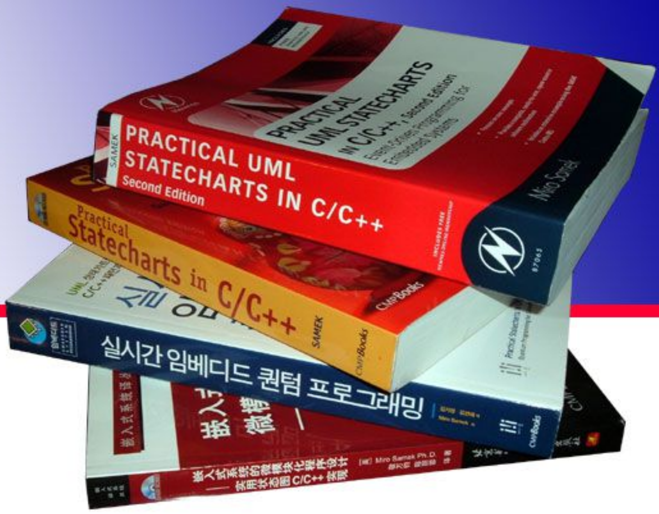
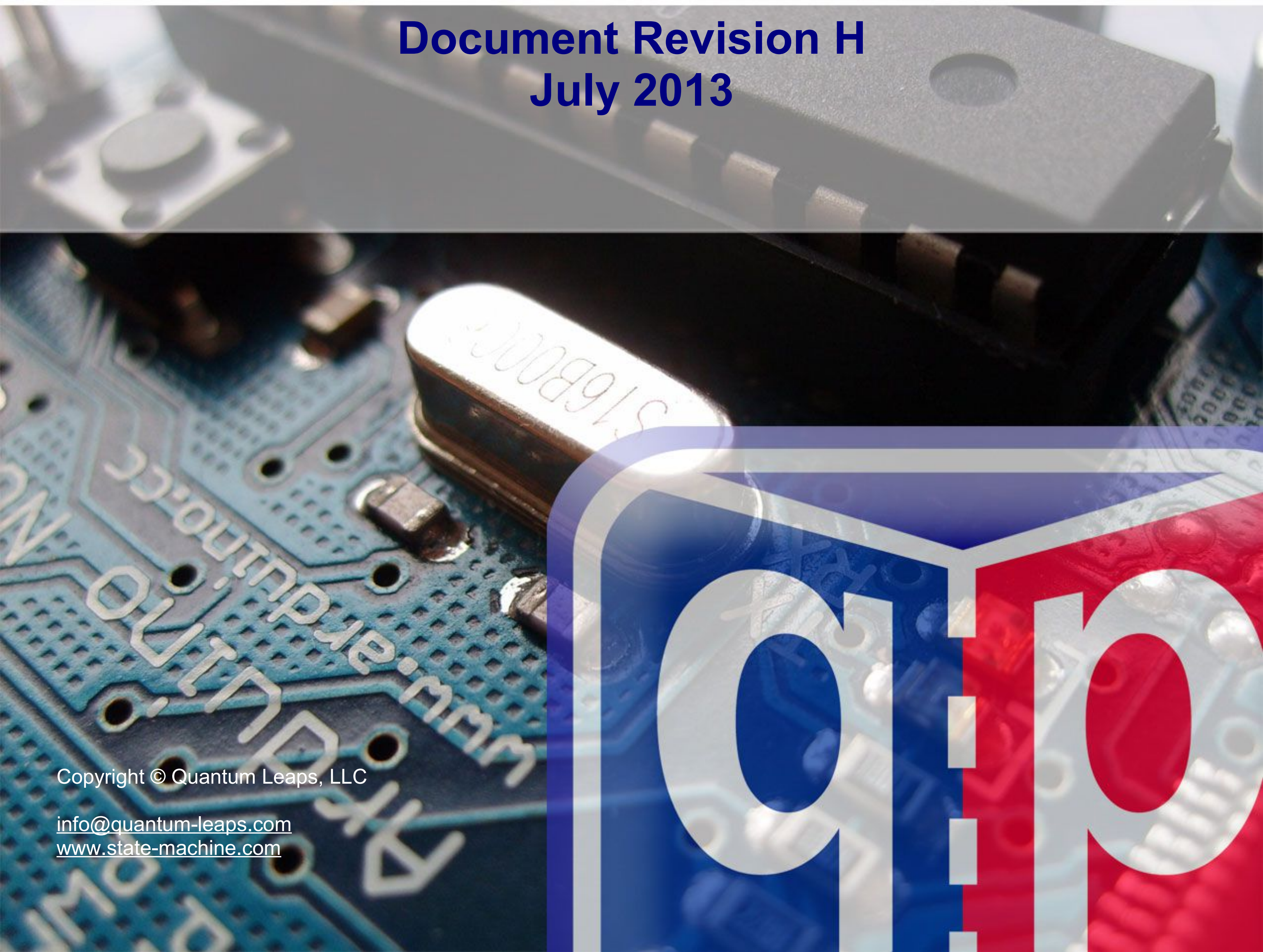
quantum® LeaPs
innovating embedded systems



Application Note
Event-Driven Arduino™ Programming with QP™



Document Revision H
July 2013

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

Table of Contents

1 Introduction

This document describes how to apply the **event-driven programming** paradigm with modern **state machines** to develop software for Arduino™ **graphically**. Specifically, you will learn how to build responsive, robust, and power-efficient Arduino programs with the open source QP™/C++ active object framework, which is like a modern **real-time operating system** (RTOS) specifically designed for executing event-driven, encapsulated state machines (active objects). You will also see how to take Arduino programming to the next level by using the the free QM™ modeling tool to **generate** Arduino code **automatically** from state diagrams.

NOTE: The QM modeling tool together with the build script provided in the QP/Arduino disribution allow you to build and upload the Arduino sketches entirely from the QM tool. However, the provided examples remain **compabilbe with the standard Arduino IDE**.

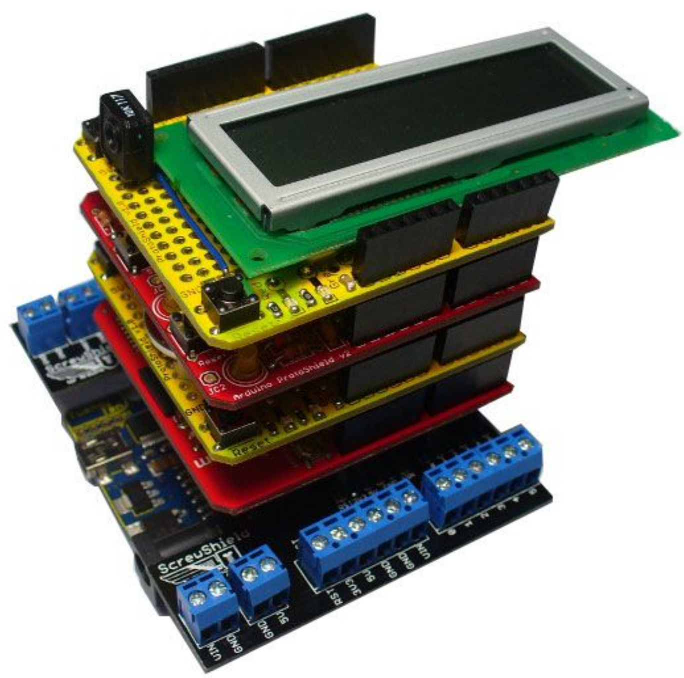
1.1 About Arduino™

Arduino™ (see www.arduino.cc) is an open-source electronics prototyping platform, designed to make digital electronics more accessible to non-specialists in multidisciplinary projects. The hardware consists of a simple Arduino printed circuit board with an Atmel AVR microcontroller and standardized pin-headers for extensibility. The Arduino microcontroller is programmed using the C++ language (with some simplifications, modifications, and Arduino-specific libraries), and a Java-based integrated development environment (called Processing) that runs on a desktop computer (Windows, Linux, or Mac).

Arduino boards can be purchased pre-assembled at relatively low cost ($20-$50). Alternatively, hardware design information is freely available for those who would like to assemble an Arduino board by themselves.

Arduino microcontroller boards are extensible by means of Arduino "shields", which are printed circuit boards that sit on top of an Arduino microcontroller board, and plug into the standardized pin-headers (see Figure 1). Many such Arduino shields are available for connectivity (USB, CAN, Ethernet, wireless, etc.), GPS, motor control, robotics, and many other functions. A steadily growing list of Arduino shields is maintained at shieldlist.org.

Figure 1: A stack of Arduino™ shields



NOTE: This document assumes that you have a basic familiarity with the Arduino environment and you know how to write and run simple programs for Arduino.

1.2 Event-driven programming with Arduino™

Traditionally, Arduino programs are written in a **sequential** manner. Whenever an Arduino program needs to synchronize with some external event, such as a button press, arrival of a character through the serial port, or a time delay, it explicitly *waits in-line* for the occurrence of the event. Waiting "in-line" means that the Arduino processor spends all of its cycles constantly checking for some condition in a tight loop (called the polling loop). For example, in almost every Arduino program you see many polling loops like the code snippet below, or function calls, like `delay()` that contain implicit polling loops inside:

Listing 1: Sequential programming example

```
void loop() {
    while (digitalRead(buttonPin) != HIGH) ; // wait for the button press
    . . . // process the button press
    while (Serial.available() == 0) ; // wait for a character from the serial port
    char ch = Serial.read();  // obtain the character
    . . . // process the character
    delay(1000); // implicit polling loop (wait for 1000ms)
    . . . // process the timeout, e.g., switch an LED on
}
```

Although this approach is functional in many situations, it doesn't work very well when there are multiple possible sources of events whose arrival times and order you cannot predict and where it is important to handle the events in a *timely* manner. The fundamental problem is that while a sequential program is waiting for one kind of event (e.g., a button press), it is not doing any other work and is **not responsive** to other events (e.g., characters from the serial port).

