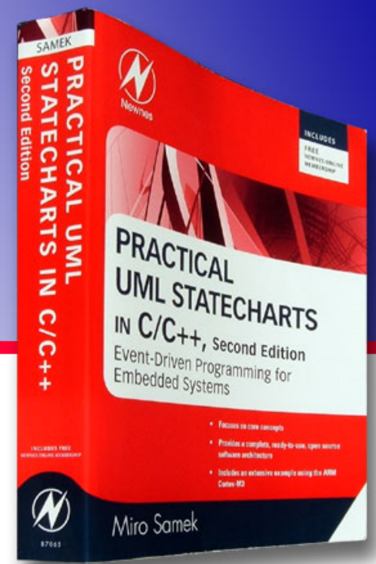
Quantum® LeaPs
innovating embedded systems



QDK™-nano
Atmel AVR-xmega-IAR

Document Revision B
February 2010

Copyright © Quantum Leaps, LLC

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

Table of Contents

1 Introduction 1
1.1 About QP-nano™ 2
1.2 What's Included in the QDK-nano? 3
1.3 Licensing QP-nano™ 3

2 Getting Started 4
2.1 Installation 4
2.2 Building the Examples 5
2.3 Programming and Debugging the AVR-xmega Device 6
2.4 Executing the Examples 6
2.5 Setting Stack and Heap Size 7

3 Non-Preemptive Configuration of QP-nano 8
3.1 The qpn_port.h Header File 8
3.2 ISRs in the Non-preemptive "Vanilla" Configuration 9
3.3 QP Idle Loop Customization in QF_onIdle() 10

4 Preemptive Configuration with QK-nano 12
4.1 The QK-specific Interrupt Processing in the BSP 14
4.1.1 The QK-nano in Assembly 15
4.2 Idle Loop Customization in the QK-nano Port 15

5 BSP for AVR-xmega 17
5.1 Board Initialization and the Timer Tick 17
5.2 Starting Interrupts in QF_onStartup() 17
5.3 Assertion Handling Policy in Q_onAssert() 18

6 Related Documents and References 19

7 Contact Information 20

Copyright © Quantum Leaps, LLC. All Rights Reserved.

AVR®

IAR SYSTEMS

1 Introduction

This **QP-nano™ Development Kit** (QDK-nano) describes how to use QP-nano™ event-driven platform with the Atmel AVR-xmega and the IAR toolset for AVR (EWAVR). The actual hardware/software used to test this QDK-nano as shown in Figure 1 and described below:

Figure 1 Atmel STK600 board with the ATxmega128A1 device and the JTAGICE-mkII emulator (see also Atmel Application Note AVR1900: Getting started with ATxmega128A1 on STK600 [Atmel 08]).

STK600 board

Power from PC

USB to PC

JTAG-mkII

ATxmega128A1 device

JTAG cable

Port D-LEDs connection

8 LEDs connected to PORTD

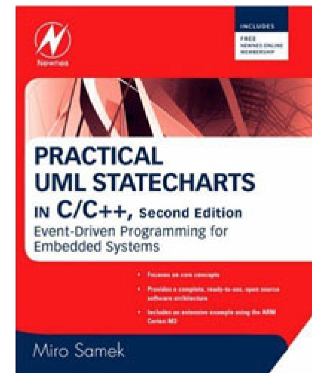
Copyright © Quantum Leaps, LLC. All Rights Reserved.

1. STK600 board with STK600-TQFP100 socket board and STK600 – RC100X-13 routing board
2. ATXmega128A1 device mounted in the STK600-TQFP100 socket board
3. JTAG-mkII debugger
4. IAR Embedded Workbench for AVR 5.30 KickStart edition
5. QP-nano v4.1.00 or higher.

As shown in Figure 1, the Atmel's STK600 is powered via a USB cable and is connected to the JTAG-mkII emulator via the JTAG ribbon cable. This QDK has been tested with the ATXmega128A1 device with 8KB of RAM and 128KB of onboard flash. However, the described port should be applicable to most AVR-xmega devices (but **not** to the classic AVRs, such as ATtiny and ATmega families) with the Programmable Multi-level Interrupt Controller, PMIC.

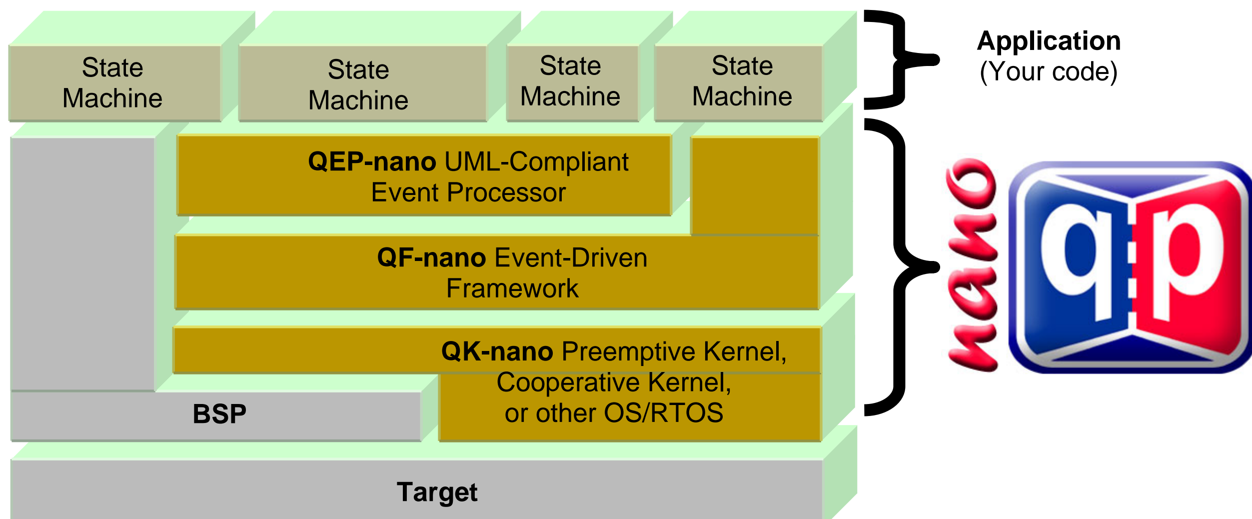
NOTE: This QDK-nano covers the AVR-xmega processor family, but does **not** apply to the classic AVRs, such as ATtiny and ATmega. AVR-xmega uses different interrupt locking policy in ISRs and different interrupt handling hardware (the Programmable Multi-level Interrupt Controller, PMIC), which is not available in the classic AVRs. Please refer to the QDK-AVR for the information about using the QP™ frameworks with the classic AVR devices.

1.1 About QP-nano™



QP-nano™ is an ultra-lightweight, open source, state machine framework and RTOS for developing real-time embedded applications. QP-nano has been specifically designed to enable event-driven programming with concurrent hierarchical state machines (UML statecharts) on low-end 8- and 16-bit single-chip MCUs and DSPs, such as **AVR**, PICmicro, PIC24/dsPIC, 8051, MSP430, 68HC08/11/12, R8C/Tiny, H8/S, TMS320C28x, Cypress PSoC, 8051 and others alike, with a few hundred bytes of RAM and a few kilobytes of ROM. With QP-nano, coding of modern state machines directly in C is a non-issue. No big design automation tools are needed.

Figure 2 QP-nano™ components and their relationship with the target hardware, board support package (BSP), and the application comprised of state machines



Copyright © Quantum Leaps, LLC. All Rights Reserved.

As shown in Figure 2, QP-nano consists of a universal UML-compliant event processor (QEP-nano), a higly portable event-driven framework (QF-nano), and a tiny run-to-completion kernel (QK-nano).

The QP-nano framework can manage up to 8 concurrently executing hierarchical state machines and requires only 1-2KB of code (ROM) and just several bytes of RAM. This tiny footprint, especially in RAM, makes QP-nano ideal for high volume, cost-sensitive applications, such as motor control, lighting control, capacitive touch sensing, remote access control, RFID, thermostats, small appliances, toys, power supplies, battery chargers, or just about any **system-on-a-chip** (SoC or ASIC) that contains a small processor inside. Also, because the event-driven paradigm naturally uses the CPU only when handling events and otherwise can very easily switch the CPU into a low-power sleep mode, QP-nano is particularly suitable for ultra-low power applications, such as wireless sensor networks or implantable medical devices.

All versions of QP, including QP-nano, are described in detail in the book "*Practical UML Statecharts in C/C++, 2nd Edition: Event-Driven Programming for Embedded Systems*" [PSiCC2] published by Newnes in 2008 (see www.state-machine.com/psicc2). QP-nano has a strong user community and has been applied worldwide in industries, such as: consumer electronics, telecommunications, equipment, industrial automation, transportation systems, medical devices, and many more. Please refer to the www.state-machine.com website for more information.

1.2 What's Included in the QDK-nano?

This QDK-nano provides the QP-nano port to AVR-xmega with the IAR EWAVR toolset, the Board Support Package (BSP) and two versions of the PEdestrian LIght CONtrolled (PELICAN) crossing example application described in the Application Note “PELICAN Crossing Example” [QL AN-PELICAN 08].

1. PELICAN crossing with the cooperative “Vanilla” kernel; and
2. PELICAN crossing with the preemptive run-to-completion QK-nano kernel.

NOTE: Even though this QDK-nano is based on a specific development board (STK600 in this case), the most important parts of the QP-nano ports are applicable to all AVR-xmega- based MCUs.

1.3 Licensing QP-nano™

The **Generally Available (GA)** distribution of QP-nano™ available for download from the www.state-machine.com/downloads website is offered with the following two licensing options:

- The GNU General Public License version 2 (GPL) as published by the Free Software Foundation and appearing in the file `GPL.TXT` included in the packaging of every Quantum Leaps software distribution. The GPL *open source* license allows you to use the software at no charge under the condition that if you redistribute the original software or applications derived from it, the complete source code for your application must be also available under the conditions of the GPL (GPL Section 2[b]).

- One of several Quantum Leaps commercial licenses, which are designed for customers who wish to retain the proprietary status of their code and therefore cannot use the GNU General Public License. The customers who license Quantum Leaps software under the commercial licenses do not use the software under the GPL and therefore are not subject to any of its terms.

For more information, please visit the licensing section of our website at: www.state-machine.com/licensing.

2 Getting Started

This section describes how to install, build, and use QDK-nano-AVR-xmega-IAR based on two examples. This information is intentionally included early in this document, so that you could start using QDK-nano™ as soon as possible.

NOTE: Every QDK-nano™ contains only example(s) pertaining to the specific MCU and compiler, but does not include the platform-independent baseline code of QP-nano™, which is available for a separate download. It is strongly recommended that you read Chapter 12 in [PSiCC2] before you start with this QDK-nano™.

2.1 Installation

The QDK-nano code is distributed in a ZIP archive (`qdkn_avr-xmega-iar_<ver>.zip`, where `<ver>` stands for a specific QDK-nano version, such as 4.1.01). You should uncompress the archive into the same directory in which you've installed QP-nano™. The QP-nano™ installation will be referred henceforth as QP-nano Root Directory `<qpn>`.

Listing 1 Directories and files after installing QP-nano baseline and QDK-nano-AVR-xmega-IAR. The highlighted directories and files are provided in the QDK-nano-AVR-xmega-IAR ZIP file.

```
<qpn>/                        - QP-nano Root Directory
  +-examples/                 - QP-nano examples
  | +-avr\                    - examples for AVR
  | | +-iar\                  - examples compiled with the IAR EWAVR toolset
  | | | +-pelican-stk600-atxmega128a1\  - PELICAN example for STK600
  | | | | +-dbg\              - directory containing the debug build
  | | | | | +-pelican.d90 - image of the application
  | | | | +-rel\              - directory containing the release build
  | | | | | +-pelican.d90 - image of the application
  | | | | | +-. . .
  | | | | +-pelican.ewp  - IAR project file to build the PELICAN application
  | | | | +-pelican.eww  - IAR workspace file to build the PELICAN application
  | | | | +-bsp.h        - Board Support Package include file
  | | | | +-bsp.c        - Board Support Package implementation
  | | | | +-pelican.h
  | | | | +-main.c
  | | | | +-pelican.c
  | | | | +-ped.c
  | | | | +-qpn_port.h   - QP-nano port
  | | | |
  | | | +-pelican-qk-stk600-atxmega128a1\  - PELICAN example for STK600
  | | | | +-dbg\             - directory containing the debug build
  | | | | | +-pelican-qk.d90 - image of the application
  | | | | +-rel\             - directory containing the release build
  | | | | | +-pelican-qk.d90 - image of the application
  | | | | | +-. . .
  | | | | +-pelican-qk.ewp  - IAR project file to build the PELICAN application
  | | | | +-pelican-qk.eww  - IAR workspace file to build the PELICAN application
  | | | | +-bsp.h        - Board Support Package include file
  | | | | +-bsp.c        - Board Support Package implementation
  | | | | +-pelican.h
  | | | | +-main.c
  | | | | +-pelican.c
```

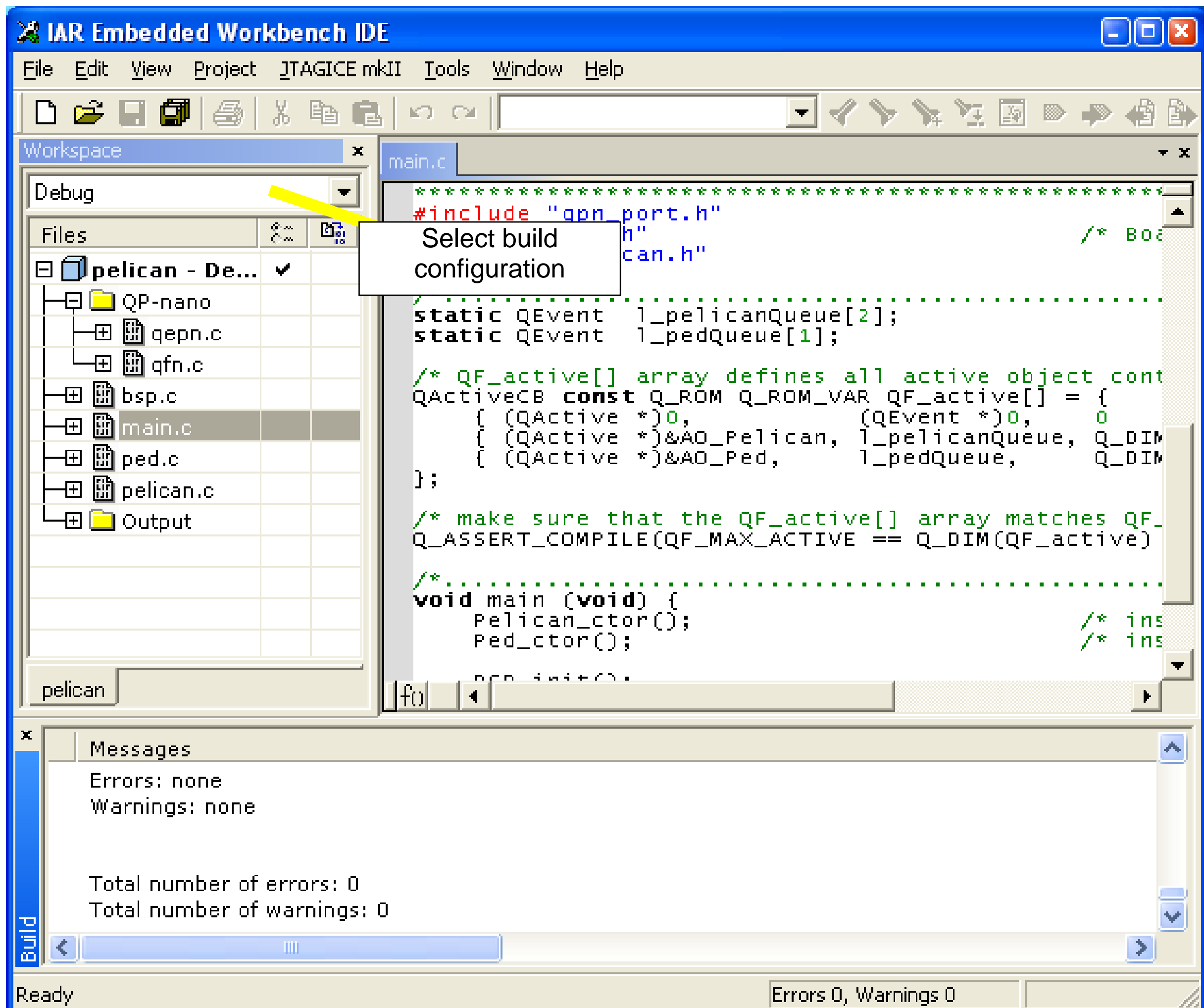
Copyright © Quantum Leaps, LLC. All Rights Reserved.

```
| | | | +-ped.c
| | | | +-qpn_port.h    - QP-nano port
+-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
+-source/                - QP-nano source files
| +-qepn.c               - QEP-nano
| +-qfn.c                - QF-nano
| +-qkn.c                - QK-nano (required only in QK-nano configuration)
```

2.2 Building the Examples

The examples included in this QDK-nano are based on the standard PELICAN crossing application implemented with active objects (see Quantum Leaps Application Note: “PELICAN Crossing Application” [QL AN-PELICAN 08] includes in this QDK-nano).

Figure 3 IAR Embedded Workbench for AVR with the pelican.eww workspace.

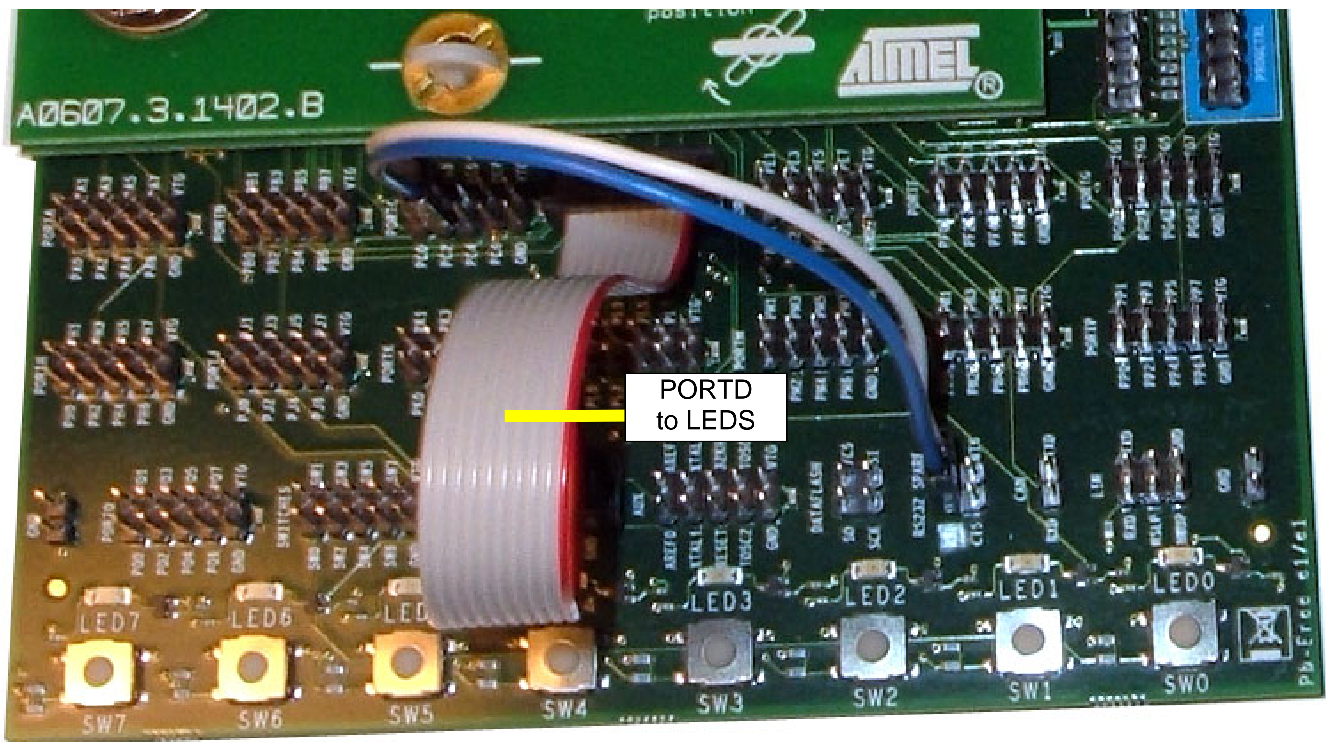


The example directory contains the IAR Embedded Workbench workspace file `pelican.eww` that you can load into the IAR EW IDE. The workspace contains two build configurations (Debug and Release) that you can select with the drop-down list, as shown in Figure 3.

2.3 Programming and Debugging the AVR-xmega Device

Figure 4Figure 4 shows how to make internal connections on the STK600 board to enable the external LEDs to test this QDK-nano (see also Figure 1). The LEDs are connected to PORTD with the flat-band cable as shown Figure 4. Please refer to the Atmel Application Note "AVR1900: Getting started with ATxmega128A1 on STK600" [Atmel 08]).

Figure 4 Making internal connections on the STK600 board to use LEDs



As described in the Atmel Application Note "AVR1900: Getting started with ATxmega128A1 on STK600", before you can start using the STK600 with the ATxmega128A1 device, you must program the STK600 for the desired VCC level of 3.3V. The programming is accomplished by means of the AVR Studio host application, which is available for free download from Atmel (http://www.atmel.com/dyn/Products/-tools_card.asp?tool_id=2725).

After you program the VCC into the STK600, you don't need the AVR Studio anymore, because the he IAR Embedded Workbench IDE (see Figure 3) and the C-SPY debugger can program the AVR-xmega device and debug the code over the JTAGICE-mkII pod.

2.4 Executing the Examples

An example run of the PELICAN application is shown in Figure 1. The external LEDs connected to PORTD should start blinking. The LEDs are connected as follows:

Copyright © Quantum Leaps, LLC. All Rights Reserved.

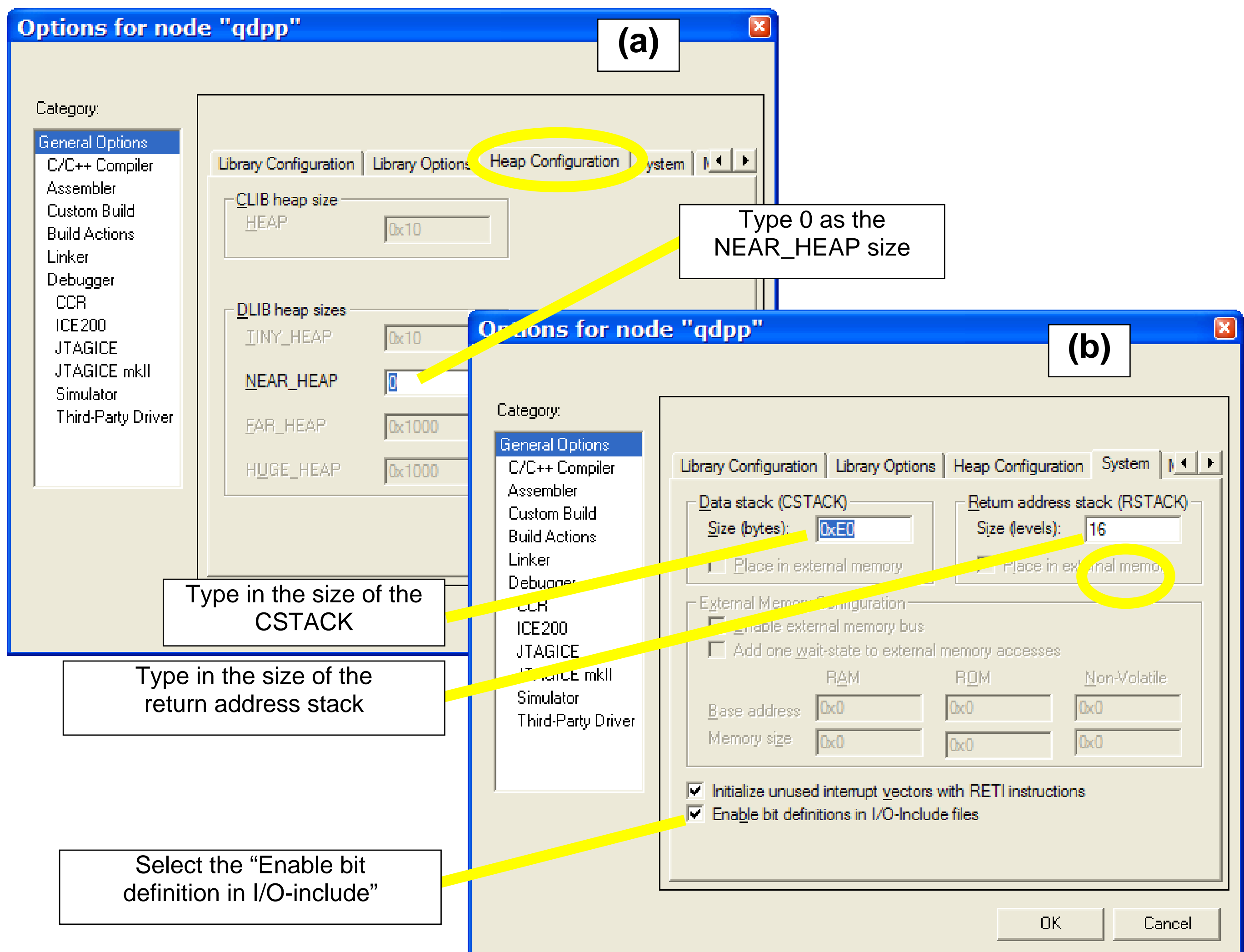
```
LED 0 (PORTD[0]) -> red light for cars
LED 1 (PORTD[1]) -> yellow light for cars
LED 2 (PORTD[2]) -> green light for cars
LED 3 (PORTD[3]) unused
LED 4 (PORTD[4]) -> WALK signal for pedestrians
LED 5 (PORTD[5]) -> DON'T WALK signal for pedestrians
LED 6 (PORTD[6]) unused
LED 7 (PORTD[7]) -> idle loop activity
```

LED 7 displays the activity of the idle loop, where higher intensity of LED 7 corresponds to more CPU cycles spent in the idle loop.

2.5 Setting Stack and Heap Size

The most important IAR linker options used for building the final application image are the CPU selection, and the stack size. The following Figure 5 shows selecting the heap size (zero) and stack size.

Figure 5 Setting the heap size to zero (a), and selecting the stack size (b)



Copyright © Quantum Leaps, LLC. All Rights Reserved.

3 Non-Preemptive Configuration of QP-nano

The example of using QP-nano with the cooperative “Vanilla” kernel is located in the directory: `<qpn>\examples\avr-xmega\iar\pelican-butterfly\`. This section describes the generic QP-nano configuration, which consist of the `qpn_port.h` header file. The board-specific elements are common for both the non-preemptive and preemptive (QK-nano) configuration and will be covered in Section 5.

3.1 The qpn_port.h Header File

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

NOTE: The QP-nano port to the cooperative “Vanilla” kernel `qpn_port.h` is generic and should not need to change (except for the `QF_MAX_ACTIVE` definition) for other AVR-xmega systems.

Listing 2 qpn_port.h header file for the non-preemptive QF-nano configuration and IAR compiler

```
 (1) #define Q_ROM                     __flash

 (2) #define Q_NFSM
 (3) #define Q_PARAM_SIZE              1
 (4) #define QF_TIMEEVT_CTR_SIZE       2

     /* maximum # active objects--must match EXACTLY the QF_active[] definition  */
 (5) #define QF_MAX_ACTIVE             2

                                        /* interrupt locking policy for task level */
 (6) #define QF_INT_LOCK()             __disable_interrupt()
 (7) #define QF_INT_UNLOCK()           __enable_interrupt()

                                   /* interrupt locking policy for interrupt level */
 (8) #define QF_ISR_NEST                                  /* nesting of ISRs is allowed */

 (9) #include <intrinsics.h>            /* prototypes for the intrinsic functions */
(10) #include <stdint.h>    /* Exact-width integer types. WG14/N843 C99 Standard */

(11) #include "qepn.h"         /* QEP-nano platform-independent public interface */
(12) #include "qfn.h"           /* QF-nano platform-independent public interface */
```

(1) The macro `Q_ROM` allows enforcing placing the constant objects, such as lookup tables, constant strings, etc. in ROM, rather than in the precious RAM. On CPUs with the Harvard architecture (such as Atmel AVR-xmega or 8051), the code and data spaces are separate and are accessed through different CPU instructions. Various compilers often provide specific extended keywords to designate code or data space, such as the “`__flash`” extended keywords in the IAR compiler.

(2) Defining the macro `Q_NFSM` eliminates the code for the simple non-hierarchical FSMs.

(3) The macro `Q_PARAM_SIZE` defines the size (in bytes) of the scalar event parameter. The allowed values are 0 (no parameter), 1, 2, or 4 bytes. If you don’t define this macro in `qpn_port.h`, the default of 0 (no parameter) will be assumed.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

(4) The macro `QF_TIMEEVT_CTR_SIZE` defines the size (in bytes) of the time event down-counter. The allowed values are 0 (no time events), 1, 2, or 4 bytes. If you don't define this macro in qpn_port.h, the default of 0 (no time events) will be assumed.

(5) You must define the `QF_MAX_ACTIVE` macro as the exact number of active objects used in the application. The provided value must be between 1 and 8 and must be consistent with the definition of the `QF_active[]` array in `main.c`.

(6-7) The macros `QF_INT_LOCK()`/`QF_INT_UNLOCK()` define the task-level interrupt locking policy for QP-nano (see Section 12.3.2 in Chapter 12 in [PSiCC2]).

(8) This QP-nano port to AVR-xmega **does** allow nesting of interrupts.

In contrast to the classic AVRs, the AVR-**xmega** devices provide interrupt prioritization in hardware through the Programmable Multi-level Interrupt Controller (PMIC). Also, the AVR-xmega devices do **not** lock interrupts upon the entry to an interrupt service routine (ISR), as it was the case with the classic AVRs. In other words, an ISR does not automatically establish a critical section. All this means that AVR-**xmega** devices can use the simple interrupt locking policy called "unconditional interrupt locking and unlocking" and described in Chapter 7 of [PSiCC2].

NOTE: The simple "unconditional interrupt locking and unlocking" policy requires that you call every QP service **outside** a critical section (with interrupts unlocked), because nesting of critical sections is **not** allowed. In particular, you should **not** lock interrupts inside the ISRs before calling any QP API.

(9) These header files are necessary for supporting intrinsic functions of the IAR compiler.

(10) The IAR compiler provides the C99-standard exact-width integer types are defined in the standard `<stdint.h>` header file.

(11) The `qpn_port.h` must include the QEP-nano event processor interface `qepn.h`.

(12) The `qpn_port.h` must include the QF-nano real-time framework interface `qfn.h`.

3.2 ISRs in the Non-preemptive "Vanilla" Configuration

The IAR AVR compiler supports writing interrupts in C. In the "vanilla" port, the ISRs are identical as in the simplest of all "superloop" (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.

Listing 3 Time tick interrupt calling QF_tick() function to manage armed time events.

```
(1) #pragma vector = TCC0_OVF_vect
(2) __interrupt void TIMER0_COMP_interrupt(void)
    {
        /* No need to clear the interrupt source since the Timer0 compare
        * interrupt is automatically cleared in hardware when the ISR runs.
        */

(3)     QF_tick();
    }
```

(1) The `#pragma vector = ...` sets up the interrupt vector for a given ISR

Copyright © Quantum Leaps, LLC. All Rights Reserved.

(2) The definition of the ISR function must begin with the `__interrupt` extended keyword.

(3) The time-tick ISR must invoke `QF_tick()`, and can also perform other actions, 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, because here interrupts are locked throughout the interrupt processing.

3.3 QP Idle Loop Customization in QF_onIdle()

The cooperative "vanilla" kernel can very easily detect the situation when no events are available, in which case `QF_run()` calls the `QF_onIdle()` callback. You can use `QF_onIdle()` to suspended the CPU to save power, if your CPU supports such a power-saving mode. Please note that `QF_onIdle()` is called repetitively from the event loop whenever the event loop has no more events to process, in which case only an interrupt can provide new events. The `QF_onIdle()` callback is called with interrupts **locked**, because the determination of the idle condition might change by any interrupt posting an event.

AVR supports several power-saving levels (consult the AVR data sheet for details). The following piece of code shows the `QF_onIdle()` callback that puts AVR into the idle power-saving mode. Please note that AVR architecture allows for very **atomic** setting the low-power mode and enabling interrupts at the same time.

Listing 4 `QF_onIdle()` for the non-preemptive ("vanilla") QP-nano port to AVR.

```
(1) void QF_onIdle(void) {         /* entered with interrupts LOCKED, see NOTE01 */

                             /* toggle the LED number 7 on and then off, see NOTE02 */
(2)     LED_ON(7);
        LED_OFF(7);

    #ifdef NDEBUG

                             /* configure idle sleep mode, , adjust to your project */
(3)     SLEEP.CTRL = SLEEP_SMODE_IDLE_gc | SLEEP_SEN_bm;
          /* never separate the following two assembly instructions, see NOTE03 */
(4)     __enable_interrupt();      /* NOTE: the following sleep instruction will */
(5)     __sleep();              /* execute before entering any pending interrupt */
    #else
(6)     QF_INT_UNLOCK();
    #endif
    }
```

(1) The `QF_onIdle()` callback is always called with interrupts locked to prevent any race condition between posting events from ISRs and transitioning to the sleep mode.

(2) The LED[7] is turned on and off to visualize the idle loop activity. Note that the LED is toggled with interrupts locked, to the period of the LED turned on is constant and does not include any time spent in the ISRs.

(3) The `SLEEP.CTRL` register is loaded with the desired sleep mode. Please note that the sleep mode is not active until the SLEEP command.

(4) The interrupts are unlocked with the intrinsic function, which emits the `SEI` instruction.

(5) The sleep mode is activated with the intrinsic function, which emits the `SLEEP` instruction.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

NOTE: The AVR datasheet is very specific about the behavior of the SEI-SLEEP instruction pair. Due to pipelining of the AVR core, the SLEEP instruction is guaranteed to execute before entering any potentially pending interrupt. This means that enabling interrupts and activating the sleep mode is **atomic**, as it should be to avoid non-deterministic sleep.

(6) In the DEBUG configuration the interrupts are simply unlocked.

NOTE: Every path through `QF_onIdle()` callback function must ultimately unlock interrupts.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

4 Preemptive Configuration with QK-nano

The QP port with the preemptive kernel (QK) is remarkably simple and very similar to the "vanilla" port. In particular, the interrupt locking/unlocking policy is the same, and the BSP is identical, except some additions to the ISRs. The PELICAN example for the QK port is provided in the directory `<qpn>\examples\avr-xmega\iar\pelican-qk-stk600-atxmega128a1`. You configure and customize QP-nano through the header file `qpn_port.h`, which is included by the QP-nano source files (`qepn.c`, `qfn.c`, and `qkn.c`) as well as in all your application C modules. The following Listing 5 shows the `qpn_port.h` header file for the QK-nano port. Except for the highlighted fragments, the listing is identical as in the non-preemptive case (Listing 2)

Listing 5 `qpn_port.h` header file for the preemptive QK-nano configuration

```
     #define Q_ROM                       __flash

     #define Q_NFSM
     #define Q_PARAM_SIZE                1
     #define QF_TIMEEVT_CTR_SIZE         2

     /* maximum # active objects--must match EXACTLY the QF_active[] definition  */
     #define QF_MAX_ACTIVE               2

                                        /* interrupt locking policy for IAR compiler */
     #define QF_INT_LOCK()               __disable_interrupt()
     #define QF_INT_UNLOCK()             __enable_interrupt()

                                     /* interrupt locking policy for interrupt level */
     #define QF_ISR_NEST
                                                     /* QK interrupt service routine */
 (1) #define QK_ISR(name_) \
 (2) __interrupt void name_(void) { \
 (3)     uint8_t pmic_sta; \
 (4)     void name_##_ISR(void); \
 (5)     name_##_ISR(); \
 (6)     __disable_interrupt(); \
 (7)     pmic_sta = PMIC.STATUS; \
 (8)     if ((pmic_sta == 0x01) || (pmic_sta == 0x02) || (pmic_sta == 0x04)) { \
 (9)         if (QF_readySet_ != 0) { \
(10)             QK_end_of_interrupt(); \
(11)             QK_schedule_(); \
(12)             __asm("sei\n" \
(13)                   "out 0x3C,r4\n" \
                       "out 0x3B,r27\n" \
                       "out 0x39,r26\n" \
                       "out 0x38,r25\n" \
                       "out 0x3F,r24\n" \
                       "ld  r16,y+\n" \
                       "ld  r17,y+\n" \
                       "ld  r18,y+\n" \
                       "ld  r19,y+\n" \
                       "ld  r20,y+\n" \
                       "ld  r21,y+\n" \
                       "ld  r22,y+\n" \
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

```
                        "ld  r23,y+\n" \
                        "ld  r0,y+\n" \
                        "ld  r1,y+\n" \
                        "ld  r2,y+\n" \
                        "ld  r3,y+\n" \
                        "ld  r30,y+\n" \
                        "ld  r31,y+\n" \
                        "ld  r24,y+\n" \
                        "ld  r25,y+\n" \
                        "ld  r26,y+\n" \
                        "ld  r27,y+\n" \
                        "ld  r4,y+\n" \
(14)                    "ret"); \
              } \
          } \
          else { \
(15)          __enable_interrupt(); \
          } \
      } \
(16) void name_##_ISR()

     void QK_end_of_interrupt(void);

