Quantum™ LeaPs
innovating embedded systems

Application Note
PEdestrian LIght CONtrolled (PELICAN) Crossing Example

Document Revision C
August 2008

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

Table of Contents

1 Introduction

This Application Note describes an example application for the QP-nano™ event-driven platform. QP-nano is a generic, portable, **ultra-lightweight**, event-driven infrastructure designed specifically for *low-end* 8- and 16-bit MCUs, such as 8051, PICmicro, AVR, 68H(S)08, MSP430, M16C/R8C, Cypress PSoC, and many others alike. QP-nano enables building well-structured embedded applications as a set of concurrently executing hierarchical state machines (UML statecharts) directly in C or C++ **without big tools**. QP-nano is described in Chapter 12 of the book "*Practical UML Statecharts in C/C++, Second Edition*" [PSiCC2] (Newnes, 2008).

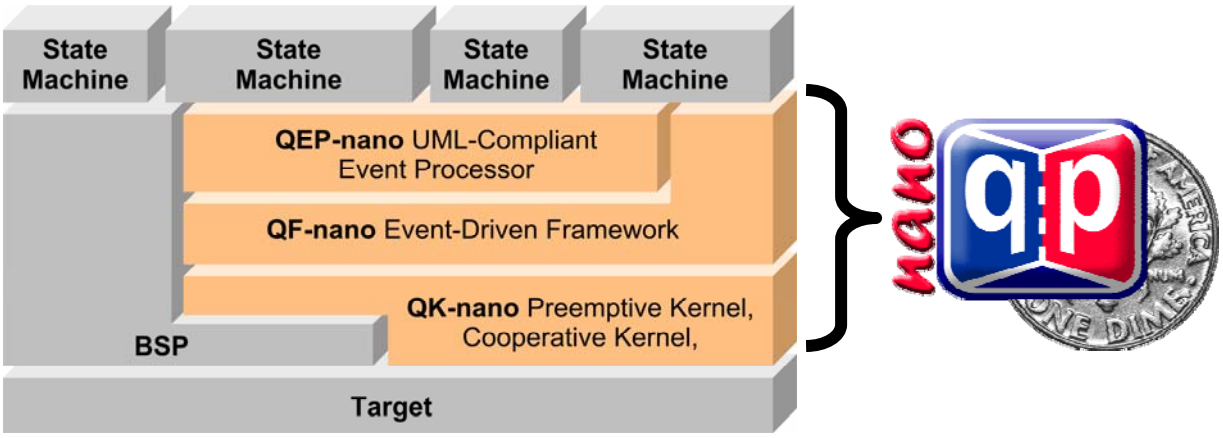


Figure 1 QP-nano components (in grey) and their relationship with the target hardware, board support package (BSP), and the application.

As shown in Figure 1, QP-nano consists of a hierarchical event processor called QEP-nano, a minimal real-time framework called QF-nano, and a choice between a preemptive run-to-completion kernel called QK-nano, or a cooperative "vanilla" kernel.

1.1 Key Features of QP-nano

The key QP-nano features are:

- Full support for hierarchical state nesting, including guaranteed entry/exit action execution on arbitrary state transition topology with up to 4 levels of state nesting
- Support for up to 8 concurrently executing active objects with deterministic, thread-safe event queues

- Support for events with one scalar parameter, configurable to 0 (no parameter), 1, 2, or 4 bytes
- Direct event delivery mechanism with first-in-first-out (FIFO) queuing policy
- One single-shot time event (timer) per active object with configurable dynamic range of 0 (no time events), 1, 2, or 4 bytes
- Built-in cooperative "vanilla" kernel (see Chapter 6 in [PSiCC2])
- Built-in preemptive RTC kernel called QK-nano (see Chapter 6 in [PSiCC2])
- Low-power architecture with idle callback function for easy implementation of power-saving modes
- Provisions in the code to handle non-standard extensions in the C compilers for popular low-end CPU architectures (e.g., allocation of constant objects in the code space, reentrant functions, etc.).
- Assertion-based error handling policy.

1.2 Tiny Size

As shown in Figure 2, a minimal QP-nano system requires some 100 bytes of RAM and 2KB of ROM. Note that these requirements pertain to the complete application, including the C-stack, as opposed to just the QP-nano footprint. Figure 2 shows for comparison minimal system sizes required for the full-version QP as well as popular RTOS/OS.