Another big problem with the sequential program structure is wastefulness in terms of **power dissipation**. Regardless of how much or how little actual work is being done, the Arduino processor is always running at top speed, which drains the battery quickly and prevents you from making truly long-lasting battery-powered devices.

NOTE: If you intend to use Arduino in a battery operated device, you should seriously consider the event-driven programming option. Please also see the upcoming Section 2.6.

For these and other reasons experienced programmers turn to the long-know design strategy called **event-driven programming**, which requires a distinctly different way of thinking than conventional sequential programs. All event-driven programs are naturally divided into the *application*, which actually handles the events, and the supervisory event-driven infrastructure (**framework**), which waits for events and dispatches them to the application. The control resides in the event-driven framework, so from the application standpoint, the control is **inverted** compared to a traditional sequential program.

Listing 2: The simplest event-driven program structure. The highlighted code conceptually belongs to the event-driven framework.

```
void loop() {
    if (event1())          // event1 occurred?
        event1Handler(); // process event1
    if (event2())          // event2 occurred?
        event2Handler(); // process event2
    . . .                  // handle other events
}
```

An event-driven framework can be very simple. In fact, many projects in the Arduino Playground / Tutorials and Resources / Protothreading, Timing & Millis section provide examples of rudimentary event-driven frameworks. The general structure of all these rudimentary frameworks is shown in Listing 2.

The framework in this case consists of the main Arduino loop and the `if` statements that check for events. Events are effectively polled during each pass through the main loop, but the main loop does **not** get into tight polling sub-loops. Calls to functions that poll internally (like `delay()`) are **not** allowed, because they would slow down the main loop and defeat the main purpose of event-driven programming (responsiveness). The application in this case consists of all the event handler functions (`event1Handler()`, `event2Handler()`, etc.). Again, the critical difference from sequential programming here is that the event handler functions are **not** allowed to poll for events, but must consist essentially of linear code that quickly **returns** control to the framework after handling each event.

This arrangement allows the event-driven program to remain **responsive** to all events all the time, but it is also the biggest challenge of the event-driven programming style, because the application (the event handler functions) must be designed such that for each new event the corresponding event handler can pick up where it left off for the last event. (A sequential program has much less of this problem, because it can hang on in tight polling loops around certain places in the code and process the events in the contexts just following the polling loops. This arrangement allows a sequential program to move naturally from one event to the next.)

Unfortunately, the just described main challenge of event-driven programming often leads to **"spaghetti" code**. The event handler functions start off pretty simple, but then `if`-s and `else`-s must be added inside the handler functions to handle the **context** properly. For example, if you design a vending machine, you cannot process the "dispense product" button-press event until the full payment has been collected. This means that somewhere inside the `dispenseProductButtonPressHandler()` function you need an `if`-statement that tests the payment status based on some global variable, which is set in the event handler function for payment events. Conversely, the payment status variable must be changed after dispensing the product or you will allow dispensing products without collecting subsequent payments. Hopefully you see how this design quickly leads to dozens of global variables and hundreds of tests (`if`-s and `else`-s) spread across the event handler functions, until no human being has an idea what exactly happens for any given event, because the event-handler code resembles a bowl of tangled spaghetti. An example of spaghetti code just starting to develop is the Stopwatch project available from the Arduino Playground.

Luckily, generations of programmers before you have discovered an effective way of solving the "spaghetti" code problem. The solution is based on the concept of a **state machine**, or actually a set of collaborating state machines that preserve the context from one event to the next using the concept of *state*. This document describes this most advanced and powerful way of combining the event-driven programming paradigm with modern state machines.

1.3 The structure of the QP/C++ event-driven framework for Arduino™

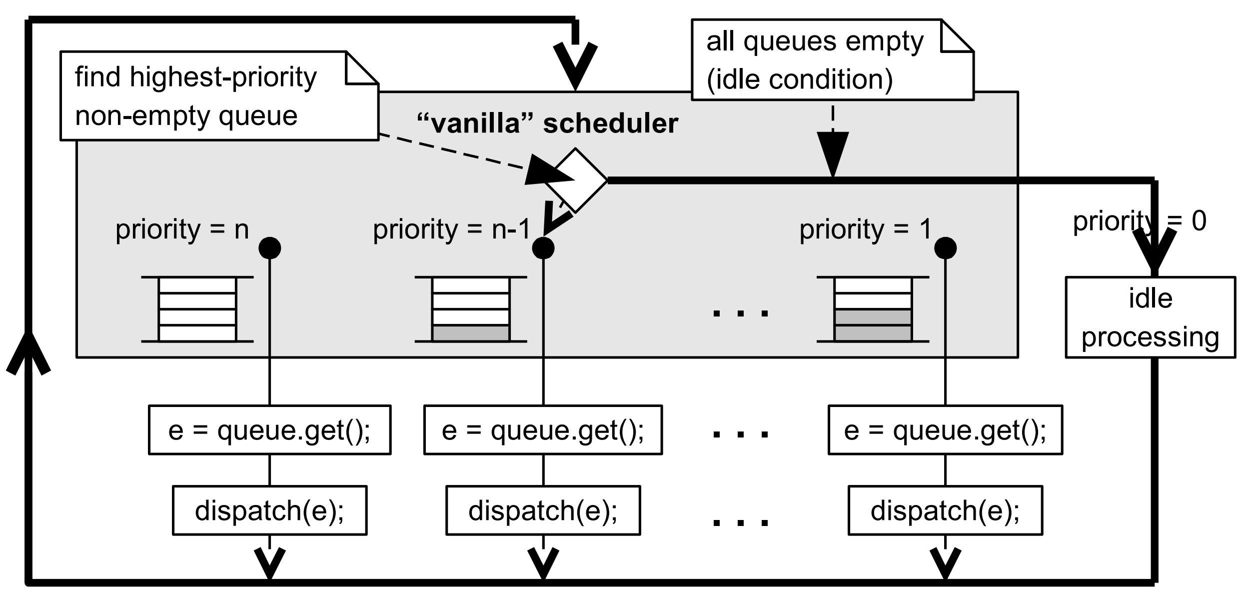
The rudimentary examples of event-driven programs currently available from the Arduino Playground are very simple, but they don't provide a true event-driven programming environment for a number of reasons. First, the simple frameworks don't perform *queuing* of events, so events can get lost if an event happens more than once before the main loop comes around to check for this event or when an event is *generated* in the loop. Second, the primitive event-driven frameworks have no safeguards against corruption of the global data shared among the event-handlers by the interrupt service routines (ISRs), which can preempt the main loop at any time. And finally, the simple frameworks are not suitable for executing state machines due to the early filtering by event-type, which does not leave room for state machine(s) to make decisions based on the internal *state*.

Figure 2 shows the structure of the QP framework for Arduino, which does provide all the essential elements for safe and efficient event-driven programming. As usual, the software is structured in an endless event loop. The most important element of the design is the presence of multiple **event queues** with a unique priority and a **state machine** assigned to each queue. The queues are constantly monitored by the *"vanilla" scheduler*, which by every pass through the loop picks up the highest-priority not-empty queue. After finding the queue, the scheduler extracts the event from the queue and sends it to the state machine associated with this queue, which is called *dispatching* of an event to the state machine.

NOTE: The event queue, state machine, and a unique priority is collectively called an **active object**.

The design guarantees that the `dispatch()` operation for each state machine always runs to completion and returns to the main Arduino loop before any other event can be processed. The scheduler applies all necessary safeguards to protect the integrity of the events, the queues, and the scheduler itself from corruption by asynchronous interrupts that can preempt the main loop and post events to the queues at any time.

Figure 2: Event-driven QP/C++ framework with multiple event queues and state machines



NOTE: The "vanilla" scheduler shown in Figure 2 is an example of a **cooperative** scheduler, because state machines naturally cooperate to implicitly yield to each other at the end of each run-to-completion step. The QP/C++ framework contains also a more advanced, fully **preemptive** real-time kernel called QK. The QK kernel can be also used with Arduino when you define the macro **`QK_PREEMPTIVE`** for building your sketch. See Section 7 for more information.

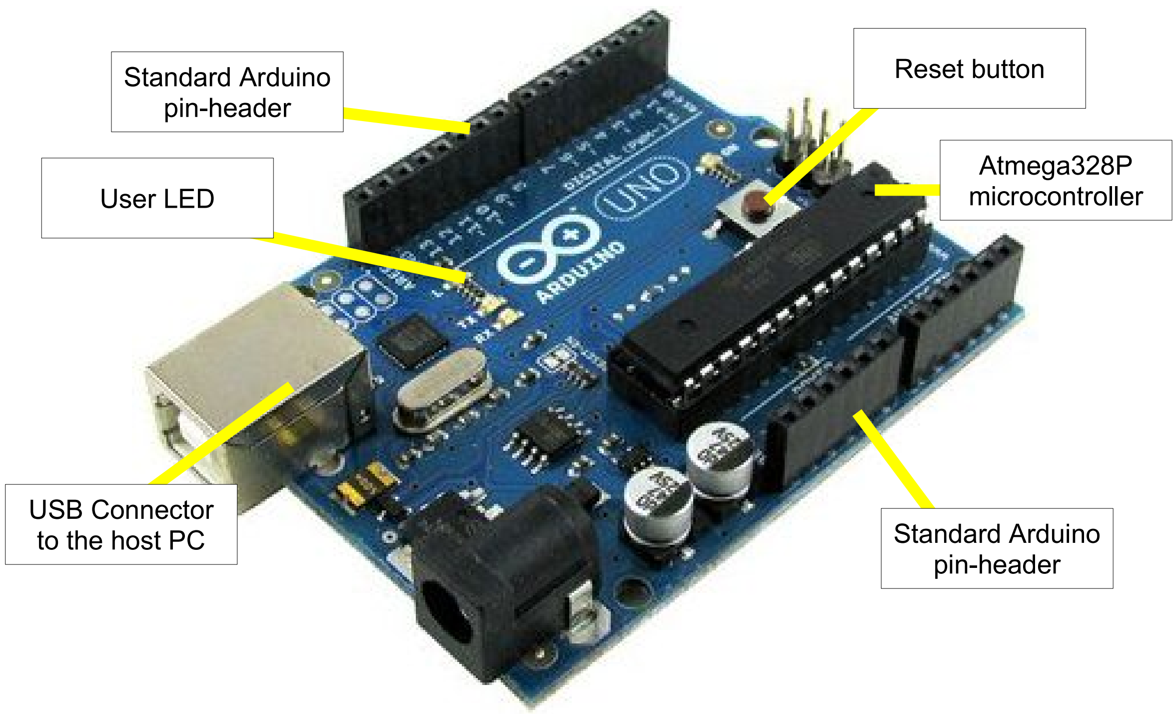
The framework shown in Figure 2 also very easily detects the condition when all event queues are empty. This situation is called the **idle condition** of the system. In this case, the scheduler calls idle processing (specifically, the function `QF::onIdle()`), which puts the processor to a low-power sleep mode and can be customized to turn off the peripherals inside the microcontroller or on the Arduino shields. After the processor is put to sleep, the code stops executing, so the main Arduino loop stops completely. Only an external interrupt can wake up the processor, but this is exactly what you want because at this point only an interrupt can provide a new event to the system.