     #include <intrinsics.h>           /* prototypes for the intrinsic functions */

     #include <stdint.h>    /* Exact-width integer types. WG14/N843 C99 Standard */
     #include "qepn.h"         /* QEP-nano platform-independent public interface */
     #include "qfn.h"           /* QF-nano platform-independent public interface */
     #include "qkn.h"           /* QK-nano platform-independent public interface */
```

(1) The macro `QK_ISR()` takes the parameter ‘`name_`’, which is the name of the ISR

(2) The definition of the ISR function must begin with the `__interrupt` extended keyword.

(3) The local (register) variable `pmic_sta` is declared to hold the PMIC status.

(4) The prototype of the interrupt handler in C is provided. This is the function that performs the actual work of this ISR.

(5) The interrupt handler in C is called to performs the actual work of this ISR.

NOTE: The interrupt handler in C runs with interrupts unlocked.

(6) After the interrupt handler returns, interrupts are disabled by clearing the Interrupt Enable mask in the SR (CLI instruction).

(7) The PMIC status is loaded into the register variable `pmic_sta`.

(8) The PMIC status is examined for interrupt preemptions. The lowest-order 3 bits of the PMIC status are HILVLEX, MEDVLEX, and LOLVLEX, which are set when the corresponding interrupt level is active. In case interrupts don’t nest, only one of these three bits is set, which results in the PMIC status values of 0x01, 0x02, or 0x04. When one of these bit patterns is detected, we can be sure that the interrupts don’t nest, so it is appropriate to check for asynchronous preemptions.

NOTE: The standard technique of incrementing, decrementing, and testing of the `QK_intNest_` level is not applicable to AVR-xmega, because the hardware does not lock interrupts upon the entry

Copyright © Quantum Leaps, LLC. All Rights Reserved.

to the ISR. This means that incrementing `QK_intNest_`, no matter how early in the interrupt sequence, is **not** a reliable way of detecting interrupt nesting, because interrupt preemption can occur before `QK_intNest_` gets incremented. The test of the `PMIC.STATUS` register is the only reliable way of detecting interrupt nesting on AVR-xmega.

(9) The `QF_readySet_` is examined. Invoking the QK scheduler makes sense only when the system has any events to process, that is when the ready set is not empty.

(10) The function `QK_end_of_interrupt()` to called to end the interrupt in the PMIC. The function is coded in assembly and contains just one instruction `RETI`. After the `QK_end_of_interrupt()`, returns, the PMIC state changes to clear the current interrupt level bit in the PMIC status register.

NOTE: A function call on AVR-xmega generates exactly the same stack frame as an interrupt, that is, the three-byte PC is pushed to the stack. The function `QK_end_of_interrupt()` is empty, but unlike a regular function, it returns with the `RETI` instruction. On AVR-xmega, executing `RETI` has a side-effect of **changing the state** of the PMIC to end processing the current interrupt level.

(11) The QK scheduler `QK_schedule_` is called to perform any asynchronous preemption. Please note that the scheduler is called with interrupts **locked**, as required by the QK.

(12) After the QK scheduler returns, interrupts are unlocked to return to the task level with interrupts enabled.

(13) This set of assembly instructions is exactly the ISR epilogue generated by the IAR compiler for the all `__interrupt` functions, with one difference.

NOTE: The ISR epilogue has been tested for the small and large memory models supported by the IAR AVR compiler. The epilogue might possibly need some adjustment for different compiler options, so that it always matches **exactly** the compiler-synthesized interrupt prologue.

(14) The epilogue returns with regular function return `RET` instead of the `RETI` instruction, because the `RETI` instruction for this interrupt level has been already executed in step (9).

(15) If the branch involving the QK scheduler was not taken, the interrupts are re-enabled and the ISR returns via the epilogue generated by the IAR compiler. The epilogue is almost identical to that one coded in assembly at step (12), except that the final return instruction is `RETI`.

(16) The interrupt C-function signature is defined, so that the body of the function can follow immediately the `QK_ISR()` macro (see the following section).

4.1 The QK-specific Interrupt Processing in the BSP

As described in the previous section, the QK port works with the C-compiler generated interrupts, but the actual body of the ISR is synthesized by the macro `QK_ISR()`. The following listing shows an example of the ISR for the QK kernel defined in the BSP of the example application.

Listing 6 The time tick ISR for the QK kernel.

```
(1) #pragma vector = TCC0_OVF_vect
(2) QK_ISR(TCC0_OVF) {                        /* interrupt service routine for QK */

        /* No need to clear the interrupt source since the Timer0 compare
        * interrupt is automatically cleared in hardware when the ISR runs.
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

```
          */
(3      QF_tick();
    }
```

(1) The `#pragma vector = ...` sets up the interrupt vector for a given ISR

(2) The definition of the ISR for QK must begin with the `QK_ISR()` macro.

(3 The time-tick ISR must invoke `QF_tick()`, and can also perform other actions, 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, because here interrupts are locked throughout the interrupt processing.

NOTE: The interrupt handler in C returns to the ISR "wrapper" synthesized by the macro `QK_ISR()`, which performs the scheduling and asynchronous preemptions, if necessary.

4.1.1 The QK-nano in Assembly

The AVR-xmega QK port requires coding the `QK_end_of_interrupt()` function in assembly. This function is provided in the module `qk_port.s`, which looks as follows:

```
     RSEG CODE:CODE:NOROOT(2)

    PUBLIC  QK_end_of_interrupt

;*****************************************************************************
;
; The QK_end_of_interrupt function executes the return-from-interrupt (RETI)
; instruction, which updates the state of the PMIC.
;
;*****************************************************************************
QK_end_of_interrupt
    RETI                  ; return from the interrupt

    END
```

4.2 Idle Loop Customization in the QK-nano Port

As described in Chapter 10 of [PSiCC2], the QK-nano idle loop executes only when there are no events to process. The QK allows you to customize the idle loop processing by means of the callback `QK_onIdle()`, which is invoked by every pass through the QK idle loop. You can define the platform-specific callback function `QK_onIdle()` to save CPU power, or perform any other "idle" processing (such as Quantum Spy software trace output).

NOTE: The idle callback `QK_onIdle()` is invoked with interrupts unlocked (which is in contrast to `QF_onIdle()` that is invoked with interrupts locked, see Section).

The following Listing 7 shows an example implementation of `QK_onIdle()` for the ATxmega128A1 MCU. Other AVR-xmega-based embedded microcontrollers handle the power-saving mode very similarly.

Listing 7 QK_onIdle() callback for AVR.

```
     void QK_onIdle(void) {
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

```
                                 /* toggle the LED number 7 on and then off, see NOTE01 */
            QF_INT_LOCK();
            LED_ON(7);
            LED_OFF(7);
            QF_INT_UNLOCK();

        #ifdef NDEBUG
            SLEEP.CTRL = SLEEP_SMODE_IDLE_gc | SLEEP_SEN_bm;
              /* never separate the following two assembly instructions, see NOTE03 */
            __sleep();              /* execute before entering any pending interrupt */
    #endif
        }
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

5 BSP for AVR-xmega

The Board Support Package (BSP) for AVR-xmega is very simple. However, there are some important details that you need to pay attention to.

5.1 Board Initialization and the Timer Tick

The BSP is minimal, but generic for most AVR-xmega devices. The BSP configures the internal 32MHz clock (out of reset, the device comes up with the 2MHz internal clock). Of course, you should adapt this part to choose one of the several available clock options for your particular application. The most important step is initialization of Timer 0 to deliver the time tick interrupt at the desired rate (`BSP_TICKS_PER_SEC`). Also, the PMIC is configured to enable all interrupt levels 1-3.

```
void BSP_init(void) {
    OSC.CTRL |= OSC_RC32MEN_bm;                /* enable internal 32MHz osc. */
    while ((OSC.STATUS & OSC_RC32MEN_bm) == 0) {    /* wait until osc. ready */
    }
    CCP = CCP_IOREG_gc;    /* enable change in the Configuration Change Reg. */
    CLK.CTRL = CLK_SCLKSEL_RC32M_gc;    /* select internal 32MHz osc. source */

    PORTD.DIRSET = 0xFF;             /* all PORTD pins are outputs for LEDs */
    LED_OFF_ALL();                                     /* trun off all LEDs */

                                  /* enable interrupt levels 1-3 in PMIC */
    PMIC.CTRL |= (1 << 0) | (1 << 1) | (1 << 2);
}
```

As you will see in the Timer0 interrupt initialization, Timer 0 is initialized with a prescaler of 1/256. If you choose a different value of the prescaler, you'd need to adjust the `OCR0A` accordingly.

5.2 Starting Interrupts in QF_onStartup()

QP-nano invokes the `QF_onStartup()` callback just before starting the event loop inside `QF_run()`. The `QF_onStartup()` function must start the interrupts configured earlier. In this BSP only the timer tick interrupt is started.

Listing 8 Configuring and enabling interrupts in the `QF_onStartup()` callback.

```
void QF_onStartup(void) {
    TCC0.CTRLA = 0x06;                  /* set TIMER C0 prescaler to CLK/256 */
    TCC0.CTRLB = 0x00;
    TCC0.CTRLC = 0x00;
    TCC0.CTRLD = 0x00;
                                          /* load 16-bit TIMER0 period */
    TCC0.PER   = (uint16_t)((F_CPU + (256 * BSP_TICKS_PER_SEC / 2))
                             / ((256 * BSP_TICKS_PER_SEC)));

    TCC0.INTCTRLA = 0x01;           /* enable interrupt on TIMER0 at level 1 */
  //TCC0.INTCTRLA = 0x02;           /* enable interrupt on TIMER0 at level 2 */
  //TCC0.INTCTRLA = 0x03;           /* enable interrupt on TIMER0 at level 3 */

    TCC0.INTCTRLA |= (0x03 << 2); /* enable error int. on TIMER0 at level 3 */
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

```
        TCC0.CTRLFSET = (1 << 2);                          /* update the timer */
        TCC0.CTRLFCLR = (1 << 2);

        TCC0.CTRLFSET = (1 << 3);                         /* restart the timer */
        TCC0.CTRLFCLR = (1 << 3);
    }
```

5.3 Assertion Handling Policy in Q_onAssert()

As described in Chapter 6 of [PSiCC2], all QP components use internally assertions to detect errors in the way application is using the QP services. You need to define how the application reacts in case of assertion failure by providing the callback function `Q_onAssert()`. 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_onAssert()` callback for AVR. The function simply locks all interrupts and enters a for-ever loop. This policy is only adequate for testing, but probably is not adequate for production release.

```
void Q_onAssert(char const Q_ROM * const Q_ROM_VAR file, int line) {
     (void)file;                                  /* avoid compiler warning */
     (void)line;                                  /* avoid compiler warning */
     QF_INT_LOCK();                                  /* lock all interrupts */
     LED_ON_ALL();                                           /* all LEDs on */
     for (;;) {       /* NOTE: replace the loop with reset for final version */
     }
}
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

6 Related Documents and References

Document	Location
[PSiCC2] "Practical UML Statecharts in C/C++, Second Edition", Miro Samek, Newnes, 2008	Available from most online book retailers, such as amazon.com. See also: http://www.state-machine.com/psicc2/
[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
[QL AN-PELICAN 08] "Application Note: PELICAN Crossing Application", Quantum Leaps, LLC, 2008	http://www.state-machine.com/doc/AN_PELICAN.pdf
[Atmel 08] Application Note "AVR1900: Getting started with ATxmega128A1 on STK600", Atmel 2008	www.atmel.com/dyn/resources/prod_documents/doc8107.pdf
[Atmel 09] "8-bit XMEGA A Microcontroller XMEGA A MANUAL", Atmel 09	http://www.atmel.com/dyn/resources/prod_documents/doc8077.pdf
[IAR 07] "IAR EWAVR Compiler Reference", IAR Systems	The PDF version of this document is included in the IAR EWAVR (EWAVR_CompilerReference.pdf).
[Samek+ 06b] "Build a Super Simple Tasker", Miro Samek and Robert Ward, Embedded Systems Design, July 2006.	http://www.embedded.com/showArticle.jhtml?articleID=190302110
[Samek 07a] "Using Low-Power Modes in Foreground/Background Systems", Miro Samek, Embedded System Design, October 2007	http://www.embedded.com/design/202103425

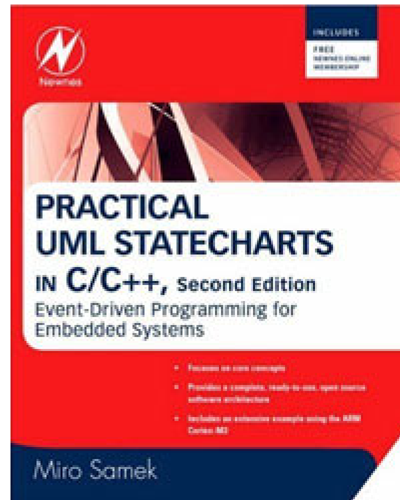
Copyright © Quantum Leaps, LLC. All Rights Reserved.

7 Contact Information

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

+1 866 450 LEAP (toll free, USA only)
+1 919 869-2998 (FAX)

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



"*Practical UML Statecharts in C/C++, Second Edition: Event Driven Programming for Embedded Systems*", by Miro Samek, Newnes, 2008

Copyright © Quantum Leaps, LLC. All Rights Reserved.