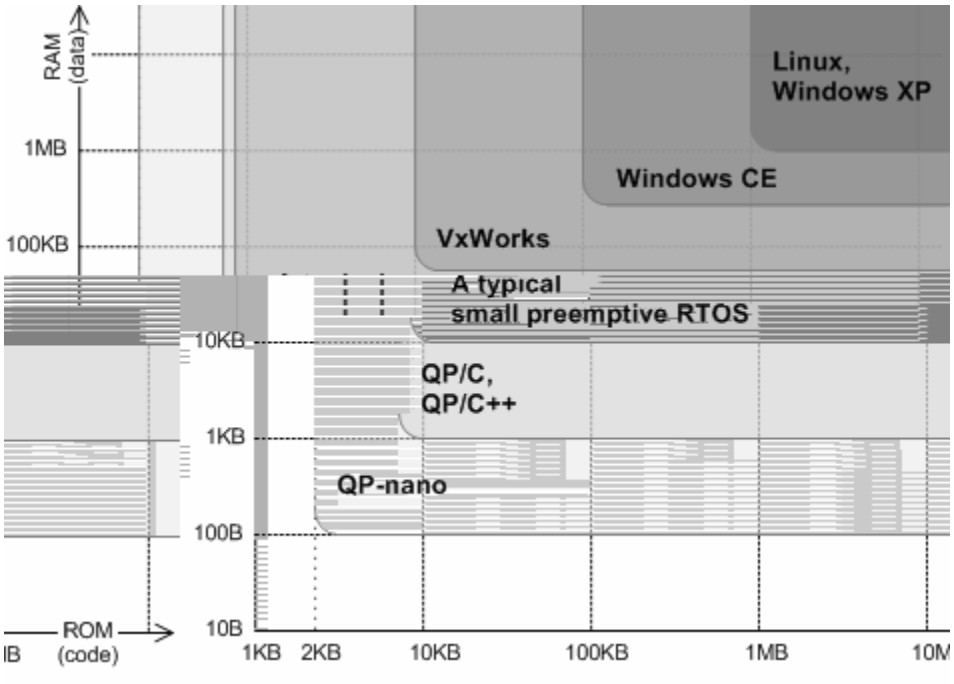


Figure 2 RAM/ROM footprints of QP/C, QP/C++, QP-nano, and other RTOS/OS. The chart shows approximate total system size, as opposed to just the RTOS/OS footprints. Please note logarithmic axes.

By far, the biggest challenge in QP-nano design is the extremely tight RAM budget, which is assumed to be only around 100 bytes, including the C stack. Obviously, with RAM in such short supply, QP-nano design carefully accounts for every single byte of RAM. This is in contrast to the full-version QP, where saving every last byte of RAM was not the highest priority if this would reduce programming convenience, flexibility, or performance.

Perhaps the most important implication of the severely limited RAM is that QP-nano does not support events with arbitrary sized parameters. Instead, QP-nano allows only fixed-size events with one scalar parameter, configurable to 1, 2, or 4 bytes (or 0 bytes, which means no event parameter). This has far reaching simplifying consequences. First, event queues in QP-nano hold entire events, not just pointers to events as in the full-version QP. Small, fixed-size events are simply copied by value into and out of event queues in inherently thread-safe manner. Second, the copy-by-value policy eliminates the need for event pools, which would not fit into the available RAM anyway. Finally, reference counting of events is unnecessary in this design.

NOTE: A single scalar event parameter means that QP-nano always associates the configured number of bytes with every event, but it does not mean that you can have only one event parameter. In fact, each event can have as many event parameters as you can squeeze into the available bits.

1.3 Source Code

The Quantum Leaps website at www.quantum-leaps.com/downloads/ contains the complete source code for all QP-nano components. The source code is very clean and consistent. The code has been written in strict adherence to the coding standard documented at www.quantum-leaps.com/doc/-AN_QL_Coding_Standard.pdf.

All QP-nano source code is "lint-free". The compliance was checked with PC-lint/FlexLint static analysis tool from Gimpel Software (www.gimpel.com). The QP distribution includes the `<qp>\ports\lint\` subdirectory, which contains the batch script `make.bat` for compiling all the QP components with PC-lint.

The QP-nano source code is also 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. Full details of this standard can be obtained directly from the MISRA web site at www.misra.org.uk. The PC-lint configuration used to analyze QP code includes the MISRA rule checker.

Finally and most importantly, simply giving you the source code is not enough. To gain real confidence in event-driven programming, you need to understand how a real-time framework is ultimately implemented and how the different pieces fit together. The [PSiCC2] book, and numerous Application Notes and QDKs, provide this kind of information.

1.4 Portability

All QP-nano source code is written in portable ANSI-C, with all processor-specific, compiler-specific, or operating system-specific code abstracted into a clearly defined platform abstraction layer (PAL).

QP-nano runs on "bare-metal" target CPU completely replacing the traditional RTOS at a small fraction of the RAM and ROM footprint required by even the smallest conventional RTOSes. As shown in Figure 1, the QP-nano event-driven platform includes the simple non-preemptive "vanilla" sched-

uler as well as the fully preemptive QK-nano kernel. To date, QP-nano has been ported to over 10 different CPU architectures, ranging from 8-bit (e.g., 8051, PIC, AVR, 68H(S)08, Cypress PSoC), through 16-bit (e.g., MSP430, M16C, x86-real mode), to 32-bit architectures (e.g., traditional ARM, ARM Cortex-M3, ColdFire).

1.5 Support for Modern Hierarchical State Machines

As shown in Figure 1, the QP-nano includes the UML-compliant QEP-nano hierarchical event processor that executes UML state machines according to the UML semantics, while the QF-nano framework provides the infrastructure of executing such state machines concurrently.

1.6 Low-Power Architecture

Most modern embedded microcontrollers (MCUs) provide an assortment of low-power sleep modes designed to conserve power by gating the clock to the CPU and various peripherals. The sleep-modes are entered under the software control and are exited upon an external interrupt.

The event-driven paradigm is particularly suitable for taking advantage of these power-savings features, because every event-driven system can easily detect situation when the system has no more events to process, which is called the idle condition (Chapter 6 in [PSiCC2]). In both QP-nano configurations, either with the cooperative “vanilla” kernel, or with the QK-nano preemptive kernel, the QF-nano framework provides callback functions for handling the idle condition. These callbacks are carefully designed to place the MCU into a low-power sleep mode safely and without creating race conditions with active interrupts.

1.7 Assertion-Based Error Handling

All QP-nano components consistently use the Design by Contract (DbC) philosophy described in Chapter 6 of [PSiCC2]. QP-nano constantly monitors the application by means of assertions built into the framework. Among others, QF uses assertions to enforce the event delivery guarantee, which immensely simplifies event-driven application design.

2 The PELICAN Crossing Example

Many QP-nano™ Development Kits (QDKs-nano) use the PEdestrian LIght CONtrolled (PELICAN) crossing as an example application. The PELICAN crossing example demonstrates a non-trivial hierarchical state machine, event exchanges among active objects and ISRs, time events, and even the QK-nano preemptive kernel. QP-nano makes it possible to squeeze this application inside a system as small as the MSP430-F2013 ultra-low power MCU with only 128 bytes of RAM and 2KB of flash ROM.

2.1 Step1: Requirements

First, your always need to understand what your application is supposed to accomplish. In the case of a simple application, the requirements are conveyed through the problem specification, which for the PELICAN crossing is as follows.

The PELICAN crossing (see Figure 3) starts with cars enabled (green light for cars) and pedestrians disabled ("Don't Walk" signal for pedestrians). To activate the traffic light change, a pedestrian must push the button at the crossing, which generates the `PEDS_WAITING` event. In response, the cars get the yellow light, which after a few seconds changes to red light. Next, pedestrians get the "Walk" signal, which shortly thereafter changes to the flashing "Don't Walk" signal. When the "Don't Walk" signal stops flashing, cars get the green light again. After this cycle, the traffic lights don't respond to the `PEDS_WAITING` button press immediately, although the button "remembers" that it has been pressed. The traffic light controller always gives the cars a minimum of several seconds of green light before repeating the traffic light change cycle. One additional feature is that at any time an operator can take the PELICAN crossing offline (by providing the OFF event). In the "offline" mode, the cars get a flashing red light and pedestrians flashing "Don't Walk" signal. At any time the operator can turn the crossing back online (by providing the ON event).

Figure 3 PEdestrian LIght CONtrolled (PELICAN) Crossing.

2.2 Step 2: Sequence Diagrams

A good starting point in designing any event-driven system is to draw sequence diagrams for the main scenarios (main use cases) identified from the problem specification. To draw such diagrams, you need to break up your problem into active objects with the main goal of minimizing the coupling among active objects. You seek a partitioning of the problem that avoids resource sharing and requires minimal communication in terms of number and size of exchanged events.

The sequence diagram in Figure 5 shows two most representative scenarios of the PELICAN crossing operation. In panel (a), you see the scenario in which the initial minimal green light for cars elapses without the pedestrian generating the `PEDS_WAITING` event. In panel (b) you see the situation where the pedestrian generates the `PEDS_WAITING` event during the minimal green light for cars. The explanation section immediately following Figure 5 highlights the most interesting points.

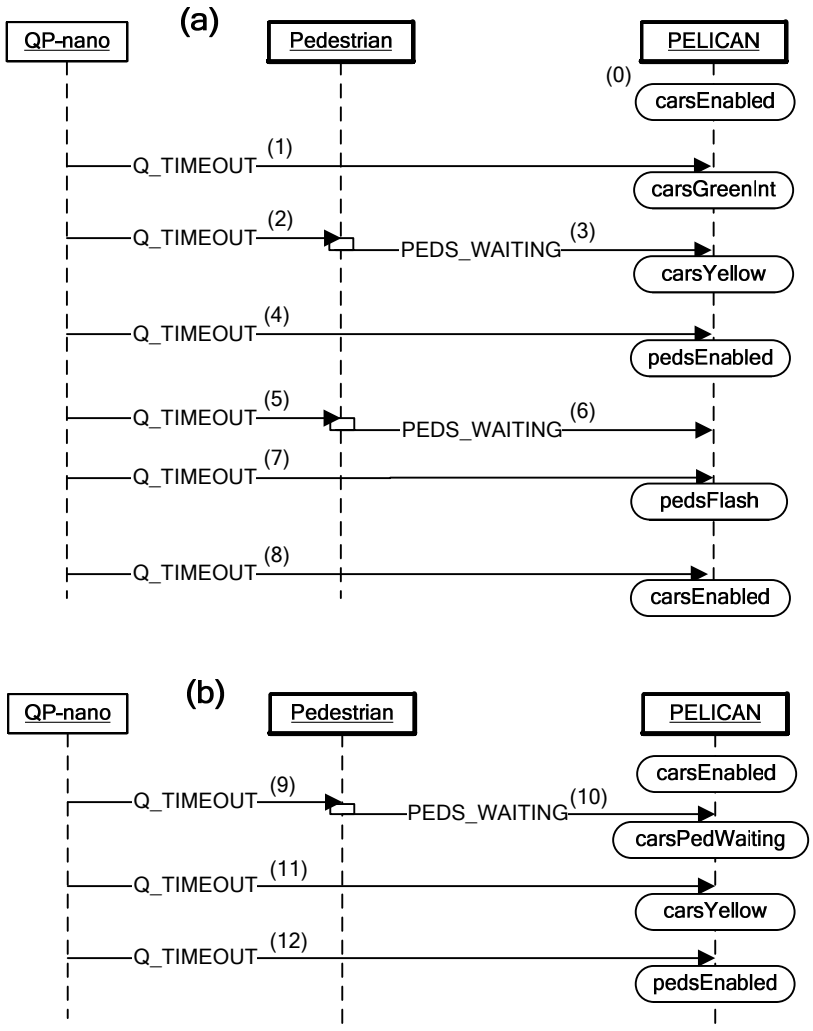


Figure 4 Sequence diagrams of the PELICAN application. PEDS_WAITING event after the minimum green light for cars (a), and PEDS_WAITING event during the minimum green light for cars (b).

(0) The QP-nano infrastructure initializes all active objects (state machines) in the system. The PELICAN state machine starts in the "carsEnabled" state, which arms a QP-nano timer to expire

after the minimum green light for cars. The Pedestrian active object arms its QP-nano timer to trigger the PEDS_WAITING event.

(1) The QP-nano timer for the PELICAN active object expires, which causes a transition to "carsGreenInt" state. This state "remembers" that the minimum green time for cars has elapsed, so the green light can be now interrupted at any time.

(2) The QP-nano timer for the PELICAN active object expires, which causes a transition to "carsGreenInt" state. This state "remembers" that the minimum green time for cars has elapsed, so the green light can be now interrupted at any time.

(3) The Pedestrian active object posts the PEDS_WAITING event directly to the PELICAN active object. This event causes immediate transition to "carsYellow", in which cars get the yellow light. The entry to the "carsYellow" light arms the QP-nano timer to trigger light change to red.

(4) After the timer expires the PELICAN active object transitions to "pedsEnabled". This transition causes displaying red light for cars and "WALK" signal to pedestrians.

(5) The QP-nano timer expires for the Pedestrian active object.

(6) As usual, the Pedestrian active object generates the PEDS_WAITING event, but this time the PELICAN active object ignores the event, as it already is allowing pedestrians to the crossing.

(7) The QP-nano timer expires to trigger flashing the "DON'T WALK" signal for pedestrians.

(8) Finally, the QP-nano timer expires and triggers the transition back to "carsEnabled" at which time the cycle repeats.

The sequence diagram in Figure 5(b) shows the situation where the pedestrian generates the PEDS_WAITING event during the minimal green light for cars.

(9) The QP-nano timer for the Pedestrian active object expires during the minimal green light for cars.

(10) As usual, the Pedestrian active object generates the PEDS_WAITING event. This time the PELICAN active object reacts by transitioning to the "carsPedsWaiting" state to "remember" that a pedestrian is waiting. The PELICAN crossing keeps showing the green light for cars, as the minimal interval has not expired yet.

(11) Eventually, the QP-nano timer for the PELICAN active object expires, which triggers transition to "carsYellow".

(12) From that point on the scenario is identical as panel (a).

2.3 Step 3: Signals, Events, and Active Objects

Sequence diagrams, like Figure 5, help you discover events exchanged among active objects. The choice of signals and event parameters is perhaps the most important design decision in any event-driven system. The events affect the other main application components: events and state machines of the active objects.

In QP-nano, signals are typically enumerated constants. Listing 1 shows signals and active objects in the PELCIAN application. The PELICAN sample code for the DOS version is located in the <qpn>\examples\80x86\tcpp101\pelican\ directory, where <qpn> stands for the installation directory you chose to install QP-nano.

NOTE: This section describes the platform-independent code of the PELICAN application. This code is actually *identical* in all PELICAN versions for different CPUs and compilers.

```
    #ifndef pelican_h
    #define pelican_h

(1) enum PelicanSignals {
(2)     PEDS_WAITING_SIG = Q_USER_SIG,
        OFF_SIG,
        ON_SIG
    };

    /* active objects ..........................................................*/
(3) extern struct PelicanTag AO_Pelican;
(4) extern struct PedTag     AO_Ped;

(5) void Pelican_ctor(void);
(6) void Ped_ctor(void);

    #endif                                                      /* pelican_h */
```

Listing 1 Signals and active objects in the PELCAN application (file `pelican.h`)

(1) All signals are defined in one enumeration, which automatically guarantees the uniqueness of signals.

(2) Note that the user signals must start with the offset `Q_USER_SIG` to avoid overlapping the reserved QEP-nano signals.

(3-4) All active object instances in the system are declared as extern variables. These declarations are necessary for the initialization of the `QF_active[]` array (see upcoming section about the `main()` function and QP-nano initialization).

NOTE: The active object structures (e.g., `struct PelicanTag`) do not need to be defined globally in the application header file. Initialization of the QP-nano (discussed later) needs only pointers to the active objects, which the compiler can resolve without knowing the full definition of the active object structure.

You should avoid declaring active object structures globally. Instead, you can declare the active object structures in the file scope of the specific active object module (e.g., `struct PelicanTag` is declared in the `pelican.c` file scope). That way, you can be sure that each active object remains fully encapsulated.

(5-6) Every active object in the system must provide a "constructor" function, which initializes the active object instance. These constructors don't take the "`me`" pointers, because they have access to the global active object instances (see (3-4)). However, the constructors can take some other initialization parameters. The constructors are called right at the beginning of `main()`.

2.4 Step 4: Designing State Machines of Active Objects

Sequence diagrams, like Figure 5, help you discover events exchanged among active objects. The choice of signals and event parameters is perhaps the most important design decision in any event-driven system. The events affect the other main application components: events and state machines of the active objects.

2.4.1 PELICAN state machine

Figure 5 shows the complete PELICAN crossing state machine. The explanation section following the diagram describes how it works.

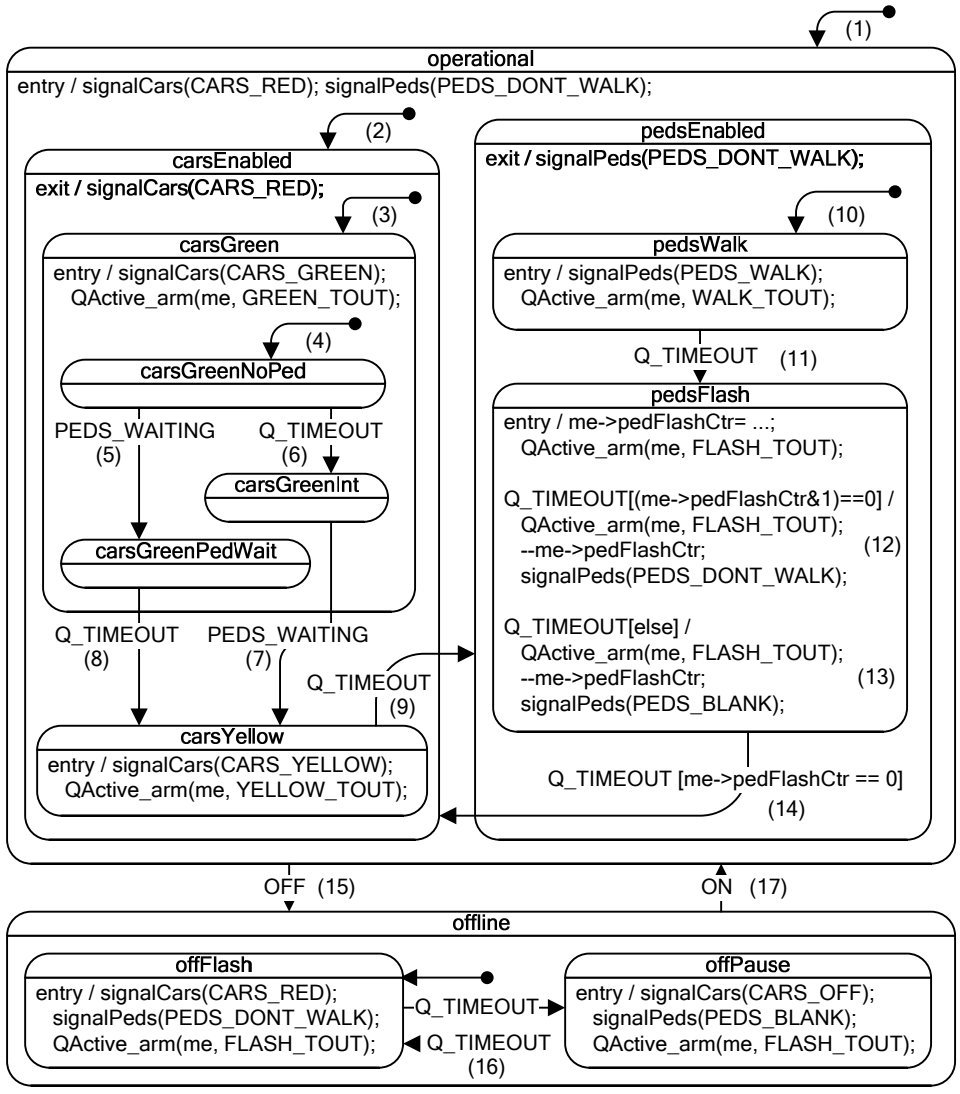


Figure 5 The PELICAN state machine.

(1) Upon the initial transition, the PELICAN state machine enters the "operational" state and displays the red light for cars and "Don't Walk" signal for pedestrians.

(2) The "operational" state has a nested initial transition to the "carsEnabled" substate. Per the UML semantics, this transition must be taken after entering the superstate.

(3) The "carsEnabled" state has a nested initial transition to the "carsGreen" substate. Per the UML semantics, this transition must be taken after entering the sperstate. Entry to "carsGreen" changes signals green light for cars and arms the time event to expire in the GREEN_TOUT clock ticks. The `GREEN_TOUT` timeout represents the minimum duration of green light for cars.

(4) The "carsGreen" state has a nested initial transition to the "carsGreenNoPed" substate. Per the UML semantics, this transition must be taken after entering the superstate. The "carsGreen-NoPed" state is a leaf state, meaning that it has no substates or initial transitions. The state machine stops and waits in this state.

(5) When the `PEDS_WAITING` event arrives in the "carsGreenNoPed" state, the state machine transitions to another leaf state "carsGreenPedWait". Please note that the state machine still remains in the "carsGreen" superstate, because the minimum green light period for cars hasn't expired yet. However, by transitioning to the "carsGreenPedWait" substate, the state machine remembers that the pedestrian is waiting.

(6) However, when the `Q_TIMEOUT` event arrives while the state machine is still in the "carsGreen-NoPed" state, the state machine transitions to the "carsGreenInt" (interruptible green light for cars) state.

(7) The "carsGreenInt" state handles the `PEDS_WAITING` event by immediately transitioning to the "carsYellow" state, because the minimum green light for cars has elapsed.

(8) The "carsGreenPedWait" state, on the other hand, handles only the `Q_TIMEOUT` event, because the pedestrian is already waiting for the expiration of the minimum green light for cars.

(9) The "carsYellow" state displays the yellow light for cars and arms the timer for the duration of the yellow light. The `Q_TIMEOUT` event causes the transition to the "pedsEnabled" state. The transition causes exit from the "carsEnabled" superstate, which displays the red light for cars.

The pair of states "carsEnabled" and "pedsEnabled" realizes the main function of the PELICAN crossing, which is to alternate between enabling cars and enabling pedestrians. The exit action from "carsEnabled" disables cars (by showing red light for cars), while the exit action from "pedsEnabled" disables pedestrians (by showing them the "Don't Walk" signal). The UML semantics of state transitions guarantees that these exit actions will be executed whichever way the states happen to be exited, so you can be sure that the pedestrians will always get the "Don't Walk" signal outside the "pedsEnabled" state and cars will get the red light outside the "carsEnabled" state.

NOTE: Exit actions in the states "carsEnabled" and "pedsEnabled" guarantee mutually exclusive access to the crossing, which is the main safety concern in this application.

(10) The "pedsEnabled" state has a nested initial transition to the "pedsWalk" substate. Per the UML semantics, this transition must be taken after entering the superstate. The entry action to "pedsWalk" shows the "Walk" signal to pedestrians and arms the timer for the duration of this signal.

(11) The `Q_TIMEOUT` event triggers the transition to the "pedsFlash" state, in which the "Don't Walk" signal flashes on and off. You can use the internal variable of the PELICAN state machine `me->pedFlashCtr` to count the number of flashes.

(12-13) The `Q_TIMEOUT` event triggers two internal transitions with complementary guards. When the `me->pedFlashCtr` counter is even, the "Don't Walk" signal is turned on. When it's odd, the "Don't Walk" signal is turned off. Either way the counter is always decremented.

(14) Finally, when the `me->pedFlashCtr` counter reaches zero, the `Q_TIMEOUT` event triggers the transition to the "carsEnabled" state. The transition causes execution of the exit action from the

"pedsEnabled" state, which displays "Don't Walk" signal for pedestrians. The lifecycle of the PELICAN crossing then repeats.

At this point, the main functionality of the PELICAN crossing is done. However, you still need to add the "offline" mode of operation, which is actually quite easy because of the state hierarchy.

(15) The OFF event in the "operational" superstate triggers the transition to the "offline" state. The state hierarchy ensures that the transition OFF is inherited by all direct or transitive substates of the "operational" superstate, so regardless in which substate the state machine happens to be, the OFF event always triggers transition to "offline". Please also note that the semantics of exit actions still applies, so the PELICAN crossing will be left in a consistent safe state (both cars and pedestrians disabled) upon the exit from the "operational" state.

(16) The `Q_TIMEOUT` events in the substates of the "offline" state cause flashing of the signals for cars and pedestrians, as described in the problem specification.

(17) The ON event can interrupt the "offline" mode at any time by triggering the transition to the "operational" state.

The actual implementation of the PELICAN state machine in QP-nano is very straightforward and follows exactly the same simple rules as described for the Ship state machine in the QP-nano Tutorial [QP-nano 08].

2.4.2 Pedestrian state machine

The actual traffic light controller hardware will certainly provide a push button for generating the PED_WAITING event, as well as a switch to generate the ON/OFF events. But many development boards for small MCUs (e.g., the eZ430 USB stick from TI) has no push-buttons or any other way to provide external inputs. For such boards, you need to simulate the pedestrian/operator in a separate state machine. This is actually a good opportunity to demonstrate how to incorporate a second state machine (active object) into the application.

The Pedestrian active object is very simple. It periodically posts the PED_WAITING event to the PELICAN active object and from time to time it turns the crossing offline by posting the OFF event followed by the OFF event. As an exercise, you should draw the state diagram of the Pedestrian state machine from the source code found in the file `<qp>\qpn\examples\msp430\iar\pelican-eZ430\ped.c`, found in the Standard Distribution of QP-nano. Please note that such "reverse engineering" of source code is very easy in QP applications, because the code is always the precise specification of the state machine.

2.5 Step 5: Initializing the QP-nano Application (main())

Listing 2 shows the main.c source file for the PELICAN application, which contains the main() function along with some important data structures required by QP-nano.

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

     /*..........................................................................*/
 (4) static QEvent  l_pelicanQueue[2];
 (5) static QEvent  l_pedQueue[1];

     /* QF_active[] array defines all active object control blocks --------------*/
 (6) QActiveCB const Q_ROM Q_ROM_VAR QF_active[] = {
 (7)     { (QActive *)0,          (QEvent *)0,    0                      },
 (8)     { (QActive *)&AO_Pelican, l_pelicanQueue, Q_DIM(l_pelicanQueue) },
 (9)     { (QActive *)&AO_Ped,     l_pedQueue,     Q_DIM(l_pedQueue)     }
     };

     /* make sure that the QF_active[] array matches QF_MAX_ACTIVE in qpn_port.h */
(10) Q_ASSERT_COMPILE(QF_MAX_ACTIVE == Q_DIM(QF_active) - 1);

     /*..........................................................................*/
     void main (void) {
(11)     Pelican_ctor();                          /* instantiate the  Pelican AO */
(12)     Ped_ctor();                              /* instantiate the  Ped     AO */

(13)     BSP_init();                                     /* initialize the board */

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

Listing 2 The file main.c of the PELICAN crossing application.

(1) Every application C-file that uses QP-nano must include the qpn_port.h header file. This header file contains the specific adaptation of QP-nano to the given processor and compiler, which is called a port. The QP port is typically located in the application directory.

(2) The bsp.h header file contains the interface to the Board Support Package and is located in the application directory.

(3) The pelican.h header file contains the declarations of events and other facilities shared among the components of the PELICAN application. This header file is located in the application directory.

(4-5) The application must provide storage for the event queues of all active objects used in the application. In QP-nano the storage is provided at compile time through the statically allocated arrays of events. Events are represented as instances of the QEvent structure declared in the <qpn>\include\qepn.h header file, included from qpn_port.h. Each event queue of an active object can have a different length and you need to decide this length based on your knowledge of the application.

(6) Every QP-nano application must provide the constant array QF_active[], which defines all active object control blocks in the application. The control block QActiveCB structure groups together: (1) the pointer to the corresponding active object instance, (2) the pointer to the event queue buffer of the active object, and (3) the length of the queue buffer.

QP-nano uses every opportunity to place data in ROM rather than in the precious RAM. The QActiveCB structure contains data elements known at compile time, so that these elements can be placed in ROM, as opposed to placing them in the active object structure (RAM). That way, you can save anywhere from 10 to 80 bytes of RAM, depending on the number of active objects and the pointer size of the target CPU.

The Q_ROM macro is necessary on some CPU architecture to enforce placement of constant objects, such as the QF_active[] array, in ROM. On Harvard architecture CPUs (such as 8051 or AVR), the code and data spaces are separate and are accessed through different CPU instructions. The const keyword is not sufficient to place data in ROM, and various compilers often provide specific extended keywords to designate the code space for placing constant data, such as the "__code" extended keyword in the IAR 8051 compiler. The macro Q_ROM hides such non-standard extensions. If you don't define Q_ROM in qepn_port.h, it will be defined to nothing in the qepn.h platform-independent header file.

The Q_ROM_VAR macro defines the compiler-specific directive for accessing a constant object in ROM. Many compilers for 8-bit MCUs provide different size pointers for accessing objects in various memories. Constant objects allocated in ROM often mandate the use of specific-size pointers (e.g., far pointers) to get access to ROM objects. The macro Q_ROM_VAR specifies the kind of the pointer to be used to access the ROM objects. An example of valid Q_ROM_VAR macro definition is: __far (Freescale HC(S)08 compiler).

(7) The first entry (QF_active[0]) corresponds to active object priority of zero, which is reserved for the idle task and cannot be used for any active object.

(8-9) The QF_active[] entries starting from index one define the active object control blocks in the order of their relative priorities. The maximum number of active objects in QP-nano cannot exceed 8.

NOTE: The order or the active object control blocks in the QF_active[] array defines the priorities of active objects. This is the only place in the code where you assign active object priorities.

(10) This compile-time assertion (see Chapter 6 in [PSiCC2]) ensures that the dimension of the QF_active[] array matches the number of active objects QF_MAX_ACTIVE defined in the qpn_port.h header file.

In QP-nano, QF_MAX_ACTIVE denotes the exact number of active objects used in the application. The macro QF_MAX_ACTIVE must be defined in qpn_port.h header file, because QP-nano uses the macro to optimize the internal algorithms based on the number of active objects. The compile-time assertion in line (10) makes sure that the configured number of active objects indeed matches exactly the number of active object control blocks defined in the QF_active[] array.

NOTE: All active objects in QP-nano must be defined at compile time. This means that all active objects exist from the beginning and cannot be started (or stopped) later, as it is possible in the full-version QP.

(11-12) The main() function must first explicitly calls all active object constructors.

(13) The board support package (BSP) is initialized.

(14) At this point, you have initialized all components and have provided to the QF-nano framework all the information it needs to manage your application. The last thing you must do is to call the function `QF_run()` to pass the control to the QF-nano framework.

Overall, the application startup is much simpler in QP-nano than in full-version QP. Neither event pools, nor publish-subscribe lists are supported, so you don't need to initialize them. You also don't start active objects explicitly. The QF-nano framework starts all active objects defined in the `QF_active[]` array automatically just after it gets control in `QF_run()`.

2.6 Step 6: Implementing Active Objects

Implementing active objects with QP-nano is very similar to the full-version QP. You derive the concrete active object structures from the `QActive` base structure provided in QP-nano. Your main job is to elaborate the state machines of the active objects, which is also very similar as in the full-version QP. The only important difference is that state handler functions in QP-nano do not take the event pointer as the second argument. In fact, QP-nano state handlers take only one argument—the "me" pointer. The current event is embedded inside the state machine itself and is accessible via the "me" pointer. QP-nano provides macros `Q_SIG()` and `Q_PAR()` to conveniently access the signal and the scalar parameter of the current event, respectively.

Listing 3 shows the implementation of the `Pelican` active object from the PELICAN crossing applicaiton, which illustrates all aspects of implementing active objects with QP-nano. Please correlate this implementation with the PELICAN state diagram in Figure 5 The PELICAN state machine.

```
    #include "qpn_port.h"
    #include "bsp.h"
    #include "pelican.h"

    /* Pelican class -----------------------------------------------------------*/
(1) typedef struct PelicanTag {
(2)     QActive super;                                  /* derived from QActive */
        uint8_t flashCtr;                             /* pedestrian flash counter */
    } Pelican;

(3) static QState Pelican_initial         (Pelican *me);
(4) static QState Pelican_offline         (Pelican *me);
    static QState Pelican_operational     (Pelican *me);
    static QState Pelican_carsEnabled     (Pelican *me);
    static QState Pelican_carsGreen       (Pelican *me);
    static QState Pelican_carsGreenNoPed  (Pelican *me);
    static QState Pelican_carsGreenPedWait(Pelican *me);
    static QState Pelican_carsGreenInt    (Pelican *me);
    static QState Pelican_carsYellow      (Pelican *me);
    static QState Pelican_pedsEnabled     (Pelican *me);
    static QState Pelican_pedsWalk        (Pelican *me);
    static QState Pelican_pedsFlash       (Pelican *me);

    enum PelicanTimeouts {                          /* various timeouts in ticks */
        CARS_GREEN_MIN_TOUT = BSP_TICKS_PER_SEC * 8,      /* min green for cars */
        CARS_YELLOW_TOUT = BSP_TICKS_PER_SEC * 3,           /* yellow for cars */
        PEDS_WALK_TOUT   = BSP_TICKS_PER_SEC * 3,      /* walking time for peds */
        PEDS_FLASH_TOUT  = BSP_TICKS_PER_SEC / 5,  /* flashing timeout for peds */
        PEDS_FLASH_NUM   = 5*2,                   /* number of flashes for peds */
        OFF_FLASH_TOUT   = BSP_TICKS_PER_SEC / 2   /* flashing timeout when off */
    };

    /* Global objects ----------------------------------------------------------*/
(5) Pelican AO_Pelican;     /* the single instance of the Pelican active object */
```

```
    /*..........................................................................*/
    void Pelican_ctor(void) {
(6)     QActive_ctor((QActive *)&AO_Pelican, (QStateHandler)&Pelican_initial);
    }

    /* HSM definition ----------------------------------------------------------*/
    QState Pelican_initial(Pelican *me) {
(7)     return Q_TRAN(&Pelican_operational);
    }
    /*..........................................................................*/
    QState Pelican_operational(Pelican *me) {
(8)     switch (Q_SIG(me)) {
            case Q_ENTRY_SIG: {
                BSP_signalCars(CARS_RED);
                BSP_signalPeds(PEDS_DONT_WALK);
(9)             return Q_HANDLED();
            }
            case Q_INIT_SIG: {
                return Q_TRAN(&Pelican_carsEnabled);
            }
            case OFF_SIG: {
                return Q_TRAN(&Pelican_offline);
            }
        }
(10)    return Q_SUPER(&QHsm_top);
    }
    /*..........................................................................*/
    QState Pelican_carsEnabled(Pelican *me) {
        switch (Q_SIG(me)) {
            case Q_EXIT_SIG: {
                BSP_signalCars(CARS_RED);
                return Q_HANDLED();
            }
            case Q_INIT_SIG: {
                return Q_TRAN(&Pelican_carsGreen);
            }
        }
        return Q_SUPER(&Pelican_operational);
    }
    /*..........................................................................*/
    QState Pelican_carsGreen(Pelican *me) {
        switch (Q_SIG(me)) {
            case Q_ENTRY_SIG: {
(11)            QActive_arm((QActive *)me, CARS_GREEN_MIN_TOUT);
                BSP_signalCars(CARS_GREEN);
                return Q_HANDLED();
            }
            case Q_INIT_SIG: {
                return Q_TRAN(&Pelican_carsGreenNoPed);
            }
        }
        return Q_SUPER(&Pelican_carsEnabled);
    }
    . . .
    /* other state handlers ... */
```

Listing 3 The PELICAN active object definition (file pelican.c). Boldface indicates QP-nano facilities.

(1) This structure defines the Pelican active object.

(2) The Pelican active object structure derives from the framework structure QActive, as described in the sidebar “Single Inheritance in C” in Chapter 1 of [PSiCC2].

(3) The `Pelican_initial()` function defines the top-most initial transition in the `Pelican` state machine. The only difference from the full-version QP is that the initial pseudostate function does not take the initial event parameter.

(4) The state-handler functions in QP-nano also don't take the event parameter. (In QP-nano, the current event is embedded in the state machine.) As in the full-version QP, a state handler function in QP-nano returns a pointer the superstate handler function.

(5) In this line the global `AO_Pelican` active object is defined. Note that actual structure definition for the `Pelican` active object is accessible only locally at the file scope of the `pelican.c` file.

NOTE: QP-nano assumes that all global or static variables without explicit initialization value are initialized to zero upon the system startup, which is a requirement of the ANSI-C standard. You should make sure that your startup code clears the `.BSS` section before calling `main()`.

(6) As always, the derived structure is responsible for initializing the part inherited from the base structure. The "constructor" of the base class `QActive_ctor()` puts the state machine in the initial pseudostate `&Pelican_initial`. The constructor also initializes the priority of the active object based on the `QF_active[]` array.

(7) The top-most initial transition to state `&Pelican_operational` is specified with the `Q_TRAN()` macro.

(8) Every state handler is structured as a switch statement that discriminates based on the signal of the event, which in QP-nano is obtained by the macro `Q_SIG(me)`.

(9) The entry/exit actions are terminated with "`return Q_HANDLED()`", which informs QEP-nano that the initial transition has been handled.

(10) You terminate the case statement with "`return Q_HANDLED()`", which informs QEP-nano that the initial transition has been handled.

(13) The final return from a state handler function designates the superstate of that state, which is exactly the same as in the full-version QP. QEP-nano provides the "top" state as a state handler function `&QHsm_top`, and therefore the `Pelican_operational()` state handler returns the pointer `&QHsm_top`. (see the PELICAN state diagram in Figure 5)

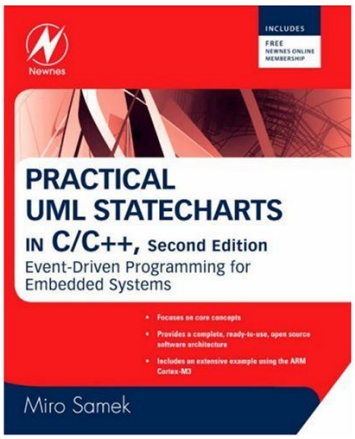
(11) You arm the QP-nano time event (timer) associated with the active object via the call to `QActive_arm()` function.

3 References

Document	Location
[PSiCC2] "Practical UML Statecharts in C/C++, Second Edition", Miro Samek, Newnes, 2008, ISBN 0750687061	Available from most online book retailers, such as amazon.com. See also: http://www.state-machine.com/psicc2.htm
[QP-nano 08] "QP-nano Reference Manual", Quantum Leaps, LLC, 2008	http://www.state-machine.com/doxygen/qpn/
[QL AN-Directory 07] "Application Note: QP Directory Structure", Quantum Leaps, LLC, 2007	http://www.state-machine.com/doc/-AN_QP_Directory_Structure.pdf
[Samek+ 06a] "Build a Super Simple Tasker", Embedded Systems Design, Miro Samek and Robert Ward, July 2006.	http://www.embedded.com/shared/-printableArticle.jhtml?articleID=190302110
[Samek 06b]"UML Statecharts at $10.99", Dr. Dobb's Journal, Miro Samek, May 2006	http://www.ddj.com/dept/embedded/188101799

4 Contact Information

Quantum Leaps, LLC
103 Cobble Ridge Drive
Chapel Hill, NC 27516
USA
+1 866 450 LEAP (toll free, USA only)
+1 919 869-2998 (FAX)
e-mail: info@quantum-leaps.com
WEB : http://www.quantum-leaps.com
http://www.state-machine.com



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