Finally, please also note that the framework shown in Figure 2 can achieve good real-time performance, because the individual run-to-completion (RTC) steps of each state machine are typically short (execution time counted in microseconds).

2 Getting Started

To focus the discussion, this Application Note uses the Arduino UNO board based on the Atmel Atmega328P microcontroller (see Figure 3). The example code has been built entirely inside the **QM modeling tool** (version 2.3.x or higher), but it uses the compiler and libraries in the standard **Arduino 1.5.x** (the latest as of this writing), which is available for a free download from the Arduino website.

Figure 3: Arduino UNO board



NOTE: The following discussion assumes that you have downloaded and installed both the standard Arduino software and the QM tool on your computer. The free QM tool can be downloaded from: https://sourceforge.net/projects/qpc/files/QM. Just like the standard Arduino IDE, QM is available for Windows, Linux, and Mac OS X hosts.

2.1 Software Installation

The example code is distributed in a single ZIP archive `qp_arduino_<ver>`, where <ver> stands for the Arduino software version numer (e.g., 1.5.x). The contents of this archive is shown in Listing 3.

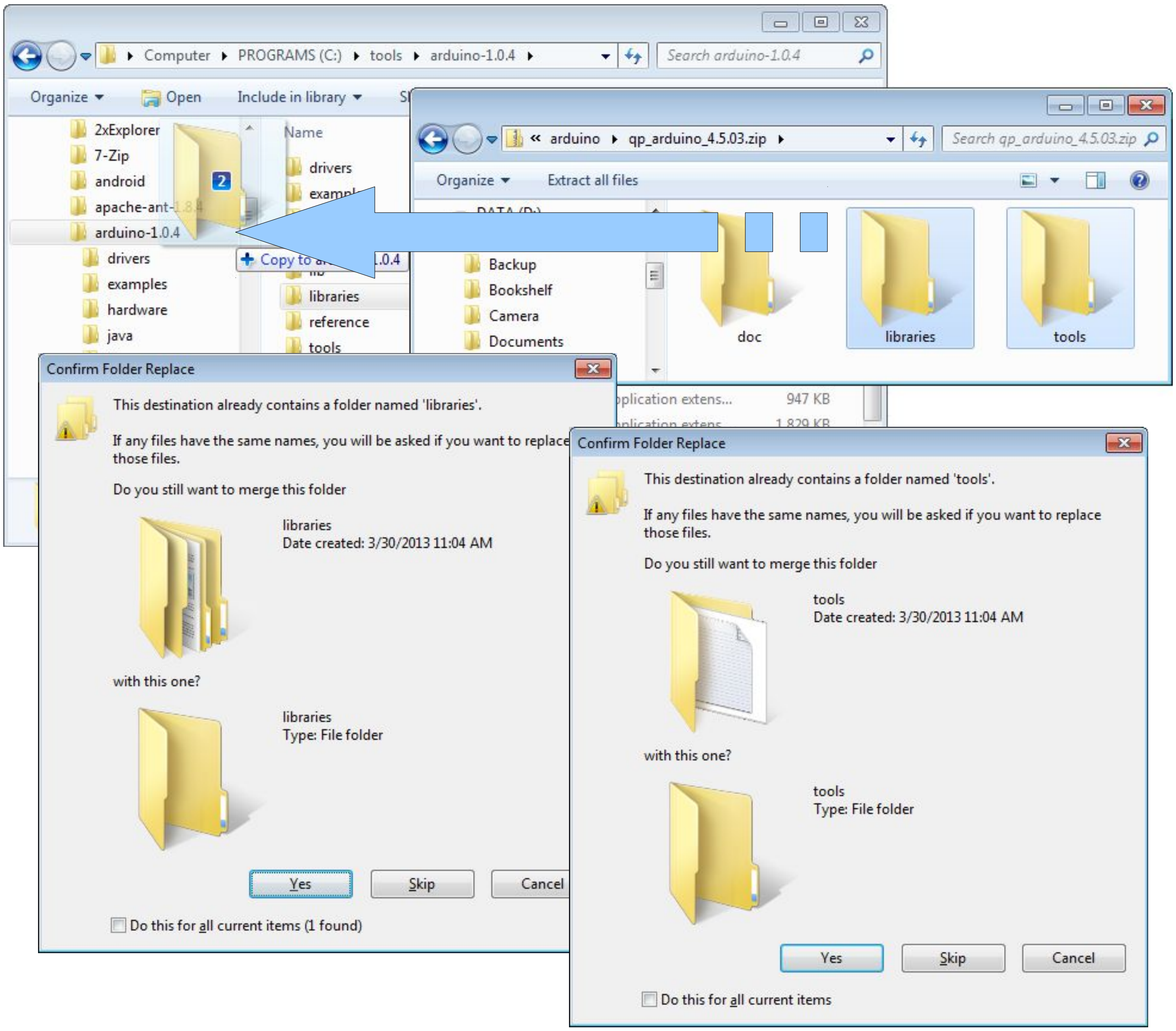
Listing 3: Contents of the qp_arduino_<ver>.zip archive for Arduion 1.5.x

```
qp_arduino_<ver>.zip   - the code accompanying this application note
+-doc/                 - documentation
| +-AN_Event-Driven_Arduino.pdf - this document
| +-AN_DPP.pdf         - Dining Philosopher Problem example application
| +-AN_PELICAN.pdf     - PEdestrian LIght CONtrolled (PELICAN) crossing example
|
+-hardware/
| +-arduino/
| | +-avr/             - Libraries for AVR-based Arduinos
| | | +-libraries/     ==> goes to <Arduino>/hardware/arduino/avr/libraries/ folder
| | | | +-QP/           - QP library for AVR-based Arduinos
| | | | | | +-examples/ - Examples for the QP library
| | | | | | | +-blinky/     - Very simple "blinky" example
| | | | | | | | +-.blinky   - The QM session file for the Blinly project
| | | | | | | | +-blinky.qm - The QM model of the Blinky application
| | | | | | | +-dpp_qk/     - DPP-QK example with preemptive QK kernel (Section 7)
| | | | | | | | +-.dpp      - The QM session file for the DPP project
| | | | | | | | +-dpp.qm    - The QM model of the DPP application
| | | | | | | +-pelican/    - PEdestrian LIght CONtrolled (PELICAN) crossing example
| | | | | | | | +-.pelican  - The QM session file for the PELICAN project
| | | | | | | | +-pelican.qm- The QM model of the PELCIAN crossing application
| | | | | | |
| | | | | | +-COPYING.txt   - terms of copying this code
| | | | | | +-GPL2.TXT      - GPL version 2 open source license
| | | | | | +-qp.cpp        - QP/C++ platform-independent implementation
| | | | | | +-qp_port.cpp   - QP/C++ port for Arduino implementation
| | | | | | +-qp_port.h     - QP/C++ port for Arduino interface
|
+-tools/              ==> goes to the <Arduino>/tools/ folder
| +-qtools/            - tools contributed Quantum Leaps
| | +-COPYING.txt      - terms of copying this code
| | +-GPL2.TXT         - GPL version 2 open source license
| | +-avrmake.tcl      - Generic TCL script for building Arduino/AVR sketches
| | +-arduiniodude.exe - Arduino programming utility (calls avrdude.exe)
| | +-rm.exe           - Command-line utility for removing files (cleanup)
```

The complete QP/C++ library code consists of just three source files: `qp_port.h`, `qp_port.cpp`, and `qp.cpp`. The QP library is accompanied with three examples: `blinky`, `pelican, and dpp_qk.` The Arduino sketch `blinky` demonstrates the very simple "blinky" sketch that blinks the User LED on the Arduino board. The sketch `pelican` demonstrates a PEdestrian LIght CONtrolled (PELICAN) crossing with a non-trivial hierarchical state machine. The scketch `dpp_qk` demonstrates the cassic Dining Philosopher Problem (DPP) with multiple state machines and the **preemptive** QK kernel. All examples contain the QM models to generate the Arduino code automatically.

You need to unzip the `qp_arduino_<ver>.zip` archive into the Arduino directory (e.g., `arduino-1.5.2\`). On Windows you can simply open the ZIP file with the Windows Explorer, select the three directories (`libraries/ examples/`, and `tools/`) and drag (or copy-and-paste) them to the Arduino directory, as shown in Figure 4. Since Arduino already has the `libraries/ examples/, and tools/` folders, you need to confirm that you want to merge these folders.

Figure 4: Unzipping `qp_arduino_<ver>.zip` into the Arduino directory. You need to confirm that you want to merge the folders `libraries` and `tools`.



Each QP example for Arduino contains a QM model, which is a file with the extension `.qm`. For example, `<qp_arduino.zip>\hardware\arduino\avr\libraries\qp\examples\pelican-\pelican.qm`, see Listing 3). These models and the QM modeling tool take Arduino programming to the next level. Instead of coding the state machines by hand, you draw them with the free QM modeling tool, attach simple action code to states and transitions, and you **generate** the complete Arduino sketch automatically—literally by a press of a button (see Figure 5).

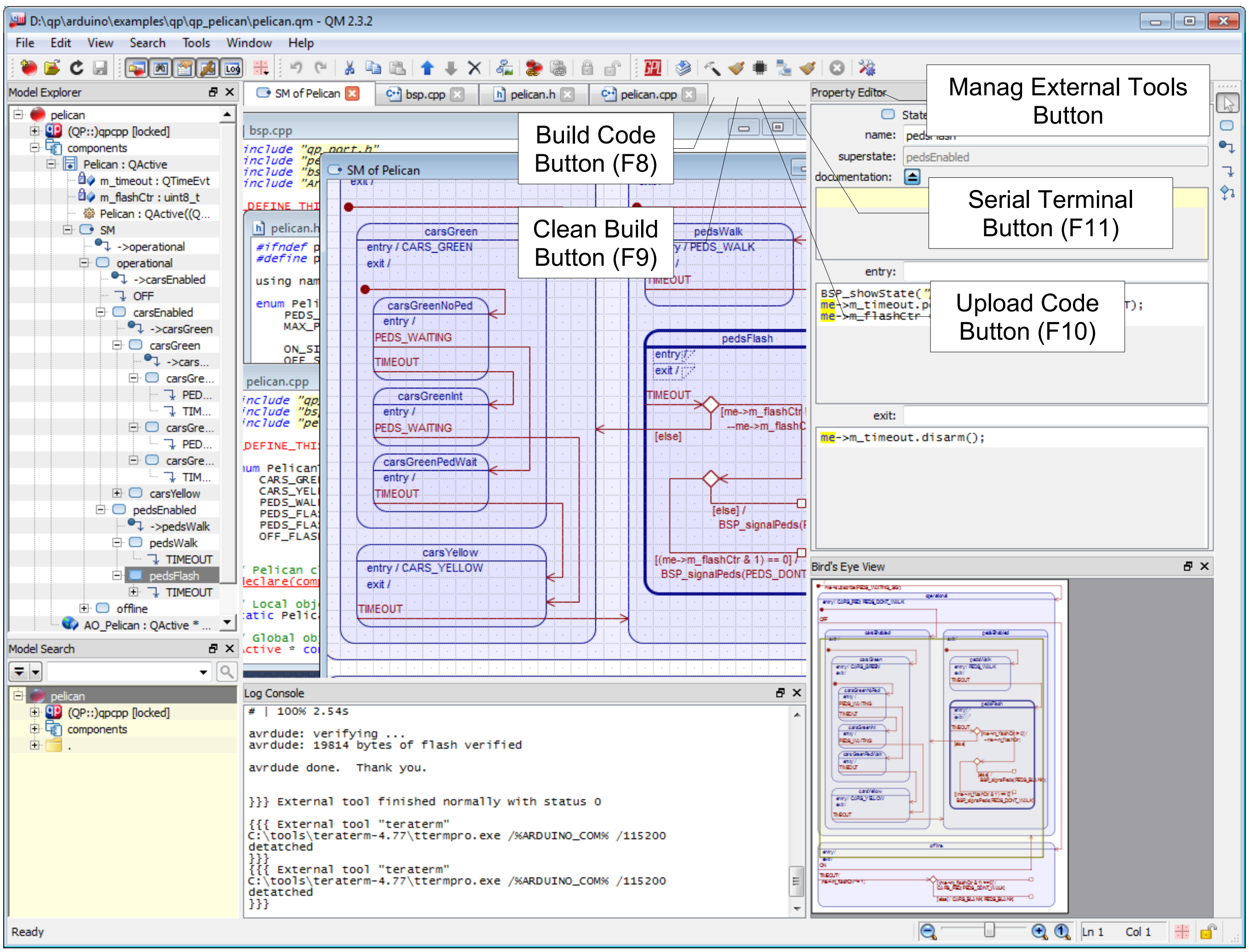
NOTE: To start working with the QM™ modeling tool, you need to download the tool from state-machine.com. QM™ is currently supported on Windows and Linux hosts. QM™ is **free** to download and **free** to use (see also Related Documents and References).

2.2 Building and Running the Blinky Example from the QM Modeling Tool

To build the provided Blinly example, change to the `<Arduino>\libraries\QP\examples\blinky` directory, and double-click on the `blinky.qm` model file. If you installed QM correctly, this will open the Blinky model in the QM tool, as shown in Figure 5. **You build and upload the QP examples directly from the QM modeling tool**, without the need to launch the standad Arduino IDE.

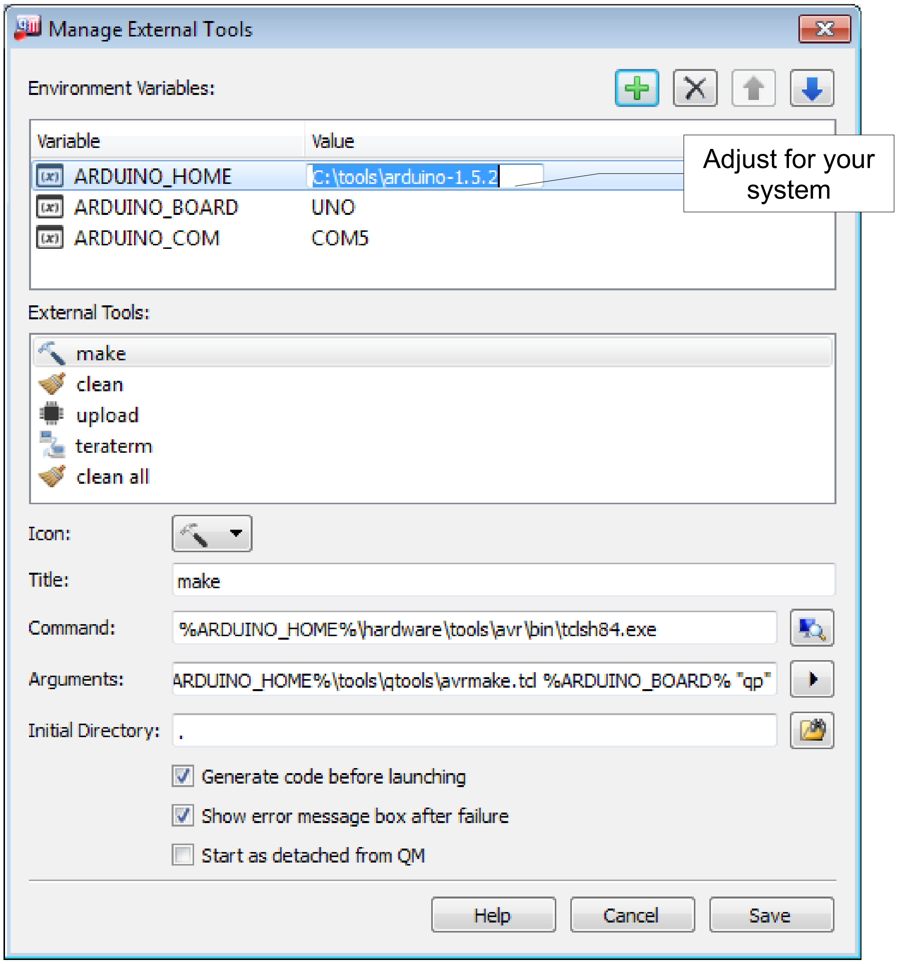
NOTE: The provided QP examples for Arduino are still compatible with the standard Arduino IDE, so you can still use the Arduino IDE, if you chose to do so, to build and upload the examples to your Arduino board.

Figure 5: The QM modeling tool showing the Blinky model and the External Tools configured for working with Arduino.



But before you can build the code, you need to adjust the location of the standard Arduino software, so that the build process can find the compiler and other required tools. You do this by clicking the Manage External Tools Button (see Figure 5) and edit the environment variable `ARDUINO_HOME` to the location of the standard Arduino software on your computer. You can also edit the name of your Arduino board (`ARDUINO_BOARD`) and the COM port used by your board (`ARDUINO_COM`).

Figure 6: Adjusting the ARDUINO_HOME environment variable in the Manage External Tools Dialog

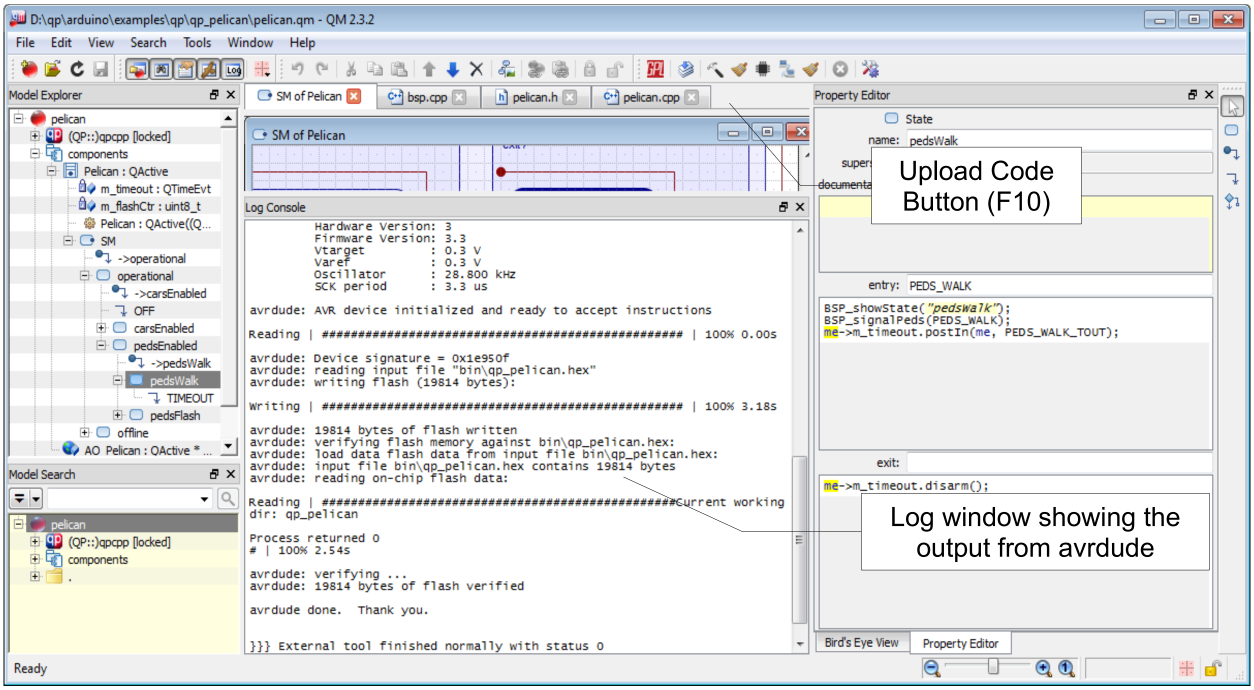


NOTE: The Manage External Tools dialog box allows you to configure up to five different external tools. The described configuration is included in the QP examples for Arduino, but you can modify the tools to suit your needs at any time

After you've adjusted the environment variables, click on the **Build Code** button (see Figure 5). The first time build takes a little longer, because it performs a clean build of all the libraries you've specified for this project. For the Blinky project, this includes the standard Arduino library plus the QP framework. The subsequent builds are much faster, because only the project files that have changed are re-built.

To upload the code to your Arduino board, you must connect the board to your computer via a USB cable. You upload the code to the Arduino microcontroller by pressing the Upload button. Please note that the upload can take several seconds to complete. After the upload, your Arduino starts executing the example. You should see the User LED blink at the rate of about once per second (see Figure 3).

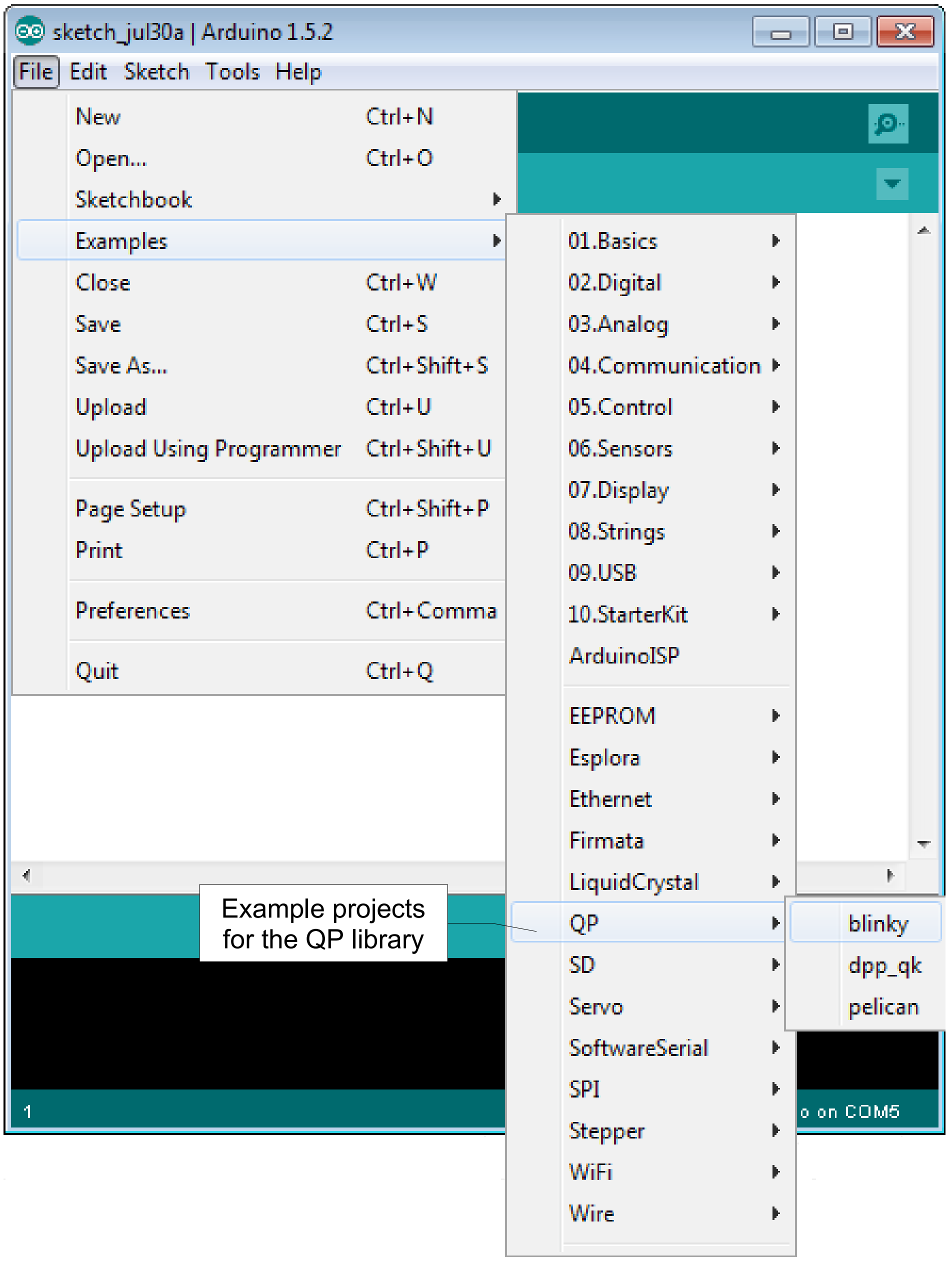
Figure 7: Uploading the code to the Arduino board



2.3 Building and Running the Blinky Example from the Arduino IDE

As mentioned before, the QP examples are still compatible with the standard Arduino IDE. For example, to build and run the Blinky example, you lanuch the Arduino Ide and open the blinky project as shown Figure 8.

Figure 8: Launching the Blinky example from the Arduino IDE



NOTE: Even though you can build the example code with the Arduino IDE, you still need to use QM to (re)generate the code. If you don't wish to use QM at all, you can code the state machines by hand, which is not that hard, because the underlying QP framework has been specifically designed for manual coding of state machines.

2.4 Running the PELICAN Crossing Example in the QM tool

The PEdestrian LIght CONtrolled (PELICAN) crossing example is built and uploaded in the similar way as the Blinky example, except you open the `pelican.qm` example QM model located in the directory `<Arduino>\hardware\arduino\avr\libraries\QP\examples\pelican`.

Before you can test the example, you need to understand the how it is supposed to work. So, the PELICAN crossing operates as follows: The crossing (see Figure 9) starts with cars enabled (green light for cars) and pedestrians disabled ("Don't Walk" signal for pedestrians). To activate the traffic light change a pedestrian must push the button at the crossing, which generates the PEDS_WAITING event. In response, oncoming cars get the yellow light, which after a few seconds changes to red light. Next, pedestrians get the "Walk" signal, which shortly thereafter changes to the flashing "Don't Walk" signal. After the "Dont' Walk" signal stops flashing, cars get the green light again. After this cycle, the traffic lights don't respond to the PEDS_WAITING button press immediately, although the button "remembers" that it has been pressed. The traffic light controller always gives the cars a minimum of several seconds of green light before repeating the traffic light change cycle. One additional feature is that at any time an operator can take the PELICAN crossing offline (by providing the OFF event). In the "offline" mode the cars get the flashing red light and the pedestrians get the flashing "Don't Walk" signal. At any time the operator can turn the crossing back online (by providing the ON event).

NOTE: The design and implementation of the PELICAN crossing application, including the PELICAN state machine, is described in the Application Note "PELICAN Crossing Application" (see Related Documents and References).

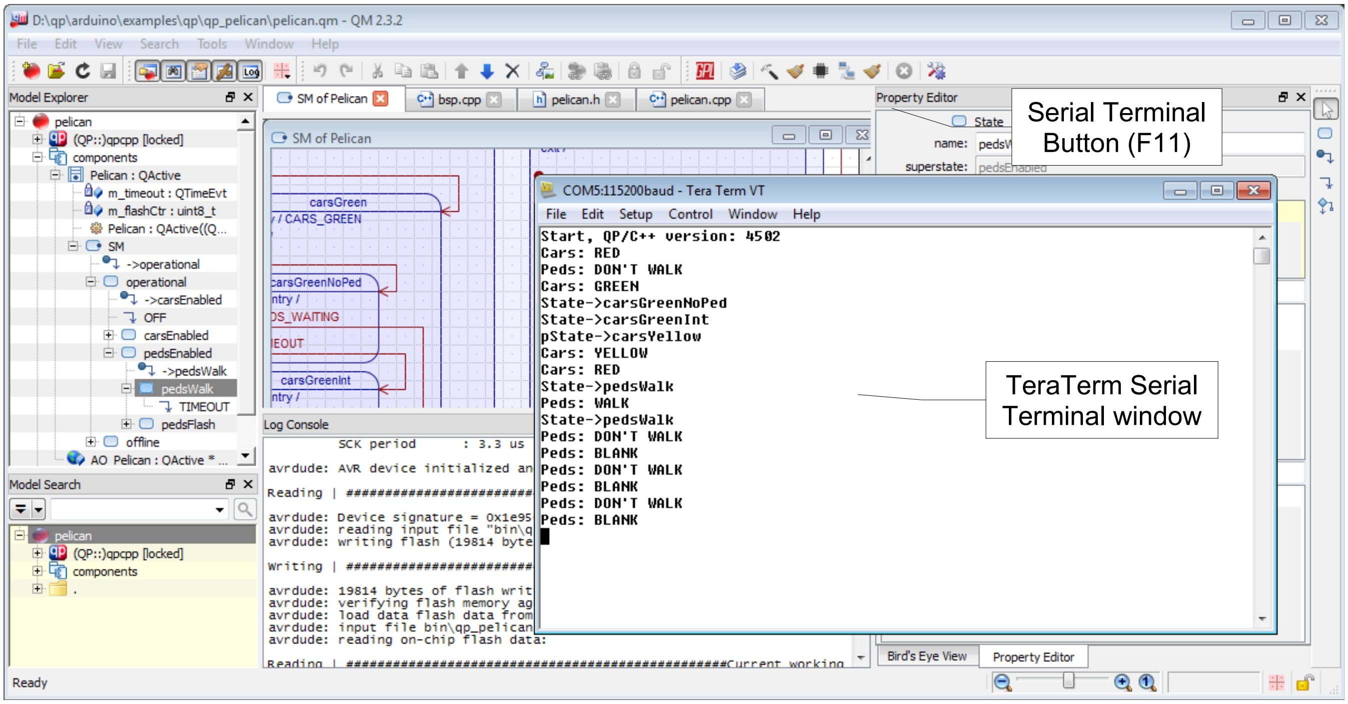
Figure 9: Pedestrian Light CONtrolled (PELICAN) crossing

After you build and upload the software to the Arduino UNO board, the User LED should start to glow with low intensity (not full on). In the PELICAN example, the User LED is rapidly turned on and off in the Arduino idle loop, which appears as a constant glow to a human eye.

To see the actual output from the PELICAN example, you need to open a **Serial Terminal** by pressing the Serial Terminal button on the QM toolbar (see Figure 5). For the Arduino UNO board, the Serial Terminal should be configured to **115200 baud rate.** To activate the PELICAN crossing, type 'p' on the terminal. This will trigger the sequence of signals for cars and pedestrians, as shown in Figure 10.

NOTE: The Serial Terminal button opens the **Tera Term** application, which a freeware serail terminal application for Winows that you need to install separately (see http://ttssh2.sourceforge.jp). You can also use any other serial terminal software instead.

Figure 10: TeraTerm Serial Terminal with the output from the PELICAN crossing

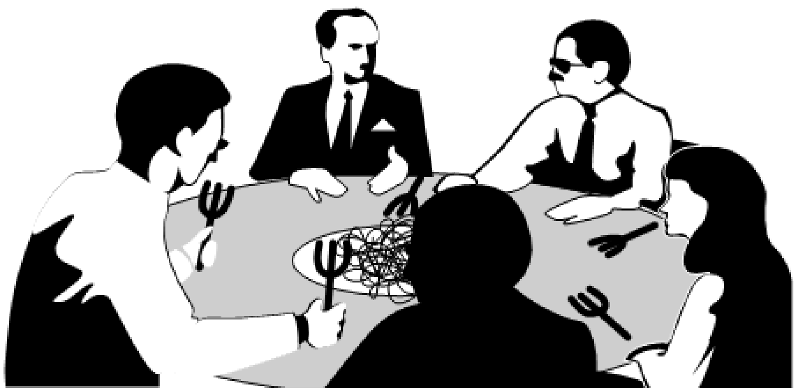


2.5 The Dining Philosophers Problem Example with the Preemptive QK Kernel.

Finally, the example code contains the Dining Philosphers Problem (DPP) example, which demonstrates mutilple state machines and the **preemptive** QK kernel (see also Section 7). This example is located in the directory `<Arduino>\hardware\arduino\avr\libraries\QP\examples\dpp_qk`.

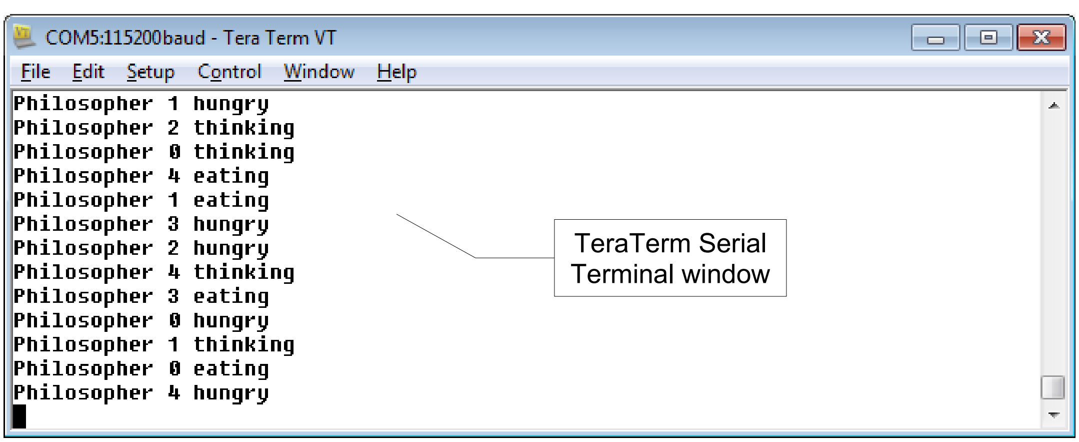
The classical Dining Philosopher Problem (DPP) was posed and solved originally by Edsger Dijikstra in the 1970s and is specified as follows: Five philosophers are gathered around a table with a big plate of spaghetti in the middle (see Figure 11). Between each two philosophers is a fork. The spaghetti is so slippery that a philosopher needs two forks to eat it. The life of a philosopher consists of alternate periods of thinking and eating. When a philosopher wants to eat, he tries to acquire forks. If successful in acquiring forks, he eats for a while, then puts down the forks and continues to think. The key issue is that a finite set of tasks (philosophers) is sharing a finite set of resources (forks), and each resource can be used by only one task at a time.

Figure 11: Dining Philosophers Problem



You build and upload the DPP-QK example to the Arduino UNO board the same way as the PELICAN example. Just like in the PELICAN example, the User LED in DPP-QK is rapidly turned on and off in the idle loop, which appears as a constant glow to a human eye.

To see the actual output from the DPP-QK example, you need to open a **Serial Terminal** by pressing the Serial Terminal button on the QM toolbar (see Figure 5). For the Arduino UNO board, the Serial Terminal should be configured to **115200 baud rate**.



2.6 Modifying the Examples to Reduce Power Consumption

As mentioned in Section 1.2, the QP/C++ framework allows you to take advantage of the Arduino processor's low-power sleep mode, which is the only way to achieve really low-power design. Both provided examples can be very easily modified to switch to the sleep mode when no events are available. In fact, the code is already provided for you, so all you need to do is just to enable this code. As shown in Listing 9, you uncomment the definition of the `SAVE_POWER` macro in the file `bsp.cpp` (Board Support Package).

After you recompile the code and download to Arduino, you will see that the User LED is no longer glowing. Actually, it is glowing, but only for a few microseconds out of every 10 milliseconds, so you cannot see it. This very low brightness of the User LED means that the Arduino Background loop uses very little power, yet the application performs exactly as before! The upcoming Section 4.1 explains what exactly happens when you define the macro `SAVE_POWER` in `bsp.cpp`.

3 The Structure of an Arduino Sketch for QP

Every event-driven Arduino sketch for QP consists of four rough groups: setup, events, active objects, and board support package (BSP). Typically, the main sketch file (the .ino file) contains the Aruino setup() function. Since the events are shared, they are defined in a header file (the .h file). Active objects are defined in source files (the .cpp files), one active object per file. The following sections describe these elements in more detail.

3.1 The setup() function

Listing 4 shows an example Arduino sketch for QP. This sketch defines only the setup() function. The explanation section immediately following the listing highlights the main points.

Listing 4: Typical Arduino sketch for QP (file dpp.ino)

```
 (0) #include "Arduino.h" // don't forget to include!!!
 (1) #include "qp_port.h"
 (2) #include "dpp.h"
 (3) #include "bsp.h"

     // Local-scope objects -------------------------------------------------------
 (4) static QEvt const *l_tableQueueSto[N_PHILO];
 (5) static QEvt const *l_philoQueueSto[N_PHILO][N_PHILO];
 (6) static QSubscrList l_subscrSto[MAX_PUB_SIG];
 (7) QF_MPOOL_EL(TableEvt) l_smlPoolSto[2*N_PHILO]; // storage for small event pool

     //............................................................................
 (8) void setup() {
 (9)     BSP_init();                                           // initialize the BSP

(10)     QF::init();       // initialize the framework and the underlying RT kernel

                                                      // initialize event pools...
(11)     QF::poolInit(l_smlPoolSto, sizeof(l_smlPoolSto), sizeof(l_smlPoolSto[0]));

(12)     QF::psInit(l_subscrSto, Q_DIM(l_subscrSto));     // init publish-subscribe

                                                        // start the active objects...
         uint8_t n;
         for (n = 0; n < N_PHILO; ++n) {
(13)         AO_Philo[n]->start((uint8_t)(n + 1),
                                l_philoQueueSto[n], Q_DIM(l_philoQueueSto[n]));
         }
(14)     AO_Table->start((uint8_t)(N_PHILO + 1),
                         l_tableQueueSto, Q_DIM(l_tableQueueSto));
     }
```

(0) Each sketch for must include the standard **"Arduino.h"** header file. This is necessary to make the sketch compatible with the arduinomake.tcl build script (see also Section ???)

(1) Each sketch for QP imports the QP library port to Arduino (the <qp_port.h> header file).

(2) The application header file (dpp.h in this case) contains the definition of signals and events for the application. This file is shared among most files in the sketch.

(3) The header file bsp.h contains the facilities provided by the Board Support Package. This file is also shared among most files in the sketch.

(4-5) The application must provide storage for event queues of all active objects used in the application. Here the storage is provided at compile time through the statically allocated arrays of immutable (const) pointers to events.

(6) If the application uses the publish-subscribe event delivery mechanism supported by QP, the application must provide the storage for the subscriber lists. The subscriber lists remember which active objects have subscribed to which events. The size of the subscriber database depends on the number of published events `MAX_PUB_SIG` found in the application header file.

(7) The application must also provide storage for the event pools that the QP framework uses for fast and deterministic dynamic allocation of events. Each event pool can provide only fixed-size memory blocks. To avoid wasting the precious memory (RAM) by using massively oversized pools, the QP framework can manage up to three event pools of different sizes. The DPP application uses only one pool, which can hold `TableEvt` events and events without parameters (`QEvt`).

(8) You provide only the definition of the Arduino `setup()` function. You don't define the `loop()` function, because it is provided in the framework (see Section 5.2).

(9) The function `BSP_init()` initializes the Arduino board for this application and is defined in the `bsp.cpp` file.

(10) The function `QF::init()` initializes the QF framework and you need to call it before any other QF services.

(11) The function `QF::poolInit()` initializes the event pool. The parameters of this function are: the pointer to eh event pool storage, the size of this storage, and the block-size of the this pool. You can call this function up to three times to initialize up to three event pools. The subsequent calls to `QF::poolInit()` must be made in the increasing order of block-size. For instance, the small block-size pool must be initialized before the medium block-size pool

(12) The function `QF::psInit()` initializes the publish-subscribe event delivery mechanism of QP. The parameters of this function are: the pointer to the subscriber-list array and the dimension of this array.

NOTE: The utility macro `Q_DIM()` provides the dimension of a one-dimensional array `a[]` computed as `sizeof(a)/sizeof(a[0])`, which is a compile-time constant.

(13-14) The function `start()` is defined in the framework class `QActive` and tells the framework to start managing an active object as part of the application. The function takes the following parameters: the pointer to the active object, the priority of the active object, the pointer to its event queue, the dimension (length) of that queue. The active object priorities in QP are numbered from 1 to `QF_MAX_ACTIVE`, inclusive, where a higher priority number denotes higher urgency of the active object. The constant `QF_MAX_ACTIVE` is defined in the QP port header file `qf_port.h` (see Section 5.1).

At this point you have provided the QP framework with all the storage and active object information it needs to manage your application. The next step of execution of the Arduino sketch is the call to the `loop()` function, which is defined in the QP port to Arduino. As described in the upcoming Section 5.2, the `loop()` function executes the main event-loop shown in Figure 2.

NOTE: You do **NOT** provide the defnition of the Arduino `loop()` function.

3.2 The Application Interface (Events and Signals)

An **event** represents occurrence that is interesting to the system. An event consists of two parts. The part of the event called the **signal** conveys the type of the occurrence (what happened). For example, in the DPP application the `EAT_SIG` signal represents the permission to eat to a Philosopher active object, whereas `HUNGRY_SIG` conveys to the Table active object that a Philosopher wants to eat. An event can also contain additional quantitative information about the occurrence in the form of **event parameters**. For example, the `HUNGRY_SIG` signal is accompanied by the number of the Philosopher. In QP events are represented as instances of the `QEvt` class provided by the framework. Specifically, the `QEvt` class contains the member sig, to represent the signal of that event. Event parameters are added in the subclasses of `QEvt`, that is classes that inherit from `QEvt`.

Because events are shared among most of the application components, it is convenient to declare them in a separate header file (e.g., `dpp.h` for the DPP application). Listing 5 shows the interface for the DPP application. The explanation section immediately following the listing highlights the main points.

Listing 5: The application header file for DPP (file `dpp.h`)

```
(1) using namespace QP;//assume the QP namespace for all files in this application

(2) enum DPPSignals {
(3)     EAT_SIG = Q_USER_SIG,         // published by Table to let a philosopher eat
        DONE_SIG,                      // published by Philosopher when done eating
        TERMINATE_SIG,               // published by BSP to terminate the application
(4)     MAX_PUB_SIG,                                     // the last published signal

        HUNGRY_SIG,                          // posted from hungry Philosopher to Table
(5)     MAX_SIG                                                   // the last signal
    };

(6) struct TableEvt : public QEvt {
         uint8_t philoNum;                                     // philosopher number
    };

    enum { N_PHILO = 5 };                                    // number of philosophers
```

(1) This directive establishes the default namespace to be QP, meaning that all elements not qualified explicitly are assumed to come from the namespace QP.

> **NOTE:** Starting from QP version 4.5.xx, the directive "`using namespace QP`" is necessary to use the QP framework without explicitly adding the prefix "`QP::`" to every element coming from the framework. This directive is also necessary for compatibility with the code generated by the QM tool version 2.3.x or higher.

(2) In QP, event signals are enumerated constants. Placing all signals in a single enumeration is particularly convenient to avoid inadvertent overlap in the numerical values of the different signals.

(3) The application-level signals do not start form zero but rather must be offset by the constant `Q_USER_SIG`. Also note that by convention the suffix `_SIG` is attached to the signals, so you can easily distinguish signals from other constants in your code. Typically the suffix `_SIG` is dropped in the state diagrams to reduce the clutter.

(4) The constant MAX_PUB_SIG delimits the published signals from the rest. You can save some RAM by providing a lower limit of published signals to QP (MAX_PUB_SIG) rather than the maximum of all signals used in the application.

(5) The last enumeration MAX_SIG indicates the maximum of all signals used in the application.

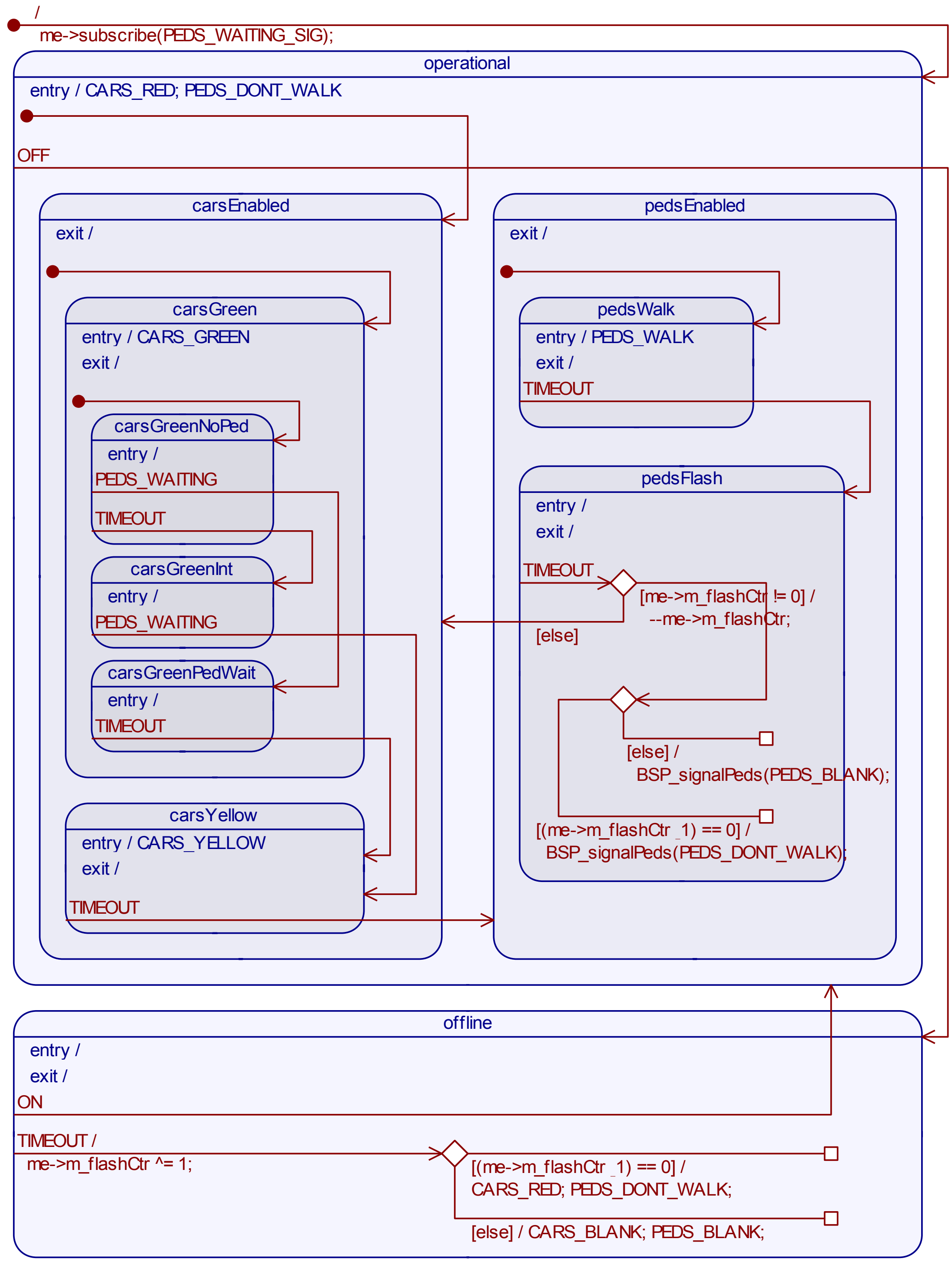
(6) The structure TableEvt represents a class of events to convey the Philosopher number in the event parameter. TableEvt inherits QEvt and adds the philoNum parameter to it.

(7-8) These global pointers represent active objects in the application and are used for posting events directly to active objects. Because the pointers can be initialized at compile time, they are declared const. The active object pointers are "opaque" because they cannot access the whole active object, only the part inherited from QActive.

3.3 The State Machines

The QP framework allows you to work with the modern **hierarchical** state machines (a.k.a., UML statecharts). For example, Figure 12 shows the HSM for the PELICAN crossing.

Figure 12: The hierarchical state machine of the PELICAN crossing



The biggest advantage of hierarchical state machines (HSMs) compared to the traditional finite state machines (FSMs) is that HSMs remove the need for repetitions of actions and transitions that occur in the non-hierarchial state machines. Without this ability, the complexity of non-hierarchical state machines “explodes” exponentially with the complexity of the modeled system, which renders the formalism impractical for real-life problems.

NOTE: The state machine concepts, state machine design, and hierarchical state machine implementation in C++ are not specific to Arduino and are out of scope of this document. The design and implementation of the DPP example is described in the separate Application Note “Dining Philosophers Problem” (see Related Documents and References). The PELICAN crossing example is described in the Application Note “PELICAN Crossing”. Both these application notes are included in the ZIP file that contains the QP library and examples for Arduino (see Listing 3).

Once you design your state machine(s), you can code it by hand, which the QP framewrok makes particularly straightforward, or you can use the QM tool to generate code automatically, as described in Section 14.

4 Board Support Package (BSP) for Arduino™

The QP example sketches (DPP and PELICAN) contain the file `bsp.cpp`, which stands for Board Support Package (BSP). This BSP contains all board-related code, which consists of the board initialization, interrupt service routines (ISRs), idle processing, and assertion handler. The following sections explain these elements.

4.1 BSP Initialization

The BSP initialization is done in the function `BSP_init()`. It is minimal, but generic for most Arduino boards. The most important step is initialization of the User LED (connected to PORTB) and the serial port, which is used to output the data in the example applications. If you use other peripherals as well, you `BSP_init()` is the best for such additional initialization code.

Listing 6: BSP_init() function for the DPP example (file `bsp.cpp`)

```
void BSP_init(void) {
    DDRB  = 0xFF;                    // All PORTB pins are outputs (user LED)
    PORTB = 0x00;                                       // drive all pins low
    Serial.begin(115200);
    Serial.println("Start");
}
```

4.2 Interrupts

An **interrupt** is an asynchronous signal that causes the Arduino processor to save its current state of execution and begin executing an Interrupt Service Routine (ISR). All this happens in hardware (without any extra code) and is very fast. After the ISR returns, the processor restores the saved state of execution and continues from the point of interruption.

You need to use interrupts to work with QP. At the minimum, you must provide the **system clock tick** interrupt, which allows the QP framework to handle the timeout request that active objects make. You might also want to implement other interrupts as well.

When working with interrupts you need to be careful when you enable them to make sure that the system is ready to receive and process interrupts. QP provides a special callback function `QF::onStartup()`, which is specifically designed for configuring and enabling interrupts. `QF::onStartup()` is called after all initialization completes, but before the framework enters the endless event loop. Listing 7 shows the implementation of `QF::onStartup()` for QP, which configures and starts the system clock tick interrupt. If you use other interrupts, you need to add them to this function as well.

Listing 7: Configuring and starting interrupts in `QF::onStartup()`

```
    void QF::onStartup(void) {
(1)           // 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;
(2)     OCR2A = TICK_DIVIDER;      // must be loaded last for Atmega168 and friends
    }
```

(1) This BSP uses Timer2 as the source of the periodic clock tick interrupt (Timer1 is already used to provide the Arduino `milli()` service).

(3) The output compare register (OCR2A) is loaded with the value that determines the length of the clock tick interrupt. This value, in turn, is determined by the `BSP_CLICKS_PER_SEC` constant, which currently is set to 100 times per second.

For each enabled interrupt you need to provide an Interrupt Service Routine (ISR). ISRs are **not** the regular C++ functions, because they need a special code to enter and exit. Nonetheless, the Arduino compiler (WinAVR) supports writing ISRs in C++, but you must inform the compiler to generate the special ISR code by using the macro `ISR()`. Listing 8 Shown the system clock tick ISR for Arduino.

Listing 8: System clock tick ISR for Arduino (file `bsp.cpp`)

```
(1) ISR(TIMER2_COMPA_vect) {
        // No need to clear the interrupt source since the Timer2 compare
        // interrupt is automatically cleard in hardware when the ISR runs.

(2)     QF::tick();                              // process all armed time events
    }
```

(1) The definition of every ISR must begin with the `ISR()` macro.

(2) The system clock tick must invoke `QF::tick()` and can also perform other actions, if necessary.

4.3 Idle Processing

The following Listing 9 shows the `QF::onIdle()` "callback" function, which is invoked repetitively from the Arduino `loop()` function whenever there are no events to process (see Figure 2). This callback function is located in the `bsp.cpp` file in each QP sketch.

Listing 9: Activating low-power sleep mode for Arduino (file `bsp.cpp`)

```
(1) void QF::onIdle() {

(2)     USER_LED_ON();      // toggle the User LED on Arduino on and off, see NOTE1
(3)     USER_LED_OFF();

(4) #ifdef SAVE_POWER

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

        // never separate the following two assembly instructions, see NOTE2
(6)     __asm__ __volatile__ ("sei" "\n\t" :: );
(7)     __asm__ __volatile__ ("sleep" "\n\t" :: );

(8)     SMCR = 0;                                              // clear the SE bit

    #else
(9)     QF_INT_ENABLE();
    #endif
    }
```

(1) The callback function `QF::onIdle()` is called from the Arduino loop whenever the event queues have no more events to process (see also Section 3), in which case only an external 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.

NOTE: The article "Using Low-Power Modes in Foreground / Background Systems" (http://www.embedded.com/design/202103425) explains the race condition associated with going to power saving mode and how to avoid this race condition safely in the simple foreground/background type schedulers.

(2-3) The Arduino's User LED is turned on and off to visualize the idle loop activity. Because the idle callback is called very often the human eye perceives the LED as glowing at a low intensity. The brightness of the LED is proportional to the frequency of invocations of the idle callback. Please note that the LED is toggled with interrupts locked, so no interrupt execution time contributes to the brightness of the User LED.

(4) When the macro `SAVE_POWER` is defined, the following code becomes active.

(5) 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.

(6) The interrupts are unlocked with the `SEI` instruction.

(7) 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.

(8) As recommended in the AVR datasheet, the `SMCR` register should be explicitly cleared upon the exit from the sleep mode.

(9) If the macro `SAVE_POWER` is not defined the low-power mode is not activated so the processor never goes to sleep. However, even in this case you **must** enable interrupts, or else they will stay disabled forever and your Arduino will freeze.

4.4 Assertion Handler `Q_onAssert()`

As described in Chapter 6 of the book "Practical UML Statecharts in C/C++, Second Edition" [PSiCC2] (see Section Related Documents and References), 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 disables all interrupts to prevent any further damage, turns on the User LED to alert the user, and then performs the software reset of the Arduino processor.

```
void Q_onAssert(char const Q_ROM * const Q_ROM_VAR file, int line) {
    QF_INT_DISABLE();                                // disable all interrupts
    USER_LED_ON();                                  // User LED permanently ON
    asm volatile ("jmp 0x0000");    // perform a software reset of the Arduino
}
```

5 QP/C++ Library for Arduino™

The QP framework is deployed as an Arduino library, which you import into your sketch. As shown in Listing 3, the whole library consists just of three files. The following sections describe these files.

5.1 The `qp_port.h` Header File

When you import the library (Arduiono IDE, menu Sketch | Import Library | qp), the Arduino IDE will insert the line “`#include "qp_port.h"`” into your currently open sketch file. The `qp_port.h` file (shown in Listing 10) contains the adaptations (port) of QP to the AVR processor followed by the platform-independent code for QP. Typically, you should not need to edit this file, except perhaps when you want to increase the maximum number of state machines you can use in your applications. This number is configured by the macro `QF_MAX_ACTIVE` and currently is set to 8. You can increase it to 63, inclusive, but this costs additional memory (RAM), so you should not go too high unnecessarily.

Listing 10: The qp_port.h header file for the QP library

```
#ifndef qp_port_h
#define qp_port_h

#include <stdint.h>                          // C99-standard exact-width integers
#include <avr/pgmspace.h>    // accessing data in the program memory (PROGMEM)
#include <avr/io.h>                                          // SREG definition
#include <avr/interrupt.h>                                         // cli()/sei()

                   // the macro QK_PREEMPTIVE selects the preemptive QK kernel
                   // (default is the cooperative "Vanilla" kernel)
//#define QK_PREEMPTIVE         1
                                  // allow using the QK priority ceiling mutex
//#define QK_MUTEX  1

                        // various QF object sizes configuration for this port
#define QF_MAX_ACTIVE           8
#define QF_EVENT_SIZ_SIZE       1
#define QF_EQUEUE_CTR_SIZE      1
#define QF_MPOOL_SIZ_SIZE       1
#define QF_MPOOL_CTR_SIZE       1
#define QF_TIMEEVT_CTR_SIZE     2

                         // the macro 'PROGMEM' allocates const objects to ROM
#define Q_ROM                   PROGMEM

                               // the macro 'Q_ROM_BYTE' reads a byte from ROM
#define Q_ROM_BYTE(rom_var_)    pgm_read_byte_near(&(rom_var_))

                                              // QF interrupt disable/enable
#define QF_INT_DISABLE()        cli()
#define QF_INT_ENABLE()         sei()

                                           // QF critical section entry/exit
#define QF_CRIT_STAT_TYPE       uint8_t
#define QF_CRIT_ENTRY(stat_)    do { \
    (stat_) = SREG; \
    cli(); \
```

```
} while (0)
#define QF_CRIT_EXIT(stat_)      (SREG = (stat_))

#ifdef QK_PREEMPTIVE
                                                // QK interrupt entry and exit
#define QK_ISR_ENTRY()    (++QK_intNest_)

#define QK_ISR_EXIT()      do { \
    --QK_intNest_; \
    if (QK_intNest_ == (uint8_t)0) { \
        uint8_t p = QK_schedPrio_(); \
        if (p != (uint8_t)0) { \
            QK_sched_(p); \
        } \
    } \
} while (0)
                                  // allow using the QK priority ceiling mutex
#define QK_MUTEX  1

#endif                                                        // QK_PREEMPTIVE

//////////////////////////////////////////////////////////////////////////////
// DO NOT CHANGE ANYTHING BELOW THIS LINE
. . .
#endif                                                     // qp_port_h
```

5.2 The qp_port.cpp File

The `qp_port.cpp` source file (shown in Listing 11) contains the Arduino-specific adaptation of QP. This file defines the Arduino `loop()` function, which calls the QP framework to run the application. The function `QF::run()` implements the event loop shown in Figure 2.

Listing 11: The qp_port.cpp source file for the QP library

```
#include "qp_port.h"                                              // QP port

//Q_DEFINE_THIS_MODULE("qp_port")

//............................................................................
extern "C" void loop() {
    (void)QP_ QF::run(); //run the application, NOTE: QