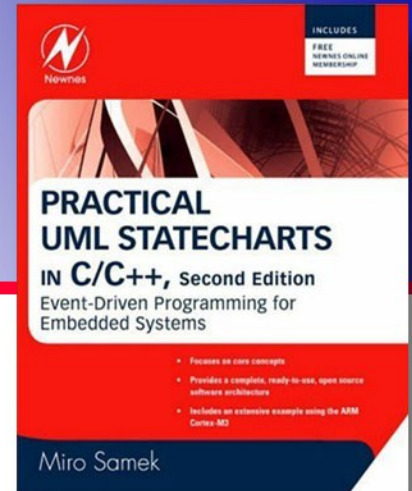




Quantum™ Leaps
innovating embedded systems



QDK™-nano Atmel AVR-IAR

Document Revision G
March 2012

Copyright © Quantum Leaps, LLC

www.quantum-leaps.com
www.sfate-machine.com

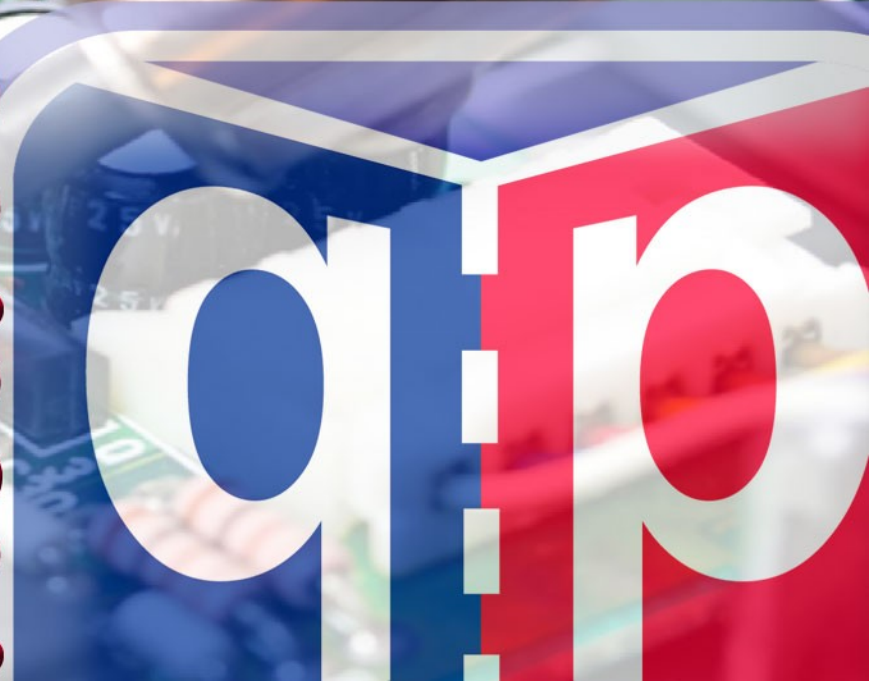


Table of Contents

1 Introduction	1
1.1 About QP-nano™	2
1.2 About QM™	3
1.3 What's Included in the QDK-nano?	3
1.4 Licensing QP-nano™	4
1.5 Licensing QM™	4
2 Getting Started	5
2.1 Installation	5
2.2 Building the Examples	6
2.3 Programming and Debugging the AVR Device	6
2.4 Setting Stack and Heap Size	7
3 Non-Preemptive Configuration of QP-nano	9
3.1 ISRs in the Non-preemptive "Vanilla" Configuration	10
3.2 QP Idle Loop Customization in QF_onIdle()	11
4 Preemptive Configuration with QK-nano	13
4.1 ISRs in the Preemptive Configuration with QK-nano	14
4.2 Idle Loop Customization in the QK Port	14
5 BSP for AVR	15
5.1 Board Initialization and the Timer Tick	15
5.2 Starting Interrupts in QF_onStartup()	15
5.3 Assertion Handling Policy in Q_onAssert()	15
6 Related Documents and References	16
7 Contact Information	17

AVR[®]

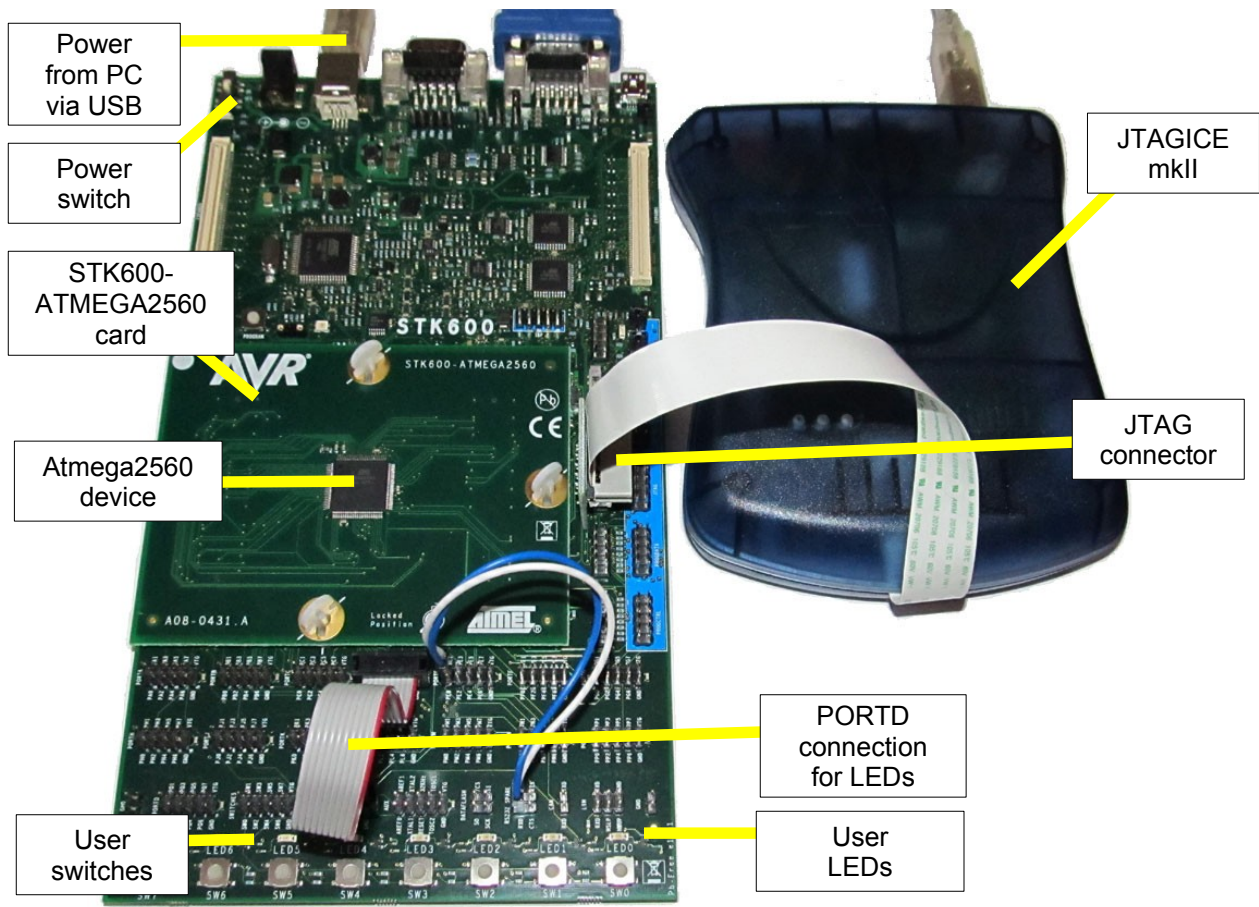
IAR SYSTEMS



1 Introduction

This QP-nano™ Development Kit (QDK-nano) describes how to use QP-nano™ state machine framework with the Atmel AVR and the IAR Embedded Workbench for AVR (EWIAR). The actual hardware/software used to test this QDK as shown in [Figure 1](#) and described below:

Figure 1: Atmel AVR Butterfly evaluation board connected to the AVR Dragon emulator, the RS232, external power, and external LEDs for PORTD.



The actual hardware/software used to test this QDK is described below:

- STK600 board with STK600-ATMEGA2560 routing board
- JTAGICE mkII debugger
- IAR Embedded Workbench for AVR **6.10**
- QP-nano **4.4.00** or higher and QM v **2.1.01** 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 on the ATmega2560 device with 8KB of RAM and 256KB of on-board flash. However, the described port should be applicable to most AVRmega devices (such as ATtiny and ATmega families) big enough to accommodate QP-nano.

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**. 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]

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

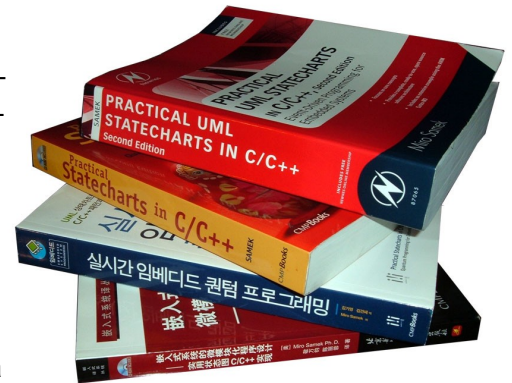
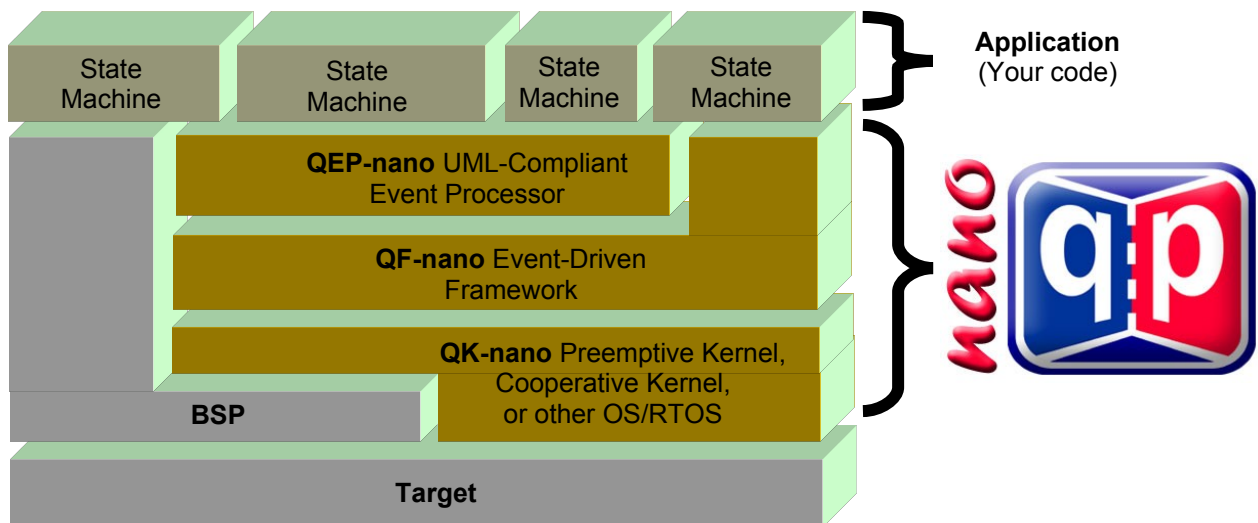


Figure 2: QP-nano™ components and their relationship with the target hardware, board support package (BSP), and the application comprised of state machines



1.2 About QM™

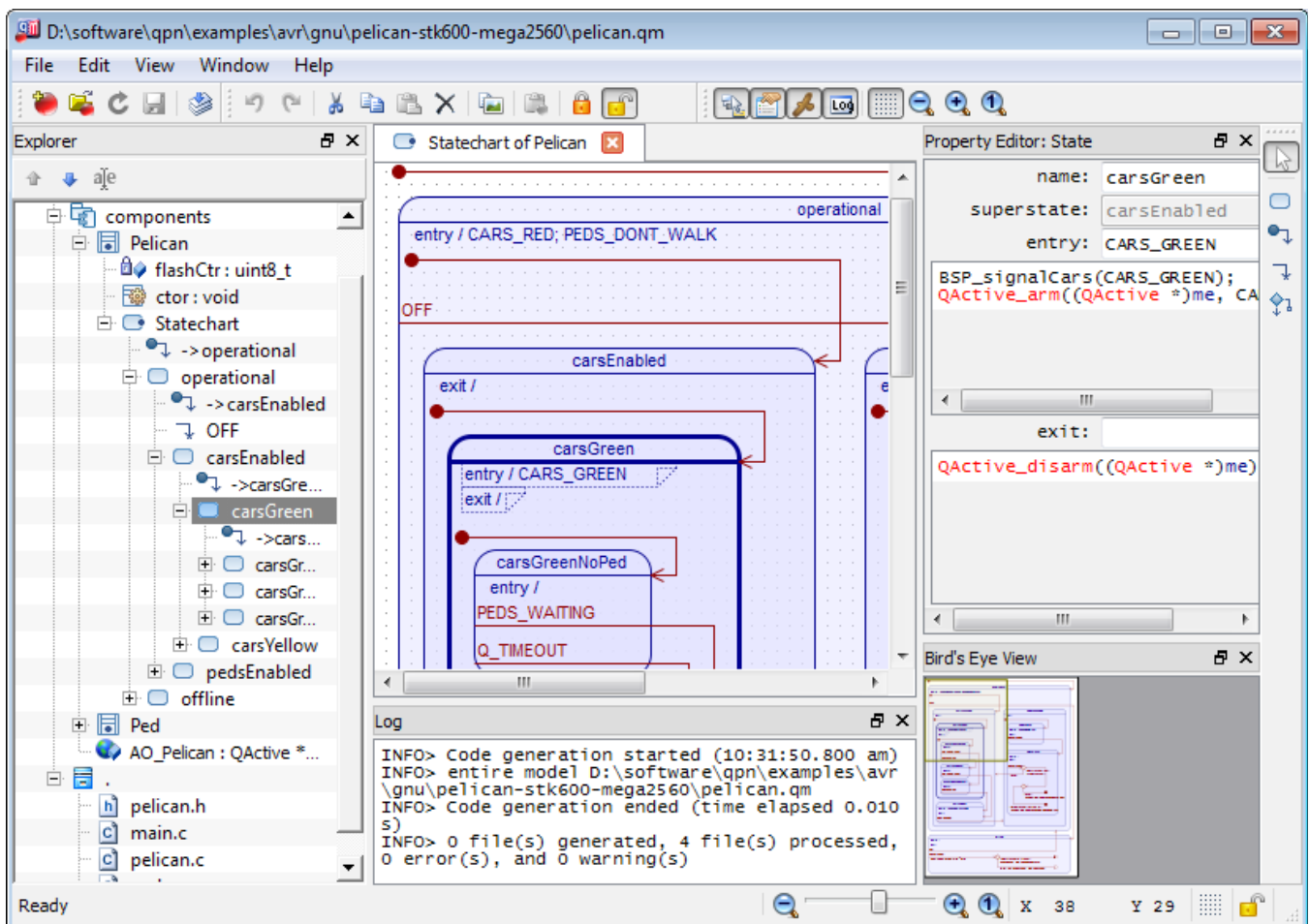
QM™ (QP™ Modeler) is a free, cross-platform, graphical UML modeling tool for designing and implementing real-time embedded applications based on the QP™ state machine frameworks. QM™ is available for Windows, Linux, and Mac OS X. QM™ provides intuitive diagramming environment for creating good looking hierarchical state machine diagrams and hierarchical outline of your entire application. QM™ eliminates coding errors by automatic generation of compact C or C++ code that is 100% traceable from your design. Please visit state-machine.com/qm for more information about QM™.



1.3 What's Included in the QDK-nano?

This QDK-nano provides the QP-nano port to AVR with the IAR 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], for cooperative and preemptive kernels, respectively.

Figure 3: The example model opened in the QM™ modeling tool



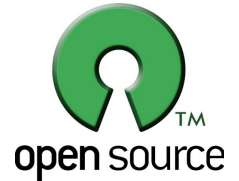
NOTE: The entire source code included with this QDK can be edited manually in a traditional code editor. However, significant parts of the code have been generated **automatically** by the QM™

modeling tool from the pelican .qm model file included in the QDK. The preferred way of developing QP™ applications is to make all the changes in the model and generate the code automatically..

1.4 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.

1.5 Licensing QM™

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



2 Getting Started

This section describes how to install, build, and use QDK-nano-AVR-IAR based on two examples.

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-IAR.zip). 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 code and the QDK-nano-AVR-IAR distribution. The highlighted directories and files are provided in the QDK-nano-AVR-IAR ZIP file.

```

qpn\                - QP-nano Root Directory
|
+-examples\        - QP-nano examples
| +-avr\           - examples for AVR
| | +-iar\         - examples compiled with the IAR AVR compiler
| | | +-pelican-stk600-atmega2560/ - PELICAN example for AVR STK600-ATMEGA2560
| | | | +-dbg\     - directory containing the debug build
| | | | +-rel\     - directory containing the release build
| | | | +-pelican.ewp - IAR project
| | | | +-pelican.eww - IAR workspace
| | | | +-pelican.qm - QM model for 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-atmega2560/ - PELICAN example for AVR STK600-ATMEGA2560
| | | | +-dbg\     - directory containing the debug build
| | | | +-rel\     - directory containing the release build
| | | | +-pelican-qk.ewp - IAR project
| | | | +-pelican-qk.eww - IAR workspace
| | | | +-pelican.qm - QM model for 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
|
+-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
  
```

```

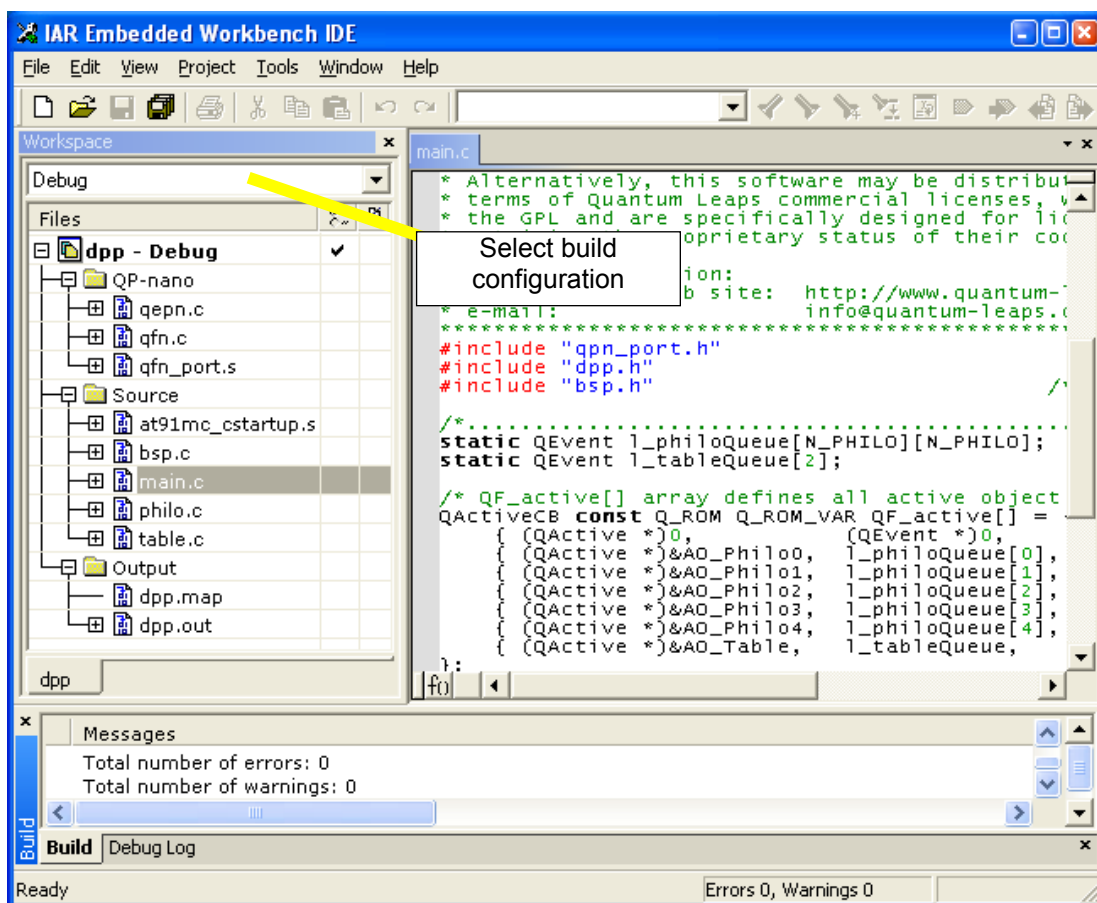
| +-qfn.h           - The platform-independent QF-nano header file
| +-qkn.h           - The platform-independent QK-nano header file
|
+-source/           - QP-nano source files
| +-qepn.c          - QEP-nano
| +-qfn.c           - QF-nano
| +-qkn.c           - QK-nano (required only in QK-nano configuration)

```

2.2 Building the Examples

The QDK-nano contains the IAR workspace is located in <qpn>\examples\avr\iar\pelican-stk600-atmega2560\pelican.eww for the “vanilla” version, and in <qpn>\examples\avr\iar\pelican-**qk**-stk600-atmega2560\pelican-**qk**.eww for the QK-nano version, respectively.

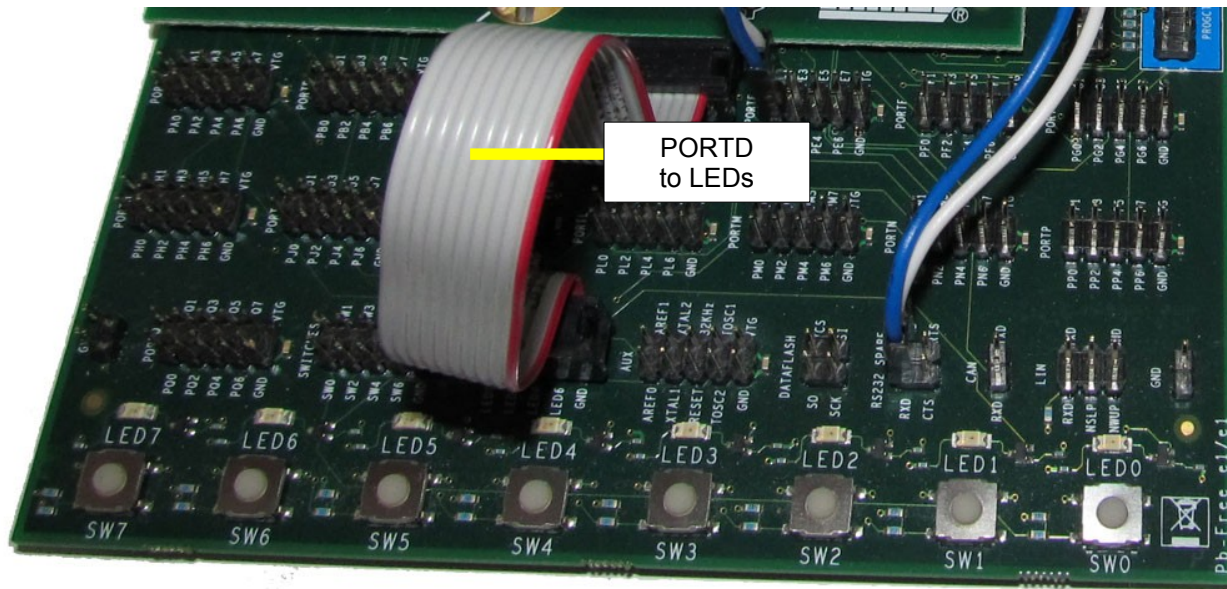
Figure 4: The PELICAN example project opened in IAR EWAVR



2.3 Programming and Debugging the AVR Device

Figure 5 shows how to make internal connections on the STK600 board to enable the external LEDs (see also Figure 1). The LEDs are connected to PORTD with the flat-band cable..

Figure 5: Connecting AVR Butterfly to AVR Dragon and external LEDs



After loading the PELICAN example to the board, the external LEDs connected to PORTD should start blinking. The LEDs are allocated as follows:

```
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.

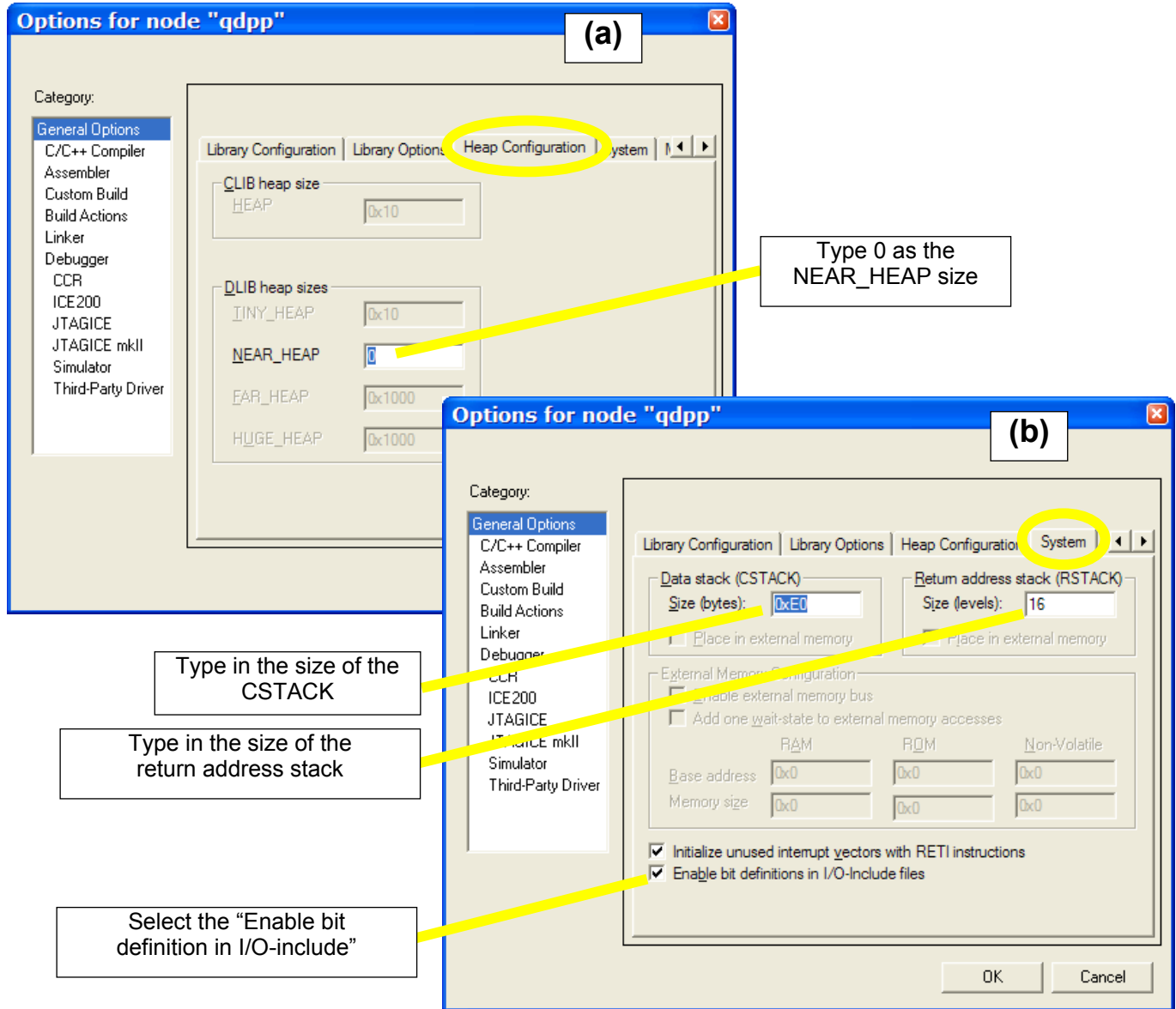
NOTE: The Release configuration puts the AVR core to sleep, so the idle LED(7) is not blinking. Actually, if you can watch it in the dark, you will see a very faint glow, corresponding to the system clock rate of 50 Hz.

NOTE: Also, the Release configuration is compiled without debug information.

2.4 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 6](#) shows selecting the heap size (zero) and stack size.

Figure 6: Setting the heap size to zero (a), and selecting the stack size (b)



3 Non-Preemptive Configuration of QP-nano

The example of using QP-nano with the cooperative “Vanilla” kernel is located in the directory: <qp>\-examples\arm\iar\pelican-stk600-atmega2560\. 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.

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 projects.

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 disabling policy for task level */
(6) #define QF_INT_DISABLE()    __disable_interrupt()
(7) #define QF_INT_ENABLE()     __enable_interrupt()

                                /* interrupt disabling policy for interrupt level */
(8) /* #define QF_ISR_NEST */    /* nesting of ISRs not 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 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.
- (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_DISABLE/QF_INT_ENABLE` define the task-level interrupt disabling policy for QP-nano (see Section 12.3.2 in Chapter 12 in [PSiCC2]).
- (8) This QP-nano port to AVR does **not** allow nesting of interrupts.

AVR does not provide any support for prioritizing interrupts in hardware. Therefore, enabling interrupts inside ISRs (the AVR hardware automatically disables interrupts upon entry to an ISR) is not advisable. Allowing interrupts to nest can lead to all sorts of priority inversions, including the pathological case of an interrupt preempting itself.

NOTE: This QP-nano port assumes that you never enable interrupts inside ISRs.

- (9) These header files are necessary for supporting interrupts in C.
- (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`.

3.1 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 = TIMER2_COMPA_vect
(2) __interrupt void tiemr2_ISR(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
- (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 disabled throughout the interrupt processing.

3.2 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 **disabled**, 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 DISABLED, see NOTE01 */  
  
                                /* toggle the LED number 7 on and then off, see NOTE02 */  
(2)     LED_ON(7);  
        LED_OFF(7);  
  
        #ifdef NDEBUG  
(3)     SMCR = (0 << SM0) | (1 << SE); /*idle sleep mode, adjust to your project */  
  
        /* 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, NOTE01 */  
  
(6)     SMCR = 0;                  /* clear the SE bit */  
        #else  
(7)     QF_INT_ENABLE();  
        #endif  
    }
```

- (1) The `QF_onIdle()` callback is always called with interrupts disabled 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 disabled, to the period of the LED turned on is constant and does not include any time spent in the ISRs.
- (3) The `SMCR` register is loaded with the desired sleep mode (idle mode in this case) and the Sleep Enable (SE) bit is set. Please note that the sleep mode is not active until the SLEEP command.
- (4) The interrupts are enabled with the intrinsic function, which emits the `SEI` instruction.
- (5) The sleep mode is activated with the intrinsic function, which emits the `SLEEP` instruction.

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) As recommended in the AVR datasheet, the `SMCR` register should be explicitly cleared upon the exit from the sleep mode.
 - (7) In the `DEBUG` configuration the interrupts are simply enabled.

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

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 disabling policy is the same, and the BSP is identical, except some small additions to the ISRs. The PELICAN example for the QK port is provided in the directory

```
<qp>\examples\avr\gnu\pelican-qk-stk600-atmega2560.
```

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 disabling policy for IAR compiler */
#define QF_INT_DISABLE()    __disable_interrupt()
#define QF_INT_ENABLE()    __enable_interrupt()

                                /* interrupt disabling policy for interrupt level */
/* #define QF_ISR_NEST */    /* nesting of ISRs not allowed */

                                /* interrupt entry/exit for QK-nano */
(1) #define QK_ISR_ENTRY()    ((void)0)
(2) #define QK_ISR_EXIT()    do { \
    uint8_t p = QK_schedPrio_(); \
    if (p != (uint8_t)0) { \
        QK_sched_(p); \
    } \
} while (0)

#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 */
(3) #include "qkn.h"         /* QK-nano platform-independent public interface */
```

- (1-2) The interrupt entry and exit macro for QK-nano are defined consistently with the interrupt nesting policy, which does not allow interrupt nesting.
- (3) The preemptive configuration of QP-nano is selected by including the `qkn.h` header file.

When interrupt nesting is *not* allowed (the macro `QF_ISR_NEST` is *not* defined in `qpn_port.h`), QK-nano allows for a simpler interrupt handling compared to the full-version QK. Specifically, when interrupts

cannot nest you don't need to increment the interrupt nesting counter (`QK_intNest_`) upon the ISR entry, and you don't need to decrement it upon the ISR exit. In fact, when the macro `QF_ISR_NEST` *not* defined, the `QK_intNest_` counter is not even available. This simplification is possible, because QP-nano uses special ISR-version of the event posting function `QActive_postISR()`, which does *not* call the QK-nano scheduler. Consequently, there is no need to prevent the synchronous preemption within ISRs.

4.1 ISRs in the Preemptive Configuration with QK-nano

As all preemptive kernels, QK-nano must be notified about interrupt entry and exit. You achieve this by means of the QK-nano macros `QK_ISR_ENTRY()` and `QK_ISR_EXIT()`, as shown in [Listing 6](#).

Listing 6: Time tick interrupt calling `QF_tick()` function to manage armed time events and QK-nano ISR entry/exit macros.

```
#pragma vector = TIMER2_COMPA_vect
__interrupt void tiemr2_ISR(void) {
    /* No need to clear the interrupt source since the Timer0 compare
     * interrupt is automatically cleared in hardware when the ISR runs.
     */

    QK_ISR_ENTRY();                               /* inform QK about entering the ISR */

    QF_tick();

    QK_ISR_EXIT();                               /* inform QK about exiting the ISR */
}
```

NOTE: This QP-nano port assumes that you never enable interrupts inside ISRs.

4.2 Idle Loop Customization in the QK Port

As described in Chapter 10 of [PSiCC2], the QK 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 enabled (which is in contrast to `QF_onIdle()` that is invoked with interrupts locked, see [Section 3.2](#)).

5 BSP for AVR

The Board Support Package (BSP) for AVR 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 devices. The most important step is initialization of Timer 0 to deliver the time tick interrupt at the desired rate (`BSP_TICKS_PER_SEC`):

```
void BSP_init(void) {
    DDRD = 0xFF;          /* All PORTD pins are outputs for LEDs */
    LED_OFF_ALL();       /* turn off all LEDs */
}
```

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 7: Configuring and enabling interrupts in the `QF_onStartup()` callback.

```
void QF_onStartup(void) {
    /* set Timer2 in CTC mode, 1/1024 prescaler, start the timer ticking */
    TCCR2A = ((1 << WGM21) | (0 << WGM20));
    TCCR2B = ((1 << CS22) | (1 << CS21) | (1 << CS20)); /* 1/2^10 */
    ASSR  &= ~(1 << AS2);
    TIMSK2 = (1 << OCIE2A); /* enable TIMER2 compare Interrupt */
    TCNT2 = 0;
    OCR2A = ((F_CPU / BSP_TICKS_PER_SEC / 1024) - 1); /* keep last */
}
```

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 disables 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) {
    QF_INT_DISABLE();
    LED_ON_ALL(); /* light up all LEDs */
    for (;;) { /* hang here in the for-ever loop */
    }
}
```

6 Related Documents and References

Document

[PSiCC2] "Practical UML Statecharts in C/C++, Second Edition", Miro Samek, Newnes, 2008

[QP-nano 08] "QP-nano Reference Manual", Quantum Leaps, LLC, 2008

[QL AN-Directory 07] "Application Note: QP Directory Structure", Quantum Leaps, LLC, 2007

[QL AN-PELICAN 08] "Application Note: PELICAN Crossing Application", Quantum Leaps, LLC, 2008

[Pardue 05] "C Programming for Microcontrollers", Joe Pardue, Smiley Micros, 2005

[Atmel 07] "ATmega169V, ATmega169 Data Sheet", Atmel

[IAR EWAVR] "IAR AVR C/C++ Compiler: Reference Guide", IAR Systems

[Samek+ 06b] "Build a Super Simple Tasker", Miro Samek and Robert Ward, Embedded Systems Design, July 2006.

[Samek 07a] "Using Low-Power Modes in Foreground/Background Systems", Miro Samek, Embedded System Design, October 2007

Location

Available from most online book retailers, such as amazon.com. See also: <http://www.state-machine.com/psicc2.htm>

<http://www.state-machine.com/doxygen/qpn/>

http://www.state-machine.com/doc/AN_QP_Directory_Structure.pdf

http://www.state-machine.com/doc/AN_PELICAN.pdf

<http://www.smileymicros.com>

http://www.atmel.com/dyn/resources/prod_documents/doc2514.pdf

The PDF version of this document is included in the IAR Embedded Workbench for AVR.

<http://www.embedded.com/showArticle.jhtml?articleID=190302110>

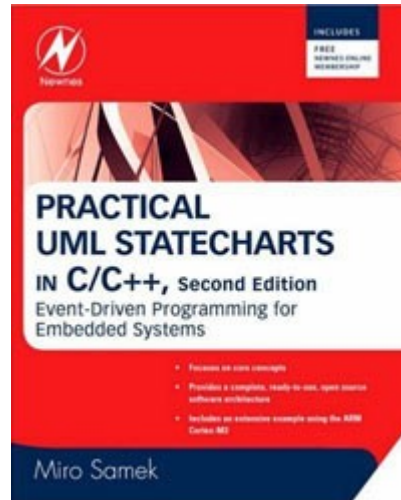
<http://www.embedded.com/design/202103425>

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

