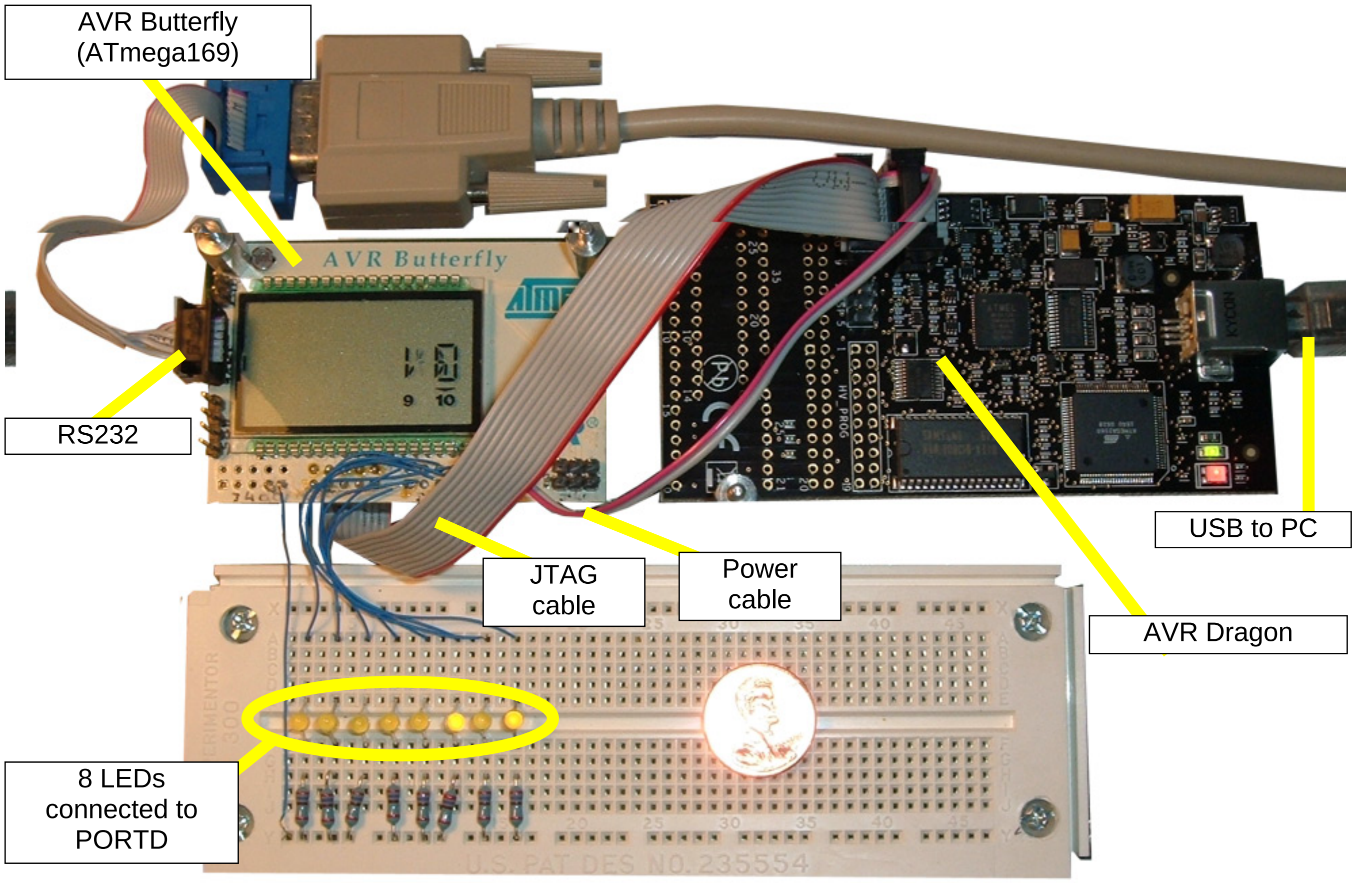
Table of Contents

1 Introduction

This **QP-nano™ Development Kit** (QDK-nano) describes how to use QP-nano™ event-driven platform with the Atmel AVR and the GNU compiler for AVR (WinWVR). The actual hardware/software used to test this QDK-nano 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.

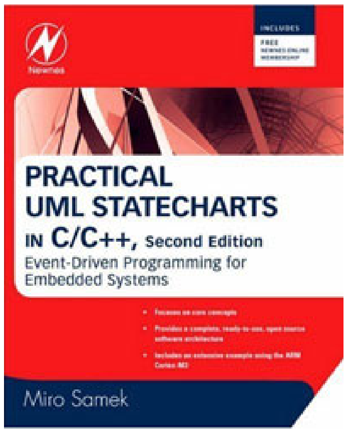


1. "AVR Butterfly" (containing ATmega169 MCU)
2. AVR Dragon programmer, emulator, and debugger
3. Atmel AVR Studio 4.14
4. AVR-GCC for windows version 4.1.1 (WinAVR 20070122)
5. QP-nano v4.0.01 or higher.

As shown in Figure 1, the Atmel's "AVR Butterfly" is connected to the AVR Dragon emulator via the 10-pin ribbon cable (the AVR Dragon connects to the PC USB port). The AVR Dragon can be used with various AVR devices. In Figure 1, the AVR Dragon is connected to the AVR Butterfly evaluation board, but most of the AVR devices could be used as well. This QDK has been tested with the ATmega169 device with 1KB of RAM and 16KB of onboard flash. However, the described port should be applicable to all AVR devices big enough to accommodate QP, that is, with RAM > 0.5KB, and ROM > 8KB.

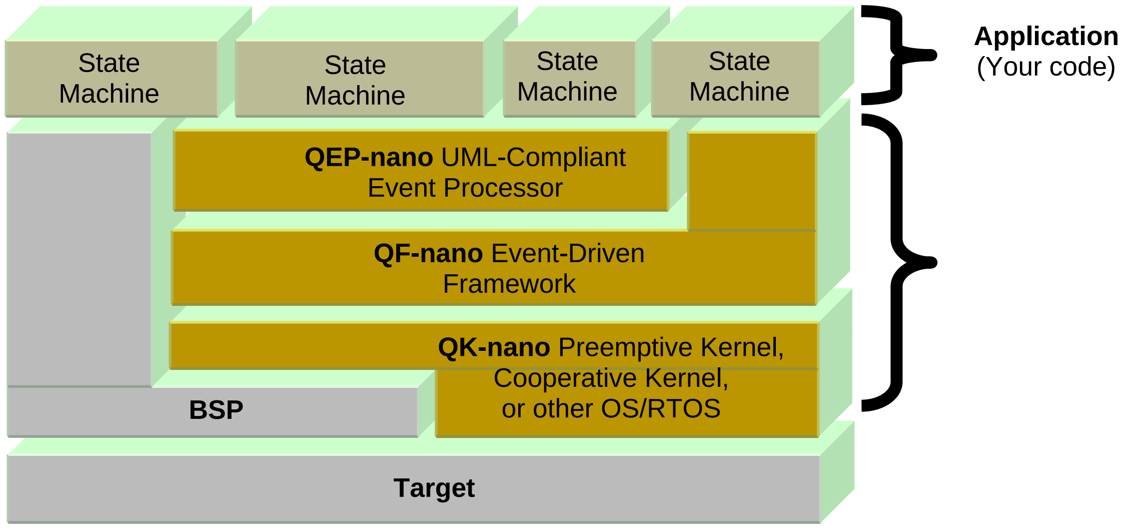
1.1 About QP-nano™

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.



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).

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


State Machine
State Machine
State Machine
State Machine
Application
(Your code)
QEP-nano UML-Compliant Event Processor
QF-nano Event-Driven Framework
QK-nano Preemptive Kernel, Cooperative Kernel, or other OS/RTOS
BSP
Target

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 with the GNU compiler, 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].

6. PELICAN crossing with the cooperative "Vanilla" kernel; and
7. PELICAN crossing with the preemptive run-to-completion QK-nano kernel.

NOTE: Even though this QDK-nano is based on a specific development board (AVR Butterfly in this case), the most important parts of the QP-nano ports are applicable to all AVR-based MCUs. In particular, the QP-nano port to the cooperative "Vanilla" kernel, as well as the QP-nano port to the preemptive QK-nano kernel are generic and should not need to change for other AVR systems.

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-GNU 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-GNU_<ver>.zip`, where `<ver>` stands for a specific QDK-nano version, such as 4.0.01). 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>.

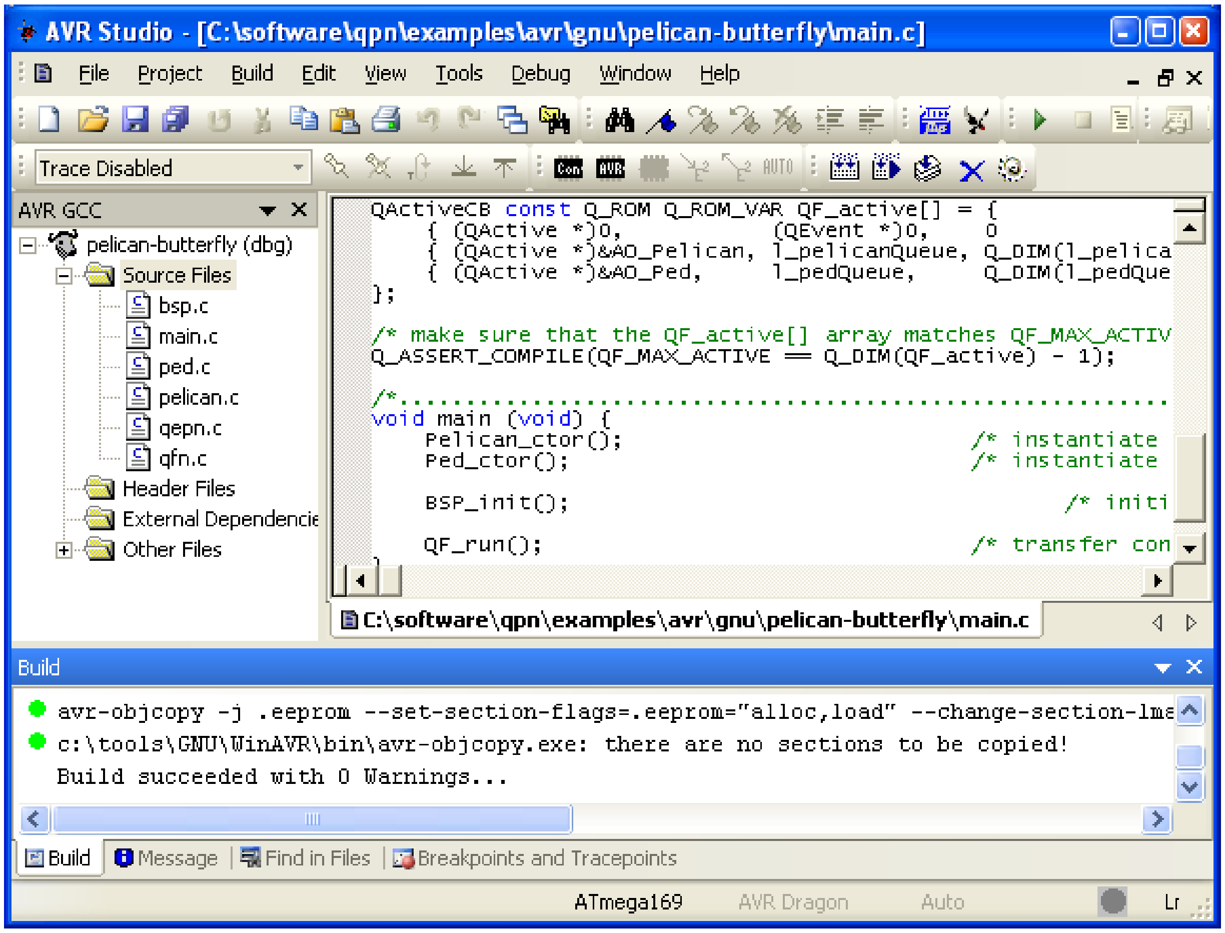
Listing 1 Directories and files after installing QP-nano baseline code and the QDK-nano-AVR-GNU distribution. The highlighted directories and files are provided in the QDK-nano-AVR-GNU ZIP file.

```
<qpn>/                     - QP-nano Root Directory
  |
  +-examples/              - QP-nano examples
  | +-avr\                 - examples for AVR
  | | +-gnu\               - examples compiled with the GNU (WinAVR) compiler
  | | | +-pelican-butterfly\  - PELICAN example for Butterfly (non-preemptive)
  | | | | +-dbg\           - directory containing the debug build
  | | | | | +-pelican-butterfly.elf - image of the application
  | | | | +-rel\           - directory containing the release build
  | | | | | +-pelican-butterfly.hex - image of the application
  | | | | | +-. . .
  | | | | +-Makefile       - Makefile for building the example
  | | | | +-dpp-butterfly.aps – AVRStudio project file to build the DPP 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-butterfly\  - PELICAN example for Butterfly (preemptive QK)
  | | | | +-dbg\           - directory containing the debug build
  | | | | | +-pelican-qk-butterfly.elf - image of the application
  | | | | +-rel\           - directory containing the release build
  | | | | | +-pelican-qk-butterfly.hex - image of the application
  | | | | +-Makefile       - Makefile for building the example
  | | | | +-dpp-qk-butterfly.aps – AVRStudio project to build the DPP 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

This QDK-nano-AVR-GNU offers two way of building the examples. The first way is based on the command-line GNU-make utility included in the WinAVR distribution.

Figure 3 AVR Studio IDE used for developing and debugging C-code with WinAVR.



```
QActiveCB const Q_ROM Q_ROM_VAR QF_active[] = {
    { (QActive *)0,            (QEvent *)0,      0
    { (QActive *)&AO_Pelican, l_pelicanQueue, Q_DIM(l_pelica
    { (QActive *)&AO_Ped,      l_pedQueue,      Q_DIM(l_pedQue
};

/* make sure that the QF_active[] array matches QF_MAX_ACTIV
Q_ASSERT_COMPILE(QF_MAX_ACTIVE == Q_DIM(QF_active) - 1);

/*..............................................................
void main (void) {
    Pelican_ctor();                              /* instantiate
    Ped_ctor();                                  /* instantiate

    BSP_init();                                        /* initi

    QF_run();                                    /* transfer con
```

```
avr-objcopy -j .eeprom --set-section-flags=.eeprom="alloc,load" --change-section-lma
c:\tools\GNU\WinAVR\bin\avr-objcopy.exe: there are no sections to be copied!
Build succeeded with 0 Warnings...
```

The QDK contains the `Makefile` to build the examples, which is located in `<qpn>\examples\avr\gnu\pelican-butterfly\Makefile` for the “vanilla” version, and in `<qpn>\examples\avr\gnu\pelican-`**`qk`**`-butterfly\Makefile` for the QK-nano version, respectively. The `Makefile` supports two build configurations: Debug and Release (`make`, `make rel`).

The second way of building the examples is based on the Atmel AVR Studio IDE. The QDK-nano contains the AVR Studio project files located in `<qpn>\examples\avr\gnu\pelican-butterfly\pelican-butterfly.aps` for the “vanilla” version, and in `<qpn>\examples\avr\gnu\pelican-`**`qk`**`-butterfly\pelican-qk-butterfly.aps` for the QK-nano version, respectively.

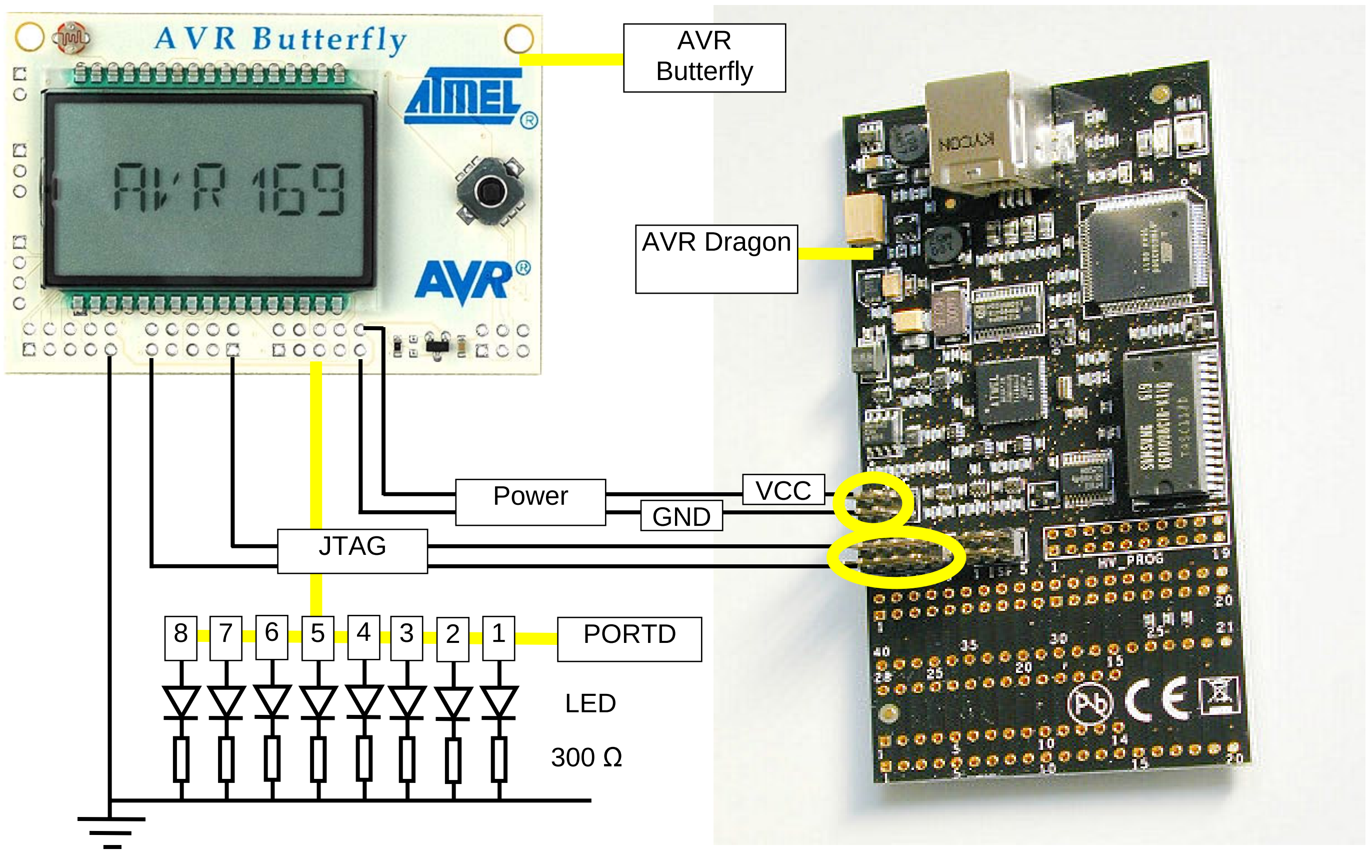
As shown in Figure 3, AVR Studio 4 supports developing C-code with the WinAVR toolset. The IDE supports generating WinAVR projects, as well as editing and compiling.

2.3 Programming and Debugging the AVR Device

Figure 4Figure 4 shows how to connect AVR Butterfly to the AVR Dragon JTAG debugger and the external LEDs, which is the setup used to test this QDK-nano (see also Figure 1). The LEDs are connected to PORTD, whereas the labels in Figure 4 correspond to pin numbers of the PORTD connector on the AVR Butterfly board. In the setup shown in Figure 4 the LEDs are connected to ground via 300 Ω resistors. The short leg of the LED should be on the resistor’s side (see also the book “C Programming for Microcontrollers” [Pardue 05]).

Power is also provided through the PORTD connector, pin 9 connected to Ground and pin 10 to VCC of AVR Dragon power connector (see Figure 4 and Figure 1).

Figure 4 Connecting AVR Butterfly to AVR Dragon and external LEDs

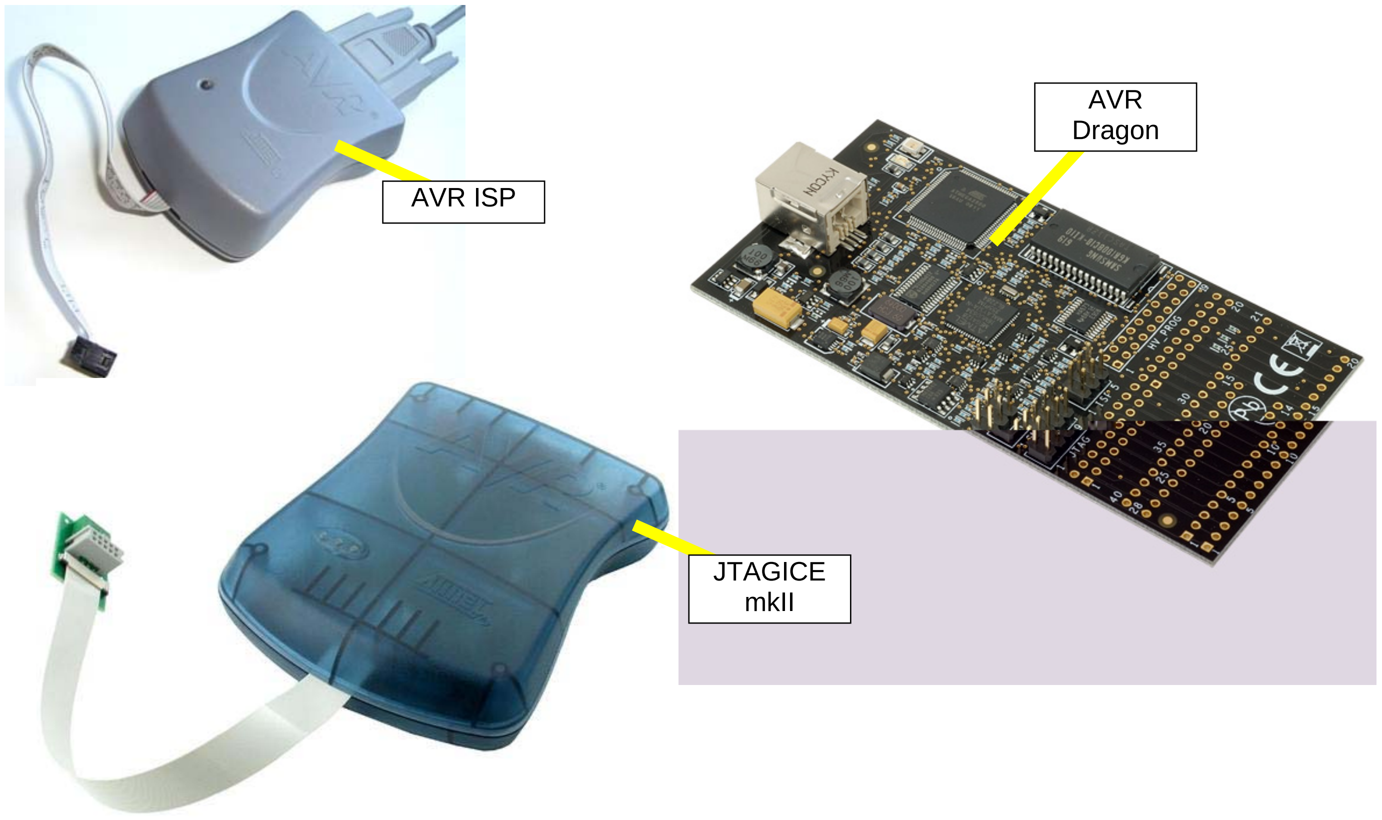


To download the code to the AVR device you can use the AVR Studio IDE (see Figure 3), which is available for a free download from the Atmel website (www.atmel.com). This QDK-nano uses AVR Dragon JTAG debugger, but the QDK can also work with JTAGICE mkII or even AVR ISP (just programming, no debugging).

Loading and executing the examples on the AVR device depends on which hardware debugger you're using to connect to the AVR target. If you're using JTAGICE mkII or AVR Dragon (see Figure 5), you can use AVR Studio to directly download and debug the application in the target AVR device. If you are using AVR ISP, you can only program the device, but you cannot run the debugger.

NOTE: The Release configuration is compiled without debug information. Consequently, you can only program the HEX file (from the rel\ directory) to the target using the AVR Studio, Tools | AVR Prog menu.

Figure 5 AVR ISP, JTAGICE mkII, and AVR Dragon



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:

```
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 PELICAN example does **not** use the LCD of the AVR Butterfly (although you might see some activity on the LCD, because it’s connected to the same PORTD as the external LEDs used in this example).

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\arm\gnu\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.

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 systems.

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

```
     #ifndef qpn_port_h
     #define qpn_port_h

 (1) #define Q_ROM                   PROGMEM
 (2) #define Q_ROM_BYTE(rom_var_)    pgm_read_byte_near(&(rom_var_))
 (3) #define Q_ROM_PTR(rom_var_)     pgm_read_word_near(&(rom_var_))

 (4) #define Q_NFSM
 (5) #define Q_PARAM_SIZE            1
 (6) #define QF_TIMEEVT_CTR_SIZE     2

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

                                    /* interrupt locking policy for IAR compiler */
 (8) #define QF_INT_LOCK()           cli()
 (9) #define QF_INT_UNLOCK()         sei()

                                 /* interrupt locking policy for interrupt level */
(10) /* #define QF_ISR_NEST */                     /* nesting of ISRs not allowed */

(11) #include <avr\io.h>
     #include <avr\interrupt.h>                                  /* cli()/sei() */
     #include <avr\pgmspace.h> /* accessing data in the program memory (PROGMEM) */

(12) #include <stdint.h>    /* Exact-width integer types. WG14/N843 C99 Standard */
(13) #include "qepn.h"         /* QEP-nano platform-independent public interface */
(14) #include "qfn.h"           /* QF-nano platform-independent public interface */

     #endif                                                       /* qpn_port_h */
```

(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 “PROGMEM” macro in the WinAVR compiler.

(2) The macro `Q_ROM_BYTE()` encapsulates a custom mechanism of retrieving a data byte from the given ROM address. WinAVR cannot generate code for accessing data allocated in the program space (ROM), even though the compiler can allocate constants in ROM. The workaround for WinAVR is to explicitly add custom assembly code to access data allocated in the program space.

(3) The macro `Q_ROM_PTR()` encapsulates a custom mechanism of retrieving a data pointer from the given ROM address. WinAVR cannot generate code for accessing data allocated in the program space (ROM), even though the compiler can allocate constants in ROM. The workaround for WinAVR is to explicitly add custom assembly code to access data allocated in the program space.

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

(5) 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.

(6) 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.

(7) 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`.

(8-9) 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]).

(10) This QP-nano port to AVR does not allow nesting of interrupts.

AVR does not provide any support for prioritizing interrupts in hardware. Therefore, unlocking interrupts inside ISRs (the AVR hardware automatically locks 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 unlock interrupts inside ISRs.

(11) These header files are necessary for supporting interrupts in C an well as accessing the PROGMEM space.

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

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

(14) 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 GNU-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) SIGNAL(SIG_OUTPUT_COMPARE0) {
```

```
            /* No need to clear the interrupt source since the Timer0 compare
            * interrupt is automatically cleard in hardware when the ISR runs.
            */
(2)         QF_tick();
        }
```

(1) The definition of the interrupt function must begin with the `SIGNAL(SIG_???)` macro.

(2) 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.

3.2 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 **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

(3)     SMCR = (0 << SM0) | (1 << SE);/*idle sleep mode, adjust to your project */

          /* never separate the following two assembly instructions, see NOTE03 */
(4)     __asm__ __volatile__ ("sei" "\n\t" :: );
(5)     __asm__ __volatile__ ("sleep" "\n\t" :: );

(6)     SMCR = 0;                                              /* clear the SE bit */
    #else
(7)     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 `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 unlocked with the `SEI` instruction.

(5) The sleep mode is activated with 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 unlocked.

NOTE: Every path through `QF_onIdle()` callback function must ultimately unlock 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 locking/unlocking 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 `<qpn>\examples\avr\gnu\pelican-qk-butterfly`.

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                     PROGMEM
    #define Q_ROM_BYTE(rom_var_)      pgm_read_byte_near(&(rom_var_))
    #define Q_ROM_PTR(rom_var_)       pgm_read_word_near(&(rom_var_))

    #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()             cli()
    #define QF_INT_UNLOCK()           sei()

                                  /* interrupt locking 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()          QK_SCHEDULE_()

    #include <avr\io.h>
    #include <avr\interrupt.h>                                   /* cli()/sei() */
    #include <avr\pgmspace.h> /* accessing data in the program memory (PROGMEM) */

    #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.

```
SIGNAL(SIG_OUTPUT_COMPARE0) {
    QK_ISR_ENTRY();                     /* inform QK about entering the ISR */
    /* No need to clear the interrupt source since the Timer0 compare
    * interrupt is automatically cleard in hardware when the ISR runs.
    */
    QF_tick();
    QK_ISR_EXIT();                       /* inform QK about exiting the ISR */
}
```

NOTE: This QP-nano port assumes that you never unlock 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 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 AT91SAM7 MCU. Other ARM-based embedded microcontrollers (e.g., LPC chips from NXP) handle the power-saving mode very similarly.

Listing 7 QK_onIdle() callback for the AT91SAM7 MCU.

```
__ramfunc void QK_onIdle(void) {
#ifdef NDEBUG     /* only if not debugging (power saving hinders debugging) */
    AT91C_BASE_PMC->PMC_SCDR = 1;/* Power-Management: disable the CPU clock */
    /* NOTE: an interrupt starts the CPU clock again */
#endif
}
```

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 */
    PORTD = 0x00;                                        /* trun off all LEDs */

                                             /* set the output compare value */
    OCR0A  = ((F_CPU / BSP_TICKS_PER_SEC / 1024) - 1);

    if (QS_INIT((void *)0) == 0) {    /* initialize the QS software tracing */
        Q_ERROR();
    }
}
```

As you will see in the Timer0 interrupt initialization (in the file `isr.c`), Timer 0 is initialized with a prescaler of 1/1024. 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. Please note that `QF_onStartup()` must also enable the global interrupt flag in the SR, but this is done with the `sei()` macro, rather than the `QF_INT_UNLOCK()` macro. The `QF_INT_UNLOCK()` macro is inappropriate, because it requires the interrupt "lock key", which is not available at this time (because `QF_onStartup()` never locks the interrupts).

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

```
void QF_onStartup(void) {
    cli();                    /* make sure that all interrupts are disabled */

    /* set Timer0 in CTC mode, 1/1024 prescaler, start the timer ticking */
    TCCR0A = ((1 << WGM01) | (0 << WGM00) | (5 << CS00));
    TIMSK0 = (1 << OCIE0A);                /* Enable TIMER0 compare Interrupt */

    sei();                      /* make sure that all interrupts are enabled */
}
```

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

6 Related Documents and References

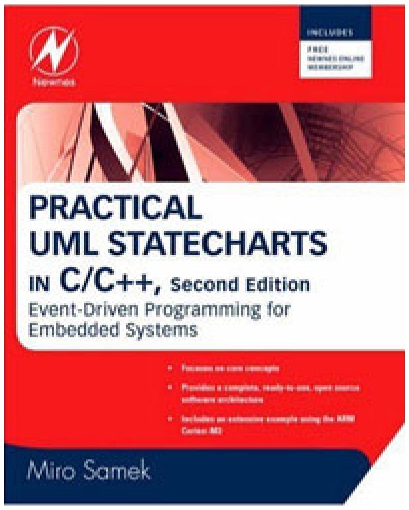
Document	Location
[PSiCC2] "Practical UML Statecharts in C/C++, Second Edition", Miro Samek, Newnes, 2008	Available from most online book retailers, such as amazon.com. See also: http://www.state-machine.com/psicc2.htm
[QP-nano 08] "QP-nano Reference Manual", Quantum Leaps, LLC, 2008	http://www.state-machine.com/doxygen/qpn/
[QL AN-Directory 07] "Application Note: QP Directory Structure", Quantum Leaps, LLC, 2007	http://www.state-machine.com/doc/AN_QP_Directory_Structure.pdf
[QL AN-PELICAN 08] "Application Note: PELICAN Crossing Application", Quantum Leaps, LLC, 2008	http://www.state-machine.com/doc/AN_PELICAN.pdf
[Pardue 05] "C Programming for Microcontrollers", Joe Pardue, Smiley Micros, 2005	http://www.smileymicros.com
[Atmel 07] "ATmega169V, ATmega169 Data Sheet", Atmel	http://www.atmel.com/dyn/resources/-prod_documents/doc2514.pdf
[WinAVR 07] "GNU AVR C/C++ Compiler: Reference Guide", GNU Systems	The PDF version of this document is included in the GNU Embedded Workbench for AVR.
[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

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
Miro Samek

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