Each state in QP-nano is represented as a C function called a state handler function. The job of QP-nano is to invoke these state handler functions in the right order to process events according to the UML semantics.

A state handler function is a regular C function that takes the state machine pointer as the argument and returns a pointer to another state handler function, which is typedefed as `QSTATE` in the QP-nano header file `qpn.h`. You need to structure your state handler functions such that they return the pointer to the superstate handler, if they don't handle the current event, or a `NULL`-pointer, if they handle the current event. QP-nano uses this information to "learn" about the nesting of states to process events hierarchically and correctly execute state transitions.

```
static QSTATE Pelican_offline(Pelican *me);           /* state handler */
static QSTATE Pelican_offFlash(Pelican *me);          /* state handler */
static QSTATE Pelican_offPause(Pelican *me);          /* state handler */
static QSTATE Pelican_operational(Pelican *me);       /* state handler */
static QSTATE Pelican_carsEnabled(Pelican *me);       /* state handler */
static QSTATE Pelican_carsGreen(Pelican *me);         /* state handler */
static QSTATE Pelican_carsGreenNoPed(Pelican *me);    /* state handler */
static QSTATE Pelican_carsGreenPedWait(Pelican *me);  /* state handler */
static QSTATE Pelican_carsGreenInt(Pelican *me);      /* state handler */
static QSTATE Pelican_carsYellow(Pelican *me);        /* state handler */
static QSTATE Pelican_pedsEnabled(Pelican *me);       /* state handler */
static QSTATE Pelican_pedsWalk(Pelican *me);          /* state handler */
static QSTATE Pelican_pedsFlash(Pelican *me);         /* state handler */
```

Listing 6 State handler functions for the PELICAN crossing HSM

4.5 Step 5: Instantiating the Active Objects

The initialization of active objects is separated into two steps. In the first step, you instantiate the objects by explicitly calling their constructors, typically from the `main()` function during the system initialization. As always, the subclass constructor is responsible for invoking the superclass' (`QActive`) constructor:

```
void Pelican_ctor(Pelican *me) {
    QActive_ctor((QActive *)me, &Pelican_initial); /* call the superclass' ctor */
}
. . .
void Ped_ctor(Ped *me, uint8_t retry) {
    QActive_ctor((QActive *)me, &Ped_initial);     /* call the superclass' ctor */
    me->retry__ = retry;                           /* initialize the attributes */
}
```

The second parameter required by the QActive_ctor() constructor is the top-most "initial" pseudostate described in the next section.

4.6 Step 6: Initializing the HSM

The second step of state machine initialization is executing the top-most initial transition.

The top-most initial transition of a hierarchical state machine requires some attention because in general case, it can be quite involved. For example, the top-most initial transition in the PELICAN crossing statechart (Figure 10) consists of the following steps: (1) entry to "operational", (2) initial

transition in "operational", (3) entry to "carsEnabled", (4) initial transition in "carsEnabled", (5) entry to "carsGreen", (6) initial transition in "carsGreen", and finally (7) entry to "carsGreenNoPed". Of course, you want QP-nano to deal with this complexity, which it can actually do, but in order to reuse the already implemented mechanisms, you need to execute the top-most initial transition as a regular transition.

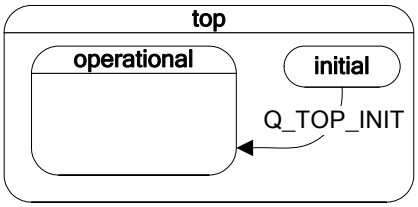

Figure 14 The top-most initial pseudostate for the PELICAN state machine.

Figure 14 illustrates how you do it. You need to provide a special pseudostate "initial" that handles the top-most initial transition as a regular state transition. The "initial" state is nested directly in the top state, which is the UML concept that denotes the ultimate root of the state hierarchy. QP-nano defines the top state handler function `QHsm_top()`, which by default "handles" all events by returning `NULL` pointer. ("Handling" events in the top state means really silently discarding them, per the UML semantics.)

The actual example of the "initial" state is shown in Listing 7. The structure of the "initial" pseudostate is exactly the same as any other state (see the following section), except the "initial" pseudostate defines only one transition, as shown in Figure 14.

```
(1) QSTATE Pelican_initial(Pelican *me) {
        switch (Q_SIG(me)) {
(2)         case Q_TOP_INIT_SIG: {
(3)             Q_TRAN(&Pelican_operational); /* take the initial transition */
(4)             return (QSTATE)0;
            }
        }
(5)     return (QSTATE)&QHsm_top;
}
```

Listing 7 The "initial" pseudostate of the PELICAN state machine

Listing 7(1) As explained in the next section, the initial pseudostate handler function has the same signature as the regular state handler function.

Listing 7(2) The initial pseudostate needs to implement only one state transition (the top-most initial transition). This transition is triggered by the reserved signal `Q_TOP_INIT_SIG`.

Listing 7(3) The target of the top-most initial transition is designated by means of the macro `Q_TRAN()`.

Listing 7(4) The initial pseudostate returns `NULL`. when it handles the transition.

Listing 7(5) The initial pseudostate must nest in the top state provided by the `QHsm` base class.

4.7 Step 6: Implementing the State Handler Functions

In the next step, you actually code the state machine by implementing one state at a time as a state handler function. To determine what elements belong the any given state handler function, you follow around the state's *boundary* in the diagram. You need to implement all transitions *originating* at the boundary, any entry and exit actions defined, as well as all internal transitions enlisted directly in the state. Additionally, if there is an initial transition embedded *directly* in the state, you need to implement it as well.

Take for example the state "carsGreen" shown in Figure 10. It has one transition Q_TIMEOUT originating at its boundary. (Note: the diagrams typically omit the suffix _SIG to reduce the clutter). In addition, the "carsGreen" state has an entry action and the initial transition to state "carsGreenNoPed".

4.7.1 General Structure of a State Handler Function

Listing 8 shows the definition of the state handler function Pelican_carsGreen() corresponding to the "carsGreen" state.

```
(1) QSTATE Pelican_carsGreen(Pelican *me) {
(2)     switch (Q_SIG(me)) { /* switch on signal of the current event */
(3)         case Q_ENTRY_SIG: {                        /* entry action */
                QActive_arm((QActive *)me, CARS_GREEN_MIN_TOUT);
                BSP_signalCars(CARS_GREEN);
(4)             return (QSTATE)0;                     /* event handled */
            }
(5)          case Q_INIT_SIG: {
(6)             Q_INIT(&Pelican_carsGreenNoPed);/* initial transition */
(7)             return (QSTATE)0;                     /* event handled */
            }
(8)         case Q_TIMEOUT_SIG: {
(9)             Q_TRAN(&Pelican_carsGreenInt);    /* state transition */
(10)            return (QSTATE)0;                     /* event handled */
            }
        }
(11)    return (QSTATE)&Pelican_carsEnabled;        /* the superstate */
    }
```

Listing 8 State handler function for the "carsGreen" state.

Each state handler must have the same signature, that is, it must take the state machine pointer "me" and it must return QSTATE, which is a pointer to the superstate handler function. (QSTATE is typedef'ed in the QP-nano header file qpn.h.)

Generally, every state handler is structured as a big switch that discriminates based on the signal of the current event. To reduce the number of arguments of the state handler function, QP-nano stores the current event in the state machine object pointed to by the "me" pointer. For convenience, QP-nano provides the macro Q_SIG() to access the signal of the event. Each case is labeled by an enumerated signal and terminates with "return (QSTATE)0". Returning a zero-pointer from a state handler informs the event processor that the particular event has been processed. On the other hand, if no case executes, the state handler exits through the final return statement, which returns the pointer to the superstate handler function (QSTATE)&Pelican_carsEnabled in this case (11). Please note that the final return statement from a state handler function is the only place where you specify the hierarchy of states. Therefore, this one line of code represents the single point of maintenance for changing the nesting level of a given state.

NOTE: The final return statement from a state handler function is the *only* place where you specify the hierarchy of states. Therefore, this one line of code represents the *single point of maintenance* for changing the nesting level of a given state.

While coding state handlers you need to keep in mind that QP-nano treats them as the specification of the state machine. QP-nano will invoke a state handler function for various reasons: for hierarchical event processing, for execution of entry and exit actions, for triggering initial transitions, or even just to elicit the superstate of a given state handler. Therefore, you should not assume that a state handler would be invoked only for processing signals enlisted in the case statements. You should avoid any code outside the `switch` statement, especially code that would have side effects.

4.7.2 Coding Entry and Exit Actions

QP-nano provides two reserved signals `Q_ENTRY_SIG` and `Q_EXIT_SIG` that the event processor passes to the appropriate state handler function to execute the state entry actions or an exit actions, respectively.

Therefore, as shown at label (3) of Listing 8, to code a state *entry actions*, you provide a case statement labeled with signal `Q_ENTRY_SIG`, enlist all the actions you want to execute upon the entry to the state, and terminate the lists with "`return (QSTATE)0`", which informs the QEP that the entry actions have been handled.

Coding the *exit actions* is identical, except that provide a case statement labeled with signal `Q_EXIT_SIG`, all the actions you want to execute upon the exit from the state, and terminate the lists with "`return (QSTATE)0`", which informs the QEP that the exit actions have been handled.

NOTE: Newcomers to QEP and the UML statechart formalism often try to code state transitions in entry or exit actions. While it seems doable in QEP (see the upcoming Section 3.6.5 "Coding Regular Transitions"), it typically crashes the event processor. When you think about it, taking a transition while entering a state doesn't really make sense and the UML semantics do not allow it. Actually, you cannot even draw a state transition from an entry or an exit action in a UML state diagram. A transition in an entry action represents a malformed state machine and violates one of the well-formedness rules of the UML [OMG 03].

4.7.3 Coding Initial Transitions

Every composite state (a state with substates) can have its own initial transition, which in the diagrams is represented as an arrow originating from a black ball. For example, state "`carsGreen`" has such a transition to substate "carsGreenNoPed".

The QP-nano provides a reserved signal `Q_INIT_SIG` that the event processor passes to the appropriate state handler function to execute the initial transition.

Therefore, as shown at label (5) in Listing 8, to code an *initial transition*, you provide a case statement labeled with signal `Q_INIT_SIG`, enlist all the actions you want to execute upon the initial transition, and then designate the target substate with the `Q_INIT()` macro. You terminate the case statement with "`return (QSTATE)0`", which informs the QEP that the initial transition has been handled.

NOTE: In QP-nano you must necessarily use the `Q_INIT()` macro for the initial transitions, and you should not code an initial transition with the macro `Q_TRAN()`, which is reserved for the regular state transitions (see the upcoming Section 3.6.5 "Coding Regular Transitions").

The UML semantics require the target of the initial transition must be a direct or indirect substate of the source state. An attempt to target a non-substate (e.g., a peer state, or a superstate) corresponds to a malformed state machine and can crash the event processor.

Please also note that initial transitions cannot have guard conditions (see the upcoming Section ?? "Coding Guard Conditions").

4.7.4 Coding Regular Transitions

State "carsGreen" has one regular state transition to "carsGreenInt", which is triggered by signal Q_TIMEOUT (see Figure 10). To code such a *regular transition* (see label 8 in Listing 8), you provide a case statement labeled with the triggering signal (`Q_TIMIEOUT_SIG`), enlist the actions, and then designate the target state with the `Q_TRAN()` macro. You terminate the case statement with "`return (QSTATE)0`", which informs the QP-nano that the event has been handled.

The `Q_TRAN()` macro can accept any target state at any level of nesting, such as a peer state, a substate, a superstate, or even the same state as the source of the transition (transition to self).

NOTE: The QP-nano *automatically* handles execution of appropriate exit and entry actions during arbitrary state transitions. Consequently, any change in state machine topology (change in state transitions or state nesting), requires only re-compilation of the state handler functions for the QEP to automatically take care of figuring out the correct sequence of exit/entry actions and initial transitions to execute for every state transition.

4.7.5 Coding Guard Conditions

Guard conditions (or simply guards) are Boolean expressions evaluated dynamically based on the value of the variables associated with the state machine (extended state variables) and/or event parameters. Guard conditions affect the behavior of a state machine by disabling actions or transitions when they evaluate to `FALSE`, and enabling them when they evaluate to `TRUE`. In the UML notation, guards are shown in square brackets immediately following the corresponding trigger (e.g. `Q_TIMEOUT [pedFlashCtr__ == 0]` in Figure 10).

```
    QSTATE Pelican_pedsFlash(Pelican *me) {
        switch (Q_SIG(me)) {
            case Q_ENTRY_SIG: {
                BSP_showState(PELICAN_PRIO, "pedsWalk");
                QActive_arm((QActive *)me, PEDS_FLASH_TOUT);
                me->pedFlashCtr__ = PEDS_FLASH_NUM*2 + 1;
                return (QSTATE)0;
            }
(1)         case Q_TIMEOUT_SIG: {
(2)             if (me->pedFlashCtr__ != 0) {               /* still flashing? */
                    if ((me->pedFlashCtr__ & 1) == 0) {       /* even counter? */
(3)                     BSP_signalPeds(PEDS_DONT_WALK);
                    }
                    else {                                /* must be odd counter */
                        BSP_signalPeds(PEDS_BLANK);
                    }
                    --me->pedFlashCtr__;
                    QActive_arm((QActive *)me, PEDS_FLASH_TOUT);
                }
                else {                                          /* done flashing */
                    Q_TRAN(&Pelican_carsEnabled);
                }
                return (QSTATE)0;
            }
```

```
        }
        return (QSTATE)&Pelican_pedsEnabled;
    }
```

Listing 9 State handler function for the "pedsFlash" state

Listing 9 shows an example of a state transition with a guard in the "pedsFlash" state of the PELICAN state machine from Figure 10. The guard condition maps simply to an `if`-statement that conditionally executes the transition.

4.7.6 Coding Internal Transitions

Internal transitions (a UML term) are not transitions at all. Internal transitions are simple reactions to events that never lead to change of state, and consequently never cause execution of exit actions, entry actions, or initial transition. The "pedsFlash" state in the PELICAN crossing statechart (see Figure 10) provides an example of internal transition.

To code an *internal transition* (see Listing 9(1)), you provide a case statement labeled with the triggering signal (e.g., Q_TIMEOT_SIG), enlist the actions, and terminate the list with "`return (QSTATE)0`" to inform the QEP that the event has been handled.

Guard conditions are allowed for internal transitions as well. In all cases a guard maps to the `if`-statement. The only difference for the internal transition is that you don't invoke the `Q_TRAN()` macro.

4.7.7 Transition Execution Sequence in QP-nano

The examples of a state handlers in Listing 8 and Listing 9 show that state handler functions implement state transitions by means of the `Q_TRAN()` macro. More specifically, the state handler corresponding to the *source* of the transition invokes the macro and specifies the target as the argument. This simple implementation cannot be compliant with the UML specification, which prescribes the following transition execution sequence [OMG 03, Section 13.3.14]:

1. execution of guard conditions (if defined for the transition);
2. execution of exit actions from the source state configuration;
3. execution of actions associated with the transition; and
4. execution of entry actions to the target state configuration.

Instead, the QP-nano implementation executes the following transition sequence:

1. execution of guard conditions (if defined for the transition);
2. execution of actions associated with the transition;
3. execution of exit actions from the source state configuration; and
4. execution of entry actions to the target state configuration.

As you can see, the steps 2 and 3 of the UML sequence are reversed in QP-nano. The QP-nano cannot strictly implement the UML specification because actions and guards in QP-nano are always defined in the state handler function corresponding to the *source* of the transition. It would be very awkward and counter-intuitive to separate the execution of the guards (the `if`-statements) from the actions on transitions (the body of the `if`).

However, the UML transition sequence is altered in QP-nano for some other reasons as well. The exit actions from the source state perform typically a cleanup—or destruction—of the exited state context. This destruction of state context proceeds up to the level of the *Least Common Ancestor* (LCA) state of the source and target states [OMG 03]. (The LCA is the lowest-hierarchy state that contains both the source and the target states.) Nevertheless, the UML specification prescribes executing the actions on transition only after exiting the stable source state configuration, which is the context of the LCA. But in general the LCA context does not correspond to any *stable* state configuration. In particular, some extended state variables often used in the actions on transitions might not yet be initialized because they have been destroyed in the exit actions, which according to the UML specification happen before the actions on transition.

In contrast, QP-nano always executes the actions on transition in the well-defined context of the source stable state configuration, before any cleanup can happen in the exit actions. The change of state, on the other hand, occurs atomically, meaning that no other processing interrupts the chain of exit actions from the source state configuration and entry to the target state configuration.

Finally, please note that the QP-nano transition execution sequence does not compromise any fundamental benefits of HSMs, like programming-by-difference or guaranteed initialization and cleanup via entry and exit actions (see the article "Deja Vu" [Samek 03c]). The QP-nano sequence still preserves the essential order of exiting a nested source state (from the most deeply nested state up the hierarchy to the LCA) and entering a target state (from the LCA state down the hierarchy to the target).

4.7.8 Local Transition Semantics

Local state transitions are a new addition in UML 2.0 [OMG 03 Section 15.3.14]. In contrast to the external transitions used in UML 1.x, local transitions don't cause exit from a composite state if the target state is a substate of the source. In addition, local state transitions don't cause exit and re-entry to the target state if the target is a superstate of the source.

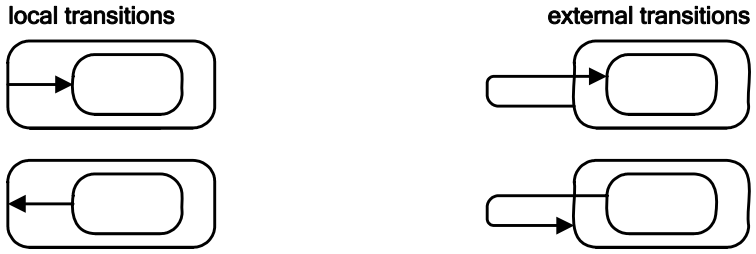

Figure 15 Local vs. External transitions. QP-nano implements only the local transitions.

Figure 16 contrasts local (left side) and external (right side) transitions. In the top row, you see the case of the source containing the target. The local transition does not cause exit from the source, while the external transition causes exit and re-entry to the source.

In the bottom row of Figure 16, you see the case of the target containing the source. The local transition does not cause entry to the target, while the external transition causes exit and re-entry to the target.

4.8 Error Management in QP-nano

The policy for managing errors in QP-nano is perhaps different than in other traditional C libraries. QP-nano does not raise SIG_ERR signals, does not use the errno facility and you'll not see any error-code returns from the QP-nano functions.

Instead, QP-nano uses *assertions* to detect internal errors or to flag incorrect use of the QP-nano services that indicates errors in the application code.

4.8.1 Embedded-Systems-Friendly Assertions

The QP-nano uses embedded-systems-friendly assertions that are very similar to the standard <assert.h> facility, except the behavior of the QP-nano assertions is more customizable. The standard C-library macro assert() is rarely applicable to embedded systems because its default behavior when the expression passed to the macro evaluates to FALSE is to print an error message and exit. Neither of these actions make much sense for most embedded systems, which rarely have a screen suitable for printing an error message, and cannot really exit either (at least not in the same sense as a desktop application can). Therefore, in an embedded environment, you usually have to define your own assertions that suit your tools and allow you to customize the error response.

```
    #ifndef qassert_h
    #define qassert_h

(0) #ifdef Q_NASSERT          /* Q_NASSERT defined--assertion checking disabled */

        #define Q_DEFINE_THIS_FILE
        #define Q_DEFINE_THIS_MODULE(name_)
        #define Q_ASSERT(ignore_)  ((void)0)
        #define Q_ALLEGE(test_)    ((void)(test_))
        #define Q_ERROR()          ((void)0)

    #else                   /* Q_NASSERT not defined--assertion checking enabled */

 (1)     #define Q_ASSERT(test_) \
            if (test_) { \
            } \
            else (Q_assert_handler(l_this_file, __LINE__))

(2)     #define Q_ALLEGE(test_)    Q_ASSERT(test_)

(3)     #define Q_ERROR() \
            (Q_assert_handler(l_this_file, __LINE__))

(4)     void Q_assert_handler(char const Q_ROM *file, int line);

(5)     #define Q_DEFINE_THIS_FILE \
            static char const Q_ROM l_this_file[] = __FILE__;

(6)     #define Q_DEFINE_THIS_MODULE(name_) \
            static char const Q_ROM l_this_file[] = #name_;

    #endif                                                  /* NASSERT */

    #define Q_REQUIRE(test_)   Q_ASSERT(test_)
    #define Q_ENSURE(test_)    Q_ASSERT(test_)
    #define Q_INVARIANT(test_) Q_ASSERT(test_)

    #define Q_ASSERT_COMPILE(test_) \
        extern char Q_assert_compile[(test_)]
```

```
#endif                                                  /* qassert_h */
```

Listing 10 qassert.h header file.

Listing 10 shows file qassert.h located in the <qpn>\include directory. The header file contains customizable assertions that have proven adequate for a very wide range of embedded projects. The file qassert.h is similar to the standard <assert.h>, except: (1) it allows customizing the error response, (2) it conserves memory by avoiding proliferation of multiple copies of the filename string, and (3) it provides additional macros for testing and documenting preconditions (Q_REQUIRE), postconditions (Q_ENSURE), and invariants (Q_INVARIANT), alongside the general-purpose Q_ASSERT() macro.

The all-purpose Q_ASSERT() macro (Listing 10(1)) is very similar to the standard assert(). If the "test_" argument passed to this macro evaluates to 0 (FALSE) and if additionally the macro Q_NASSERT is not defined, then Q_ASSERT() will invoke a global callback Q_assert_handler(). The function Q_assert_handler() Listing 10(4) gives the application developers the opportunity to customize the error response when the assertion fails. In embedded systems, Q_assert_handler() typically first disables all interrupts and then attempts to put the system in a fail-safe state before performing a system reset. (Many embedded systems come out of reset in a fail-safe state, so putting them in this state before the reset is often unnecessary.) If possible, the assertion handler callback should also leave a trail of bread crumbs of the cause, perhaps storing the filename and line number in a non-volatile memory. (The Q_assert_handler() function is also an ideal place to set a breakpoint if you work with a debugger.)

Compared to the standard assert() macro, the QP-nano macro Q_ASSERT() conserves memory (typically ROM) by passing l_this_file as the first argument to Q_assert_handler(), rather than the standard preprocessor macro __FILE__. This avoids proliferation of the multiple copies of the __FILE__ string for each instance of the Q_ASSERT() macro, but requires invoking macro Q_DEFINE_THIS_FILE (Listing 10(5)) or Q_DEFINE_THIS_MODULE (Listing 10(6)), preferably at the top of every C file. Additionally, QP-nano provides a provision for directing the compiler to allocate the file strings in ROM (see the Q_ROM macro). For example, for the 8051 IAR compiler Q_ROM could be defined __code (see Listing 2(1)).

NOTE: QP-nano has been carefully designed not to waste the precious RAM for any information available at compile time. The module file names (l_this_file[] string) are examples of such compile-time information and are allocated in the code-space by the Q_ROM modifier. Q_ROM is a macro defined in qpn_port.h that for Harvard architectures instructs the compiler to allocate an object to the code space (ROM). For example, for the IAR 8051 compiler you can define Q_ROM as __code. In von Neumann CPU architectures (such as the MSP430), the compiler will typically allocate all const objects to ROM, so the Q_ROM modifier is superfluous (should be defined to nothing).

Defining the macro Q_NASSERT (Listing 10(0)) disables checking the assertions. When disabled, the assertion macros don't generate any code; in particular they don't test the expressions passed to them as arguments, so you should be careful to avoid any side-effects required for the normal program operation inside the expressions tested with assertions. The notable exception is the macro Q_ALLEGE(), which always tests the expression, although when assertions are disabled, it does not invoke the Q_assert_handler() callback. Q_ALLEGE() is useful in situations where avoiding side-effects of the test would require introducing temporary objects, which involves additional stack usage—something you often want to minimize in embedded systems.

4.8.2 Using Assertions in QP-nano Applications

In order to build your QP-nano application, you need to define the assertion-failure handler callback: `Q_assert_handler()`. All the QP-nano examples provide a basic implementation for the callback.

The most important point to understand about assertions is that they neither prevent nor handle errors, in the same way as contracts between people do not prevent fraud. For example, asserting that a non-zero event pointer will be passed to the initial pseudostate of the time bomb application: `Q_REQUIRE(e != (QEvent const *)0)`, might give you a warm and fuzzy feeling that you have handled or prevented passing an invalid event pointer, when in fact, you haven't. You did establish a precondition contract, however, in which you spelled out that passing an invalid event pointer to this initial pseudostate is an error. From that point on the preconditoin will be checked automatically and sure enough the program will brutally abort if the contract fails. At first you might think that this must be backwards. Contracts not only do nothing to handle (let alone fix) bugs, but they actually make things worse by turning every asserted condition, however benign, into a fatal error! However, the first priority when dealing with bugs is to detect them, not to handle them. To this end, a bug that causes a loud crash (and identifies exactly which contract was violated) is much easier to find than a subtle one that manifests itself intermittently millions of machine instructions downstream from the spot where you could have easily detected it.

The use of assertions in QP-nano is really a part of bigger methodology called Design By Contract (DBC). Please consult the references at the end of this Manual for more information about the subject.

4.8.3 Shipping with Assertions

The standard practice is to use assertions during development and testing, but to disable them in the final product by defining the `NDEBUG` macro. In the `qassert.h` header file (Listing 10), this macro is replaced with `Q_NASSERT`, because many development environments define NDEBUG automatically when you switch to the production version, and the QP-nano assertions intentionally decouple the decision of disabling assertions from the version of software that you build. This leaves you an option to leave assertions *enabled* in the ship-version of the product.

The often-quoted opinion in this matter comes from C.A.R. Hoare, who considered disabling assertions in the final product like using a lifebelt during practice, but then not bothering with it for the real thing.

The question of shipping with assertions really boils down to two issues. First is the overhead that assertions add to your code. Obviously, if the overhead is too big, you have no choice. (But then you must ask yourself how will you build and test your firmware?) It's much better to consider assertions an integral part of the firmware and size the hardware adequately to accommodate them. As the prices of hardware rapidly drop while the capabilities increase, it just makes sense to trade a small fraction of the raw CPU horsepower and some memory resources (ROM) for a better system integrity. As turns out, assertions often pay for themselves by eliminating reams of "error handling" code that tries to camouflage bugs.

The other issue is the correct system response when an assertion fires in the field. As it turns out, a simple system reset is for most embedded devices the least inconvenient action from the user's perspective—certainly less inconvenient than locking-up a device and denying service indefinitely.

4.9 Initializing and Starting the QP-nano Application

After you've coded all state machines, you need to tell QP-nano about them, so that it can start managing the state machines as active objects comprising the application.

In the non-preemptive configuration, QP-nano executes all active objects in the system in a run-to-completion (RTC) fashion, meaning that each active object completely handles the current event before processing the next one. After each RTC step, QP-nano engages a simple scheduler to decide which active object to execute next. The scheduler makes this decision based on the priority statically assigned to each active object upon the system startup (priority-based scheduling). The scheduler always executes the highest-priority active object that has some events in its event queue.

```
(1) #include "qpn_port.h"                                      /* QP-nano port */
    #include "bsp.h"                                 /* Board Support Package (BSP) */
    #include "pelican.h"                                /* application interface */

    /*..........................................................................*/
(2) static Pelican l_pelican;
(3) static Ped     l_ped;

(4) static QEvent  l_pelicanQueue[1];

    /*..........................................................................*/
    /* CAUTION: the QF_active[] array must be initialized consistently with
    * the priority assignement in pelican.h
    */
(5) QActiveCB const Q_ROM QF_active[] = {
(6)     { (QActive *)0,          (QEvent *)0,    0                      },
(7)     { (QActive *)&l_pelican, l_pelicanQueue, Q_DIM(l_pelicanQueue) },
(8)     { (QActive *)&l_ped,     (QEvent *)0,    0                      }
    };
(9) uint8_t const Q_ROM QF_activeNum = Q_DIM(QF_active) - 1;

    /*..........................................................................*/
    void main (void) {
(10)    BSP_init();                                    /* initialize the board */

(11)    Pelican_init(&l_pelican);
(12)    Ped_init(&l_ped, 15);           /* init Ped, set # PED_WAITING attempts */

(13)    QF_run();                                  /* transfer control to QF-nano */
    }
```

Listing 11 Definition of the state machine objects and the main() function.

Listing 11(1) You customize QP-nano in the `qpn_port.h` header file, which contains extensive comments explaining all the options.

Listing 11(2-3) You statically allocate all active objects.

Listing 11(4) You statically allocate all event queue buffers used by active objects. Please note, that the highest-priority active object in the system, such as Pedestrian, might not need an event queue buffer at all, because the single event stored inside the state machine itself might be sufficient.

Listing 11(5) You define and initialize a constant array `QF_active[]`, in which you configure all the active objects in the system in the order of their relative priority.

NOTE: QP-nano has been carefully designed not to waste the precious RAM for any information available at compile time. The `QF_active[]` array is an example of such compile-time information and is allocated in the code-space by the `Q_ROM` modifier.

The `QF_active[]` array must be initialized consistently with the priorities assigned to the active objects through the enumeration in the application header file (e.g., `pelican.h`).

Listing 11(6) You leave the `QF_active[0]` priority level unused. The lowest priority level 0 is reserved for the QK-idle loop (in case you configure preemptive kernel). The priority levels for active objects are numbered from 1 to `QF_MAX_ACTIVE` inclusive, and higher priority number represents higher urgency.

Listing 11(7-8) You define all active object entries in the `QF_active[]` array. Please note that the initialization of the `QF_active[]` array **must** be consistent with priorities enumerated in the application header file (`pelican.h` in case of the PELICAN crossing example).

Listing 11(9) You define and initialize a constant `QF_activeNum`, which is the total number of active objects actually used in the system. Please note that the zero-element of the `QF_active[]` is unused, so the number of active objects in the application is the dimension of the `QF_active[]` array minus 1. (The `Q_DIM()` macro determines the dimension of an array and is defined in `qpn.h`)

Listing 11(10) The `main()` function starts with initialization of the Board Support Package

Listing 11(11-12) You need to initialize all active objects in the application. In particular, this initialization involves executing the top-most initial transitions in the active objects' state machines.

Listing 11(13) Finally, you transfer control to QF-nano, which enters an endless loop and never returns to `main()`.

4.10 The Board Support Package (BSP) for QP-nano Application

QP-nano calls several platform-specific callback functions that you must define, typically in the Board Support Package (BSP). Apart from the callbacks, you must also define all the Interrupt Service Routines (ISRs), which will be covered in Section 5.3 "Interrupt Processing in QP-nano".

4.10.1 BSP initialization

Nothing in the board initialization is QP-nano specific. The following `BSP_init()` function from the PELICAN crossing application for the eZ430 stops the watchdog, configures the PIO line for the LED, and sets up the Timer0 to deliver the clock-tick interrupt:

```
void BSP_init(void) {
    WDTCTL = (WDTPW | WDTHOLD);                                  /* Stop WDT */
    P1DIR |= 0x01;                                             /* P1.0 output */
    CCR0   = ((BSP_SMCLK + BSP_TICKS_PER_SEC/2) / BSP_TICKS_PER_SEC);
    TACTL  = (TASSEL_2 | MC_1);                                /* SMCLK, upmode */
}
```

4.10.2 Starting Interrupts in QF_start()

QP-nano invokes the `QF_start()` callback just before starting the background loop (in the non-preemptive case) or the idle loop (in the preemptive case). Either way the `QF_start()` function

must start the interrupts configured earlier. The following QF_start() implementation is from the PELCIAN crossing example:

```
    void QF_start(void) {
        CCTL0 = CCIE;                                 /* CCR0 interrupt enabled */
    }
```

4.10.3 QP-nano Idle Loop Customization in QF_onIdle()

QP-nano can very easily detect the situation when no events are available. In this case QP-nano calls QF_onIdle() callback, which you can use to suspended the CPU to save power, if your CPU supports such a power-saving mode.

The following piece of code shows the QF_onIdle() callback for the eZ430. Here, the F2013 micro-controller is put to LMP1 power saving mode. Only an active interrupt can wake the processor up.

```
    void QF_onIdle(void) {
        __low_power_mode_1();                                  /* Enter LPM1 */
    }
```

4.10.4 Assertion Handling Policy in Q_assert_handler()

As described in Section 4.7.2, QP-nano uses internally assertions to detect errors in the way application is using the QP-nano services. You need to define how the application reacts in case of assertion failure by providing the callback function Q_assert_handler(). Typically, you would put the system in fail-safe state and try to reset. It is also a good idea to log some information as to where the assertion failed.

The following code fragment shows the Q_assert_handler() callback for the eZ430. The function simply locks all interrupts, lights up the LED, and enters a for-ever loop. This policy is only adequate for testing.

```
    void Q_assert_handler(char const Q_ROM *file, int line) {
        file = file;                                  /* avoid compiler warning */
        line = line;                                  /* avoid compiler warning */
        QF_INT_LOCK();                 /* make sure that interrupts are locked */
        LED_on();
        for (;;) {
        }
    }
```

4.10.5 QP-nano Memory Footprint

Because the memory footprint is of primary interest in QP-nano, Listing 12 shows the Module Summary part of the MAP file for the PELICAN crossing on the eZ430.

```
                     ****************************************
                     *            MODULE SUMMARY            *
                     ****************************************
Module                CODE      DATA      CONST
------                ----      ----      -----
                     (Rel)  (Rel)  (Abs)  (Rel)
?Epilogue               18
?__exit                  2
```

```
?_exit                    4
?cstart                  24
?exit                     4
?memzero                 18
?reset_vector                                    2
bsp                      92              10
  + common                                      20
main                     28      26              19
ped                     234
pelican                 600
qepn                    436                      5
qfn                     392       1             38
N/A (command line)                64
----------           -------      --      --      --
Total:               1 852       91      10      64
  + common                                      20
```

Listing 12 The MAP file for the PELICAN application for MSP430 (non-preemptive case). The highlighted modules correspond to the QP-nano components (828 bytes of code (ROM), 43 bytes of constants (ROM) and 1 byte of RAM total).

5 QP-nano Structure

In principle, the previous sections should get you started with QP-nano and you should be able to successfully design and implement concurrent active objects applications. However, you will code with greater confidence and more efficiently when you understand the internal workings of QP-nano. This Section describes some of the internal structural aspects of QP-nano.

5.1 QP-nano Source Code Structure

The QP-nano source code consists of the following files:

```
<qpn>/                    - QP-nano Root Directory
  |
  +-doxygen\              - subdirectory containing the QP-nano reference manual
  | +-html\
  | | +-index.html        - QP-nano reference manual in HTML
  |
  +-include\              - subdirectory containing the QP-nano public interface
  | +-qassert.h           - embedded-systems-friendly assertions used in QP-nano
  | +-qepn.h              - The platform-independent QEP-nano header file
  | +-qfn.h               - The platform-independent QF-nano header file
  | +-qkn.h               - The platform-independent QK-nano header file
  |                         (not available in open source version)
  |
  +-source/               - QP-nano source files
  | +-qepn.c              - QEP-nano implementation
  | +-qfn.c               - QF-nano implementation
  | +-qkn.c               - QK-nano implementation (not available in open source version)
```

Listing 13 QP-nano source code organizatioin

As you can see from Listing 14, the QP-nano source files reflect that QP-nano is consists of three components: QEP-nano, QF-nano, and QK-nano. All QP-nano C-files are CPU- and compiler-independent. The QP-nano customization to a given CPU and compiler is handled entirely through the header file `qpn_port.h`, which is included in all QP-nano modules and your application as well. The `qpn_port.h` header file is platform-specific and you need to provide it as part of your application. Listing 2 shows an example of the `qpn_port.h` header file.

5.2 Critical Sections in QF-nano

QF-nano, just like any other multitasking software, needs to perform certain operations indivisibly. The simplest and most efficient way to protect a section of code from disruptions is to lock interrupts on entry to the section and unlock interrupts again on exit. Such a section of code is called the critical section.

NOTE: The time spent in a critical section should be kept to a minimum because this time directly affects the interrupt latency. However, as long as the critical sections within QF-nano take no more time than when disabling interrupts elsewhere in the system (e.g., inside the ISRs, the underlying multitasking kernel or device drivers), the QF-nano critical sections do not extend the maximum interrupt latency. QF-nano is very carefully designed to disable interrupts only very briefly, which should not affect the maximum interrupt latency.

Processors generally provide instructions to lock/unlock interrupts, and your C compiler must have a mechanism to perform these operations from C. Some compilers allow you to insert directly in-line assembly instructions. Other compilers provide language extensions, or at least C-callable functions to lock and unlock interrupts from C.

The hide the actual implementation method chosen by the compiler vendor, QF provides two macros to lock and unlock interrupts:

```
#define QF_INT_LOCK()       __disable_interrupt()
#define QF_INT_UNLOCK()     __enable_interrupt()
```

Listing 14 QF-nano macros for interrupt locking and unlocking for the IAR compiler.

As shown in Listing 15, QF-nano uses the simplest critical section policy, which is to unconditionally lock interrupts in `QF_INT_LOCK()` and unconditionally unlock them in `QF_INT_UNLOCK()`.This policy is simple and fast, but does not allow nesting of critical sections, because interrupts will be always unlocked upon the exit from a critical section, regardless if they were locked or unlocked upon entry.

The simple QF-nano interrupt locking policy is adequate for most low-end microcontrollers, which typically aren't designed for nesting interrupts. Such micros typically lack a fully prioritized interrupt controller, and cannot afford bigger stack requirements caused by nesting interrupts either.

NOTE: To avoid nesting QP-nano critical sections you must be **very** careful to call only dedicated QP-nano functions from ISRs. These functions are QF_postISR() and QF_tick().

Conversely, you should never call the ISR functions at the task level, because they are not designed to be used at the task context.

Finally, you should not call any QP-nano functions from within your own critical sections that you might have in your applications. Because the standard "task-level" QP-nano services use the simple critical section mechanism, they might leave the interrupts unlocked when you think they should be locked (see also the next section).

However, on processors equipped with an Interrupt Controller, such as most 32-bit CPUs (AT91 family from Atmel, 80x86 with the 8259A PIC, M16C, and many others), you can use the simple unconditional interrupt locking policy and still nest interrupts. You can explicitly enable interrupts inside ISRs, thus avoiding critical section nesting and allowing interrupt nesting, because the Interrupt Controller handles interrupt prioritization *before* the interrupts reach the processor.

Your application, not just QP-nano, can also use `QF_INT_LOCK()` and `QF_INT_UNLOCK()` to protect your own critical sections. You should be careful, however, not to nest critical sections in your code. Obviously, inside your critical sections you cannot call any QP-nano services that use a critical section internally.

5.3 Interrupt Processing in QP-nano

One of the biggest advantages of QP-nano is the simple interrupt processing, which is actually no much more complicated with QP-nano than it is in the simplest of all "super-loop" (a.k.a., main+ISRs). Because QP-nano does not rely in any way on the interrupt stack frame layout, the interrupts service routines (ISRs) can be written entirely in C, if the C compiler supports writing interrupts, which most embedded C compilers actually do.

As described in the previous section, you cannot use the regular "task-level" QP-nano services inside the ISRs. The only two QP-nano services designed to be used inside the ISRs are:

1. QF_postISR() for posting events to active objects from the ISRs.
2. QF_tick() function to be invoked from the periodic time-tick ISR. This function uses QF_postISR() internally.

5.3.1 ISRs: Non-Preemptive Case, No Interrupt Nesting Allowed

In the non-preemptive case your ISRs are identical as in the simplest of all "super-loop" (main+ISRs), and there is nothing QP-specific in the structure of the ISRs.

The only QP-specific requirement is that you provide a periodic time-tick ISR and you invoke QF_tick() in it.

```
(1) #pragma vector = TIMERA0_VECTOR
(2) __interrupt void timerA_ISR(void) {
(3)     __low_power_mode_off_on_exit();
(4)     QF_tick();
    }
```

Listing 15 Time tick interrupt for the eZ430 invoking the QF_tick() function to generate Q_TIMEOUT events.

Listing 15(1-2) The ISR in C is always compiler-specific, as the C standard does not define how to specify ISRs. In the case of the IAR compiler for MSP430, the interrupt must be preceded with the #pragma vector and must use the extended keyword __interrupt.

Listing 15(3) This line causes masking off the power-saving bits in the processor status register saved on the stack. This intrinsic IAR function is specific to the MSP430 and relates to the idle processing described in Section 4.9.3.

Listing 15(4) The time-tick ISR must invoke QF_tick(), and can also perform other things, if necessary. The function QF_tick()cannot be reentered, that is, it necessarily must run to completion and return before it can be called again. This requirement is automatically fulfilled if you call QF_tick() with interrupts locked.

5.3.2 ISRs: Non-Preemptive Case, Interrupt Nesting Allowed

On processors equipped with a prioritized Interrupt Controller, it is possible to nest interrupts without nesting critical sections. This is possible because the interrupt controller handles the interrupt prioritization even before the interrupt reaches the CPU.

This case still allows writing ISRs in C, but requires that you change the policy of locking interrupts inside the QF_postISR() function. You achieve this by defining the macro QF_ISR_NEST in the

qpn_port.h header file. If you do this, you must invoke the QP-nano ISR services with interrupts **unlocked**.

> **NOTE:** The QF_ISR_NEST macro turns on the critical section inside the QF_postISR() function. This means that interrupts must be necessarily unlocked when calling QF_postISR() or QF_tick() because the critical sections cannot nest.
>
> Conversely, unlocking interrupts inside ISRs without defining QF_ISR_NEST in the qpn_port.h header file makes the code vulnerable to concurrency hazards since the event posting occurs outside of critical section. (Please recall that interrupts can now nest on top of interrupts.)

```
(1) interrupt void tickISR(void) {
(2)     /*Clear any level-sensitive interrupt sources before enabling interrupts*/
(3)     enable();    /* enable interrupt nesting (QF_INT_NEST must be defined!) */

(4)     QF_tick();

(5)     disable();        /* disable interrupts for the exit from the interrupt */
(6)     outportb(0x20, 0x20); /* EOI (End-Of-Interrupt) to the master 8259A PIC */
    }
```

Listing 16 Time tick interrupt for the x86 PELICAN example, non-preemptive case (interrupt nesting allowed).

Listing 16(1) The usual "interrupt" extended keyword must be used to inform the C compiler to generate appropriate ISR entry and exit.

Listing 16(2) Any level-sensitive interrupt sources not cleared automatically must be cleared explicitly before unlocking interrupts.

Listing 16(3) Interrupts are enabled, which must be always consistent with defining the QF_ISR_NEST macro in the qpn_port.h header file. At this point interrupt nesting is allowed.

Listing 16(4) The QP-nano ISR service is invoked with interrupts unlocked, which is consistent with the defining the macro QF_INT_NEST in the qpn_port.h header file. The time-tick ISR must invoke QF_tick(), and can also perform other things, if necessary. The function QF_tick() cannot be reentered, that is, it necessarily must run to completion and return before it can be called again. This requirement is in this case fulfilled by the interrupt controller that does not allow the same tick interrupt preempt itself.

Listing 16(5) Interrupts are locked before writing the End-Of-Interrupt (EOI) command to the Interrupt Controller.

Listing 16(6) The EOI command is sent to the Interrupt controller and the interrupt exits.

5.3.3 Preemptive Case, No Interrupt Nesting

You can configure the QP-nano to use the fully-preemptive real-time kernel QK-nano by defining the QK_PREEMPTIVE macro in the qpn_port.h header file. Fortunately, even with the preemptive kernel, the ISRs are very simple and actually almost identical as in the non-preemptive case. The only important difference is that now the structure of ISRs must be augmented to invoke QK_schedule() just before the exit from the ISR. Calling QK_schedule() at the end of each ISR is essential, because the preemptions happen exactly in the QK scheduler.

```
    #pragma vector = TIMERA0_VECTOR
    __interrupt void timerA_ISR(void) {
        __low_power_mode_off_on_exit();
        QF_tick();
(1)     QK_schedule();                  /* handle the asynchronous preemptions */
    }
```

Listing 17 Time tick interrupt for the eZ430 in the preemptive case

As shown in Listing 17, the only difference from the non-preemptive case is calling the `QK_schedule()` function at the end of the ISR. The `QK_schedule()` function must be always called with interrupts locked and also returns with interrupts locked. `QK_schedule()` can unlock interrupts internally.

5.3.4 ISRs: Preemptive Case, Interrupt Nesting Allowed

On processors equipped with a prioritized Interrupt Controller, it is possible to use the QK-nano preemptive kernel, and nest interrupts at the same time. This is possible because the interrupt controller handles the interrupt prioritization even before the interrupt reaches the CPU.

This most advanced case still allows writing ISRs in C, but requires that you change the policy of locking interrupts inside the `QF_postISR()` function. You achieve this by defining the macro `QF_ISR_NEST` in the `qpn_port.h` header file. If you do this, you must invoke the QP-nano ISR services with interrupts **unlocked**.

> **NOTE:** The `QF_ISR_NEST` macro turns on the critical section inside the `QF_postISR()` function. This means that interrupts must be necessarily unlocked when calling `QF_postISR()` or `QF_tick()` because the critical sections cannot nest.
>
> Conversely, unlocking interrupts inside ISRs without defining `QF_ISR_NEST` in the `qpn_port.h` header file makes the code vulnerable to concurrency hazards since the event posting occurs outside of critical section. (Please recall that interrupts can now nest on top of interrupts.)

```
(1) void interrupt tickISR() {
(2)     uint8_t prio = QK_currPrio_;            /* save the current QK priority */
(3)     QK_currPrio_ = 0xFF;  /* raise the QK priority to the level of this ISR */

(4)     /*Clear any level-sensitive interrupt sources before enabling interrupts*/
(5)     enable();               /* enable the interrupts (QF_ISR_NEST defined) */

(6)     QF_tick();                          /* process all time events (timers) */
        Video_printNumAt(20, 17, VIDEO_FGND_YELLOW, ++l_tickCtr);

(7)     disable();        /* disable interrupts for the exit from the interrupt */
(8)     outportb(0x20, 0x20); /* EOI (End-Of-Interrupt) to the master 8259A PIC */
(9)     QK_currPrio_ = prio;                /* restore the original QK priority */
(10)    QK_schedule();                  /* handle the asynchronous preemptions */
    }
```

Listing 18 Time tick interrupt for the x86 PELICAN example with QK (interrupt nesting allowed).

Listing 18(1) The usual "interrupt" extended keyword must be used to inform the C compiler to generate appropriate ISR entry and exit.

Listing 18(2) The current QK priority is saved in the temporary stack variable to be restored later.

Listing 18(3) The QK priority is raised to the level of this ISR, which must be above any task (above `QF_MAX_ACTIVE+1`).

Listing 18(4) Any level-sensitive interrupt sources not cleared automatically must be cleared explicitly before unlocking interrupts.

Listing 18(5) Interrupts are enabled, consistently with the configuration defined by `QF_ISR_NEST`. At this point interrupt nesting is allowed.

Listing 18(6) The QP-nano ISR service is invoked with interrupts unlocked, which is consistent with the defining the macro `QF_INT_NEST` in the `qpn_port.h` header file.

Listing 18(7) Interrupts must be locked before writing the End-Of-Interrupt (EOI) command to the Interrupt Controller and invoking the QK scheduler.

Listing 18(8) The EOI command is sent to the Interrupt controller.

Listing 18(9) The original QK priority is restored.

Listing 18(10) The QK scheduler is invoked with interrupts locked and also returns with interrupts locked. `QK_schedule()` can unlock interrupts internally.

5.4 Event Queues in QP-nano

Event queues are necessary elements of any system of state machines, because state machines can only process events in uninterruptible Run-To-Completion (RTC) steps, but must accept events at any time. Event queues solve this dilemma by buffering events that cannot be processed immediately while the state machine is in the middle of its RTC step.

In QP-nano each active object has its own event queue. The queue consists of one event located inside the `QActive struct` (from which all active objects derive, see Section ??), plus a ring-buffer of events that is allocated outside of the active object.

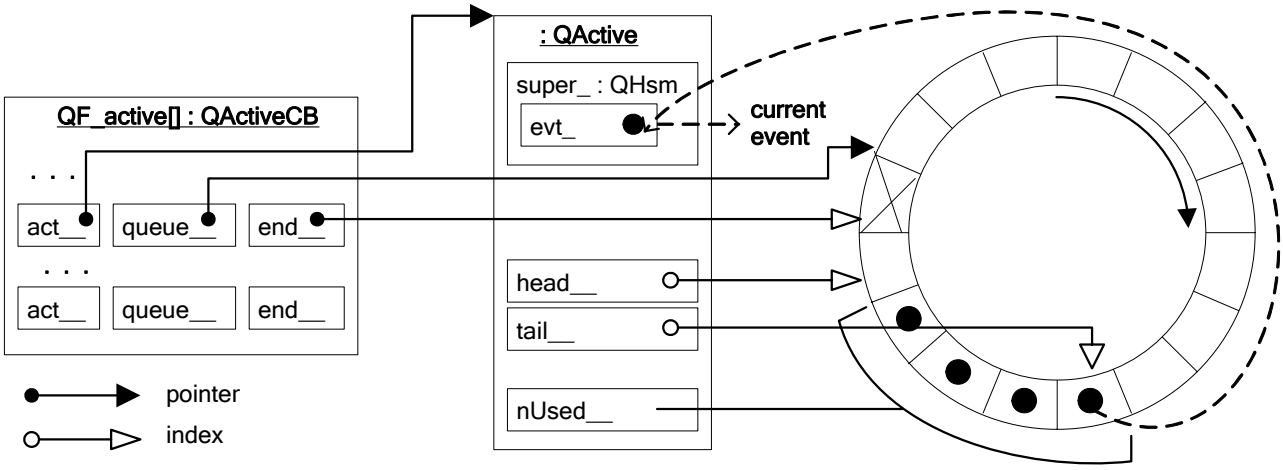

Figure 16 The relationship between the constant QF_active[] array, the QActive struct and the ring buffer of the queue in QP-nano.

Figure 16 shows the data structures that QP-nano uses to manage event queues of active objects. The constant array `QF_active[]` stores the active object "control blocks", which are instances of the `QActiveCB` struct. Each `QActiveCB` entry contains the pointer to the active object (`act__` pointer), the pointer to the ring buffer (`queue__` pointer), and the index of the last element of the ring buffer (the `end__` index). All these elements are configured at compile time for each active object.

The `QActive` struct stores the event queue elements that are changing at runtime. The queue storage consists of the external, user-allocated ring buffer `queue__` plus the current event `evt_` stored inside the state machine (see the `QHsm` struct in the `qph.h` header file). All events dispatched to the state machine must go through the "current event" `evt_` data member, which is indicated as dashed lines in Figure 16. This extra location outside the ring buffer optimizes queue operation by frequently bypassing buffering because in most cases queues alternate between empty and non-empty states with just one event present in the queue at a time. (In situations when the adequate queue depth is only one event, the whole ring buffer can be eliminated entirely.)

The indices `head__`, and `tail__` (and also `end__` from `QActiveCB`) are relative to the `queue__` pointer. These indices manage a ring buffer `queue__` that the clients must pre-allocate as a contiguous array of events of type `QEvent`. Events are always extracted from the buffer at the `tail__` index. New events are inserted at the `head__` index, which corresponds to FIFO queuing. The `tail__` index always increments when the event is extracted, as does the `head__` index when an event is inserted. The `end__` index limits the range of the `head__` and `tail__` indices that must "wrap around" to zero once they reach the `end__` index. The effect is a clockwise crawling of the indices around the buffer, as indicated by the arrow in Figure 16.

5.4.1 Task-Level Posting to the Q-nano Event Queue (QF_post())

```
    #if (Q_PARAM_SIZE != 0)
(1) void QF_post(uint8_t prio, QSignal sig, QParam par) {
    #else
(2) void QF_post(uint8_t prio, QSignal sig) {
    #endif
        QActiveCB const Q_ROM *cb;
        QActive *a;

(3)     QF_INT_LOCK();
        cb = &QF_active[prio];
        a = cb->act__;

        if (a->nUsed__ == (uint8_t)0) {
            ++a->nUsed__;                           /* update number of events */

            Q_SIG(a) = sig;                        /* deliver the event directly */
    #if (Q_PARAM_SIZE != 0)
            Q_PAR(a) = par;
    #endif
            QF_readySet_ |= l_pow2[prio];                  /* set the ready bit */

    #ifdef QK_PREEMPTIVE
(4)         QK_schedule();                  /* check for synchronous preemption */
    #endif
        }
        else {
                /* the queue must be able to accept the event (cannot overflow) */
            Q_ASSERT(a->nUsed__ <= cb->end__);
            ++a->nUsed__;                           /* update number of events */
                                    /* insert event into the ring buffer (FIFO) */
            cb->queue__[a->head__].sig = sig;
    #if (Q_PARAM_SIZE != 0)
            cb->queue__[a->head__].par = par;
    #endif
            ++a->head__;
            if (a->head__ == cb->end__) {
                a->head__ = (uint8_t)0;                       /* wrap the head */
            }
        }
(5)     QF_INT_UNLOCK();
    }
```

Listing 19 Posting to QP-nano event queue at the task level

Listing 19(1-2) The signature of the `QF_post()` function depends whether you've configured events with or without parameter.

Listing 19(3, 5) The task-level posting to the event queue always happens in a critical section.

Listing 19(4) When QP-nano is configured for preemptive scheduling with QK-nano, the QK-nano scheduler must be invoked when a new active object becomes ready. If the priority of the newly readied active object is higher than the current QK priority, the QK-nano scheduler must execute the new active object immediately.

5.4.2 ISR-Level Posting to the Q-nano Event Queue (QF_postISR())

```
    #if (Q_PARAM_SIZE != 0)
(1) void QF_postISR(uint8_t prio, QSignal sig, QParam par)
    #else
(2) void QF_postISR(uint8_t prio, QSignal sig)
    #endif
    {
        QActiveCB const Q_ROM *cb;
        QActive *a;

    #ifdef QF_ISR_NEST
(3)     QF_INT_LOCK();
    #endif
        cb = &QF_active[prio];
        a = cb->act__;
        if (a->nUsed__ == (uint8_t)0) {
            ++a->nUsed__;                           /* update number of events */
            Q_SIG(a) = sig;                       /* deliver the event directly */
    #if (Q_PARAM_SIZE != 0)
            Q_PAR(a) = par;
    #endif
            QF_readySet_ |= l_pow2[prio];                  /* set the ready bit */
        }
        else {
                /* the queue must be able to accept the event (cannot overflow) */
            Q_ASSERT(a->nUsed__ <= cb->end__);
            ++a->nUsed__;                           /* update number of events */
                                  /* insert event into the ring buffer (FIFO) */
            cb->queue__[a->head__].sig = sig;
    #if (Q_PARAM_SIZE != 0)
            cb->queue__[a->head__].par = par;
    #endif
            ++a->head__;
            if (a->head__ == cb->end__) {
                a->head__ = (uint8_t)0;                       /* wrap the head */
            }
        }

    #ifdef QF_ISR_NEST
(4)     QF_INT_UNLOCK();
    #endif
    }
```

Listing 20 Posting to QP-nano event queue at the ISR level

Listing 20(1,2) The The signature of the `QF_postISR()` function depends whether you've configured events with or without parameter.

Listing 20(3,4) Typically the ISR-level posting does NOT use critical section, because the whole interrupt executes already with interrupts locked and nesting of critical sections is not allowed. However, as described in Section 5.3.3, when nesting of interrupts is supported by the dedicated Interrupt Controller hardware, `QF_postISR()` is called with interrupts unlocked. In this special case (configured by the `QF_ISR_NEST` macro), the `QF_postISR()` function uses the critical section internally.

NOTE: The `QF_postISR()` function never calls the QK-nano scheduler, because a task can never preempt an ISR.

5.5 Scheduling in QP-nano

You can use QP-nano in two configurations: (1) with non-preemptive cooperative scheduler and (2) with fully preemptive real-time kernel (QK-nano). This section discusses the scheduling algorithms used in these two configurations.

5.5.1 Non-Preemptive, Priority-Based Scheduler

The software structure of the non-preemptive configuration closely resembles the simple foreground/background architecture, where the ISRs handle asynchronous events in a timely fashion (foreground), and an infinite loop uses all remaining CPU cycles to perform less time-critical actions (background). Most high-volume microcontroller-based applications, such as home appliances, electronic toys, vending machines, thermostats, and countless others, are typically organized as foreground/background systems.

In a traditional foreground/background system, the background loop calls directly or indirectly all modules of the application one after the other in a circular fashion. Any asynchronous event recognized by an ISR must wait, in the worst case, for the full pass through the background loop until the right piece of code comes around to process the event. This lag time, known as the *task-level response*, varies as not all passes through the background loop take the same amount of time. Also any modifications to the background loop, or to the ISR level, change the timing of the software execution.

A QP-nano-based application can easily achieve a much better task-level response and more constant execution timing than a typical foreground/background system. In the QP-nano application you can easily build a very simple *non-preemptive, cooperative* **scheduler** out of the QP-nano event queues (Section 5.4) and the QF ready-set variable (shown in Figure 17). This cooperative scheduler is engaged after each RTC step to pick the highest-priority active object ready to run for the next RTC step. Thus, the worst-case task-level response of the QP-nano application is no longer the full pass through all application modules in the background loop, but rather the longest RTC step of individual event processing, which is typically much shorter.

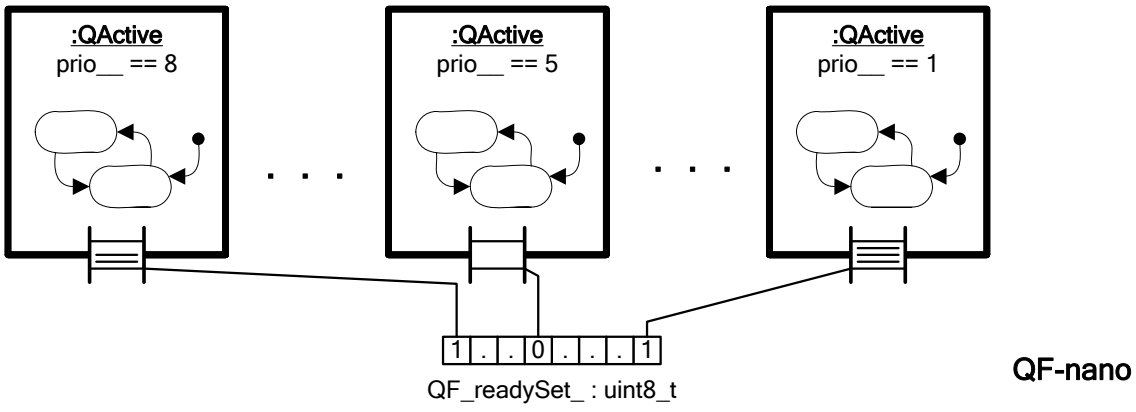

Figure 17 QF_readySet_ representing the empty/non-empty state of all event queues

Figure 17 shows the relationship between the QF event queues and QF_readySet_ data element. The QF ready-set `QF_readySet_` is a bitmask of type `uint8_t`, which is capable of representing up to 8 active object priorities numbered 1 through 8. As shown in Figure 17, each bit in the `QF_readySet_` represents one QF active object. The bit number 'n' in the `QF_readySet_` is 1 if the

event queue of the active object of priority 'n' is not empty. Conversely, bit number 'm' in QF_readySet_ is 0 if the event queue of the active object of priority 'm' is empty, or the priority level 'm' is not used. Both QP-nano event queues and QF_readySet_ are *always* accessed in a critical section to prevent data corruption.

```
    #ifndef QK_PREEMPTIVE

(1) void QF_run(void) {
        static uint8_t const Q_ROM log2Lkup[] = {
            0, 1, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4
        };
        static uint8_t const Q_ROM invPow2[] = {
            0xFF, 0xFE, 0xFD, 0xFB, 0xF7, 0xEF, 0xDF, 0xBF, 0x7F
        };

(2)     QF_start();                                              /* enable ISRs */

(3)     QF_INT_LOCK();
(4)     for (;;) {                                        /* the background loop */
(5)         if (QF_readySet_ != (uint8_t)0) {
                QActiveCB const Q_ROM *cb;
                uint8_t p;
                QActive *a;

    #if (QF_MAX_ACTIVE > 4)
                if ((QF_readySet_ & 0xF0) != 0) {         /* upper nibble used? */
(6)                 p = (uint8_t)(log2Lkup[QF_readySet_ >> 4] + 4);
                }
                else                     /* upper nibble of QF_readySet_ is zero */
    #endif
                {
(7)                 p = log2Lkup[QF_readySet_];
                }
(8)             QF_INT_UNLOCK();

                cb = &QF_active[p];
                a = cb->act__;

    #if (!defined(QF_FSM_ACTIVE) && !defined(Q_NHSM))
(9)             QHsm_dispatch((QHsm *)a);          /* dispatch to state machine */
    #else
(10)            QFsm_dispatch((QFsm *)a);          /* dispatch to state machine */
    #endif

(11)            QF_INT_LOCK();
(12)            if ((--a->nUsed__) == (uint8_t)0) {    /* queue becoming empty? */
(13)                QF_readySet_ &= invPow2[p];          /* clear the ready bit */
                }
                else {
                    Q_SIG(a) = cb->queue__[a->tail__].sig;
    #if (Q_PARAM_SIZE != 0)
                    Q_PAR(a) = cb->queue__[a->tail__].par;
    #endif
                    ++a->tail__;
                    if (a->tail__ == cb->end__) {              /* wrap around? */
                        a->tail__ = (uint8_t)0;               /* wrap the tail */
                    }
                }
            }
            else {
(14)            QF_onIdle();
(15)            QF_INT_LOCK();
```

```
            }
        }
    }

    #endif                                          /* #ifndef QK_PREEMPTIVE */
```

Listing 21 The non-preemptive QF-nano scheduler (qkn.h not included in qpn_port.h).

Listing 21(1) The QF-nano cooperative scheduler is entirely implemented in the `QF_run()` function.

Listing 21(2) `QF_run()` invokes the callback `QF_start()` to enable and release the interrupts.

Listing 21(3) The interrupts are locked before accessing the `QF_readySet_` variable.

Listing 21(4) The endless background loop is entered.

Listing 21(5) The system has events to process only if `QF_readySet_` is not empty

Listing 21(6,7) The priority of the highest-priority active object with events to process is quickly determined by a binary logarithm (log-base-2) lookup. The lookup table has actually only 16 entries, so it can be applied only to one nibble of the `QF_readySet_` byte (2^4 bit combinations). If the QF_MAX_ACTIVE is greater than 4, the scheduler first examines the most-significant nibble. For QF_MAX_ACTIVE less or equal 4 the testing of the upper nibble is not necessary.

Listing 21(8) The interrupts can be unlocked after resolving the active object priority

Listing 21(9,10) The current event is dispatched to the state machine of the active object. Depending on the QP-nano configuration, the state machine is either the fully Hierarchical State Machine (HSM), or the simpler "flat" FSM. The event processing in the state machine represents the Run-To-Completion (RTC) step in this scheduler.

Listing 21(11) After the RTC step completes, the interrupts are locked again

Listing 21(12) The next event is de-queued from the event queue

Listing 21(13) If the queue becomes empty, the corresponding bit in the `QF_readySet_` is cleared.

Listing 21(14) If no events are available, the scheduler invokes the `QF_onIdle()` callback.

NOTE: The `QF_onIdle()` callback is invoked with interrupts **locked**. This must be so, because the idle condition (no events to process) can be changed by any interrupt posting an event to a queue. If interrupts were unlocked before calling `QF_onIdle()`, and interrupt could post an event to a queue, but the `QF_onIdle()` would put the CPU to low-power mode, thus preventing the processing of the event until the next interrupt.

`QF_onIdle()` callback MUST at least unlock interrupts. A failure to unlock interrupts in `QF_onIdle()` will leave the interrupts locked all the time and would prevent the application from running.

Listing 21(15) After `QF_onIdle()` returns, the interrupts are locked again to perform another test for events to process. If no interrupts occur after they are unlocked inside `QF_onIdle()`, the test for new events will still indicate the idle condition and `QF_onIdle()` will be called again. This is the **idle loop** of this scheduler.

The scheduler is cooperative, which means that all active objects "cooperate" to share a single CPU and implicitly yield to each other after every RTC step. The scheduler is non-preemptive, which means that every active object must completely process an event before any other active object can start processing another event.

The main advantage of the non-preemptive active object execution is that active objects can share data and other resources, although such sharing always increases the coupling among active objects.

The most important drawback of the non-preemptive execution is responsiveness. A higher-priority active object that has received an event from an ISR or the currently running lower-priority active object must always wait for the completion of the current RTC. Because there is no boundary on the length of an RTC step, the task-level response is non-deterministic and often unacceptable for hard real-time deadlines. This forces designers to put more time-critical code into the ISRs, which tend to grow larger then they should. Any change in the active objects or ISRs can change the task-level response.

5.6 Time Events (Timers) in QP-nano

QP-nano manages time through time events[3] (timers). Each active object in QP-nano has one such private time event. The basic usage model of these time events is as follows. When the active object needs to arrange for a timeout, it arms one its time event to post the Q_TIMEOUT_SIG event to itself in the future. The active object interface provides a method for that purpose: `QActive_arm()`, which arms the timer for one-shot timeout in the specified number of clock ticks. Each active object has a different, private time event, so a QP application can make multiple parallel requests. When the `QF_tick()` detects that the appropriate moment has arrived, it posts the requested `Q_TIMEOUT_SIG` event directly into the recipient's event queue using the `QF_postISR()` function. The recipient then processes the time event just like any other event. If the recipient wants to receive timeout events periodically, it must arm the time event again, since it is automatically disarmed after each posting (one-shot timeout).

To arm a time event, an active object must provide its "`me`" pointer and the number of clock ticks to elapse before delivery. The timer is represented as a simple down-counter with the dynamic range adjustable to 0 (no timer), 8, 16, and 32-bits via the configuration macro `QF_TIMEEVT_CTR_SIZE`.

You can explicitly disarm any time event (periodic or one-shot) at any time by means of the `QActive_disarm()` function. At any moment a timer can be re-armed by calling `QActive_arm()`, which simply replaces the value of the timer downcounter.

[3] Time Event is a UML term.

6 QP-nano Compliance with Standards

6.1 MISRA™ Compliance

QP-nano source code is 98% compliant with the Motor Industry Software Reliability Association (MISRA) *Guidelines For The Use of The C Language in Vehicle Based Software* [MISRA 98]. These standards were created by MISRA™ to improve the reliability and predictability of C programs in critical automotive systems. Members of the MISRA™ consortium include Delco Electronics, Ford Motor Company, Jaguar Cars Ltd., Lotus Engineering, Lucas Electronics, Rolls-Royce, Rover Group Ltd., and other commercial and academic institutions dedicated to improving safety and reliability in automotive electronics. Full details of this standard can be obtained directly from the MISRA™ web site, http://www.misra.org.uk/.

The Application Note "*QP-nano MISRA Compliance Matrix*" shows how QP-nano complies with all 127 C Coding Rules. The document is available online from http://www.quantum-leaps.com/-doc/AN_QP-nano_MISRA.pdf.

6.2 PC-Lint™ Compliance

QP-nano source code is "lint-free". The compliance was checked against PC-lint™/FlexLint™, which is perhaps the most advanced and widely used static analysis tool from Gimpel Software (`http://www.gimpel.com`). The QEP/C distribution includes the `<qpc_3>\qep\lint\` subdirectory, which contains the PC-lint™ configuration files used for static analysis of the QEP/C source code.

The Application Note "*QP-nano PC-Lint Compliance Report*" reports how QP-nano complies with PC-lint™ and discusses the configuration used. The document is available online from http://www.quantum-leaps.com/doc/AN_QP-nano_PC-Lint.pdf

6.3 Coding Standard Compliance

QP-nano source code is compliant with the documented "*Quantum Leaps C/C++ Coding Standard*" available online from http://www.quantum-leaps.com/doc/AN_QL_Coding_Standard.pdf.

7 QP-nano Reference Manual

The QP-nano Reference Manual is available in the directory <qpn>\doxygen\html. The Reference Manual has been generated by Doxygen (http://www.doxygen.org), which is an open source documentation generation system for C, C++, Java, and other languages.

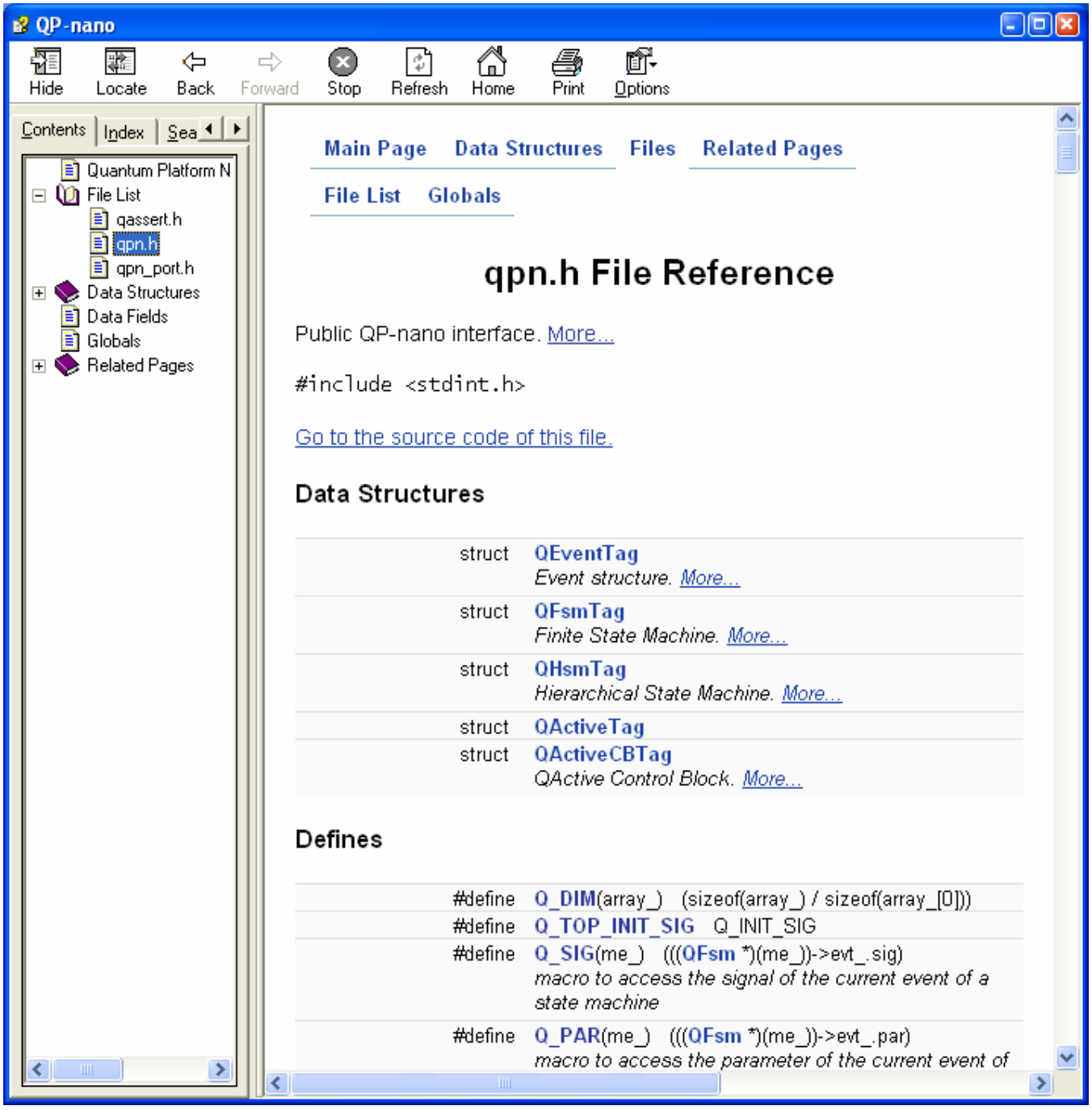

Figure 18 Screen shot of the QP-nano Reference Manual.

8 Related Documents and References

Document	Location
"Practical Statecharts in C/C++", Miro Samek, CMP Books, 2002	Available from most online book retailers, such as amazon.com. See also: http://www.quantum-leaps.com/writings/book.htm
"Quantum Platform Overview", Quantum Leaps 2005	http://www.quantum-leaps.com/doc/QP_Overview.pdf
"Brief Introduction to UML State Machines", Quantum Leaps, 2005	http://www.quantum-leaps.com/devzone/Recipe_IntroHSM.pdf
"RECIPE: Designing a Hierarchical State Machine", Quantum Leaps, 2005	http://www.quantum-leaps.com/devzone/Recipe_DesigningHSM.pdf
"QP Programmer's Manual", Quantum Leaps, LLC, 2005	http://www.quantum-leaps.com/doc/%1fQP_Manual.pdf
"Quantum Platform Overview", Quantum Leaps, LLC, 2005	http://www.quantum-leaps.com/doc/QP_Overview.pdf
"Quick UML Reference", Quantum Leaps 2005	http://www.quantum-leaps.com/resources/goodies.htm#UML
"RECIPE: Simple Encapsulation and Inheritance in C", Quantum Leaps 2005	http://www.quantum-leaps.com/devzone/Recipe_SimpleOOC.pdf
[C99] ISO/IEC 9899 C Approved Standards	http://www.open-std.org/jtc1/sc22/open/n2794
[MISRA 98] Motor Industry Software Reliability Association (MISRA), MISRA Limited, MISRA-C:1998 Guidelines for the Use of the C Language in Vehicle Based Software, April 1998, ISBN 0-9524156-9-0	See at: http://www.misra.org.uk
[OMG 03] Unified Modeling Language: Superstructure,	document ptc/03-08-02.pdf, See also http://www.omg.org
[Samek 06a] Samek, Miro, "UML Statecharts at $10.99", Dr. Dobb's Journal, May 2006	http://www.ddj.com/dept/embedded/188101799
[Samek+ 06b] Samek, Miro and Robert Ward, "Build a Super-Simple Tasker", Embedded Systems Design, July 2006	http://www.embedded.com/mag.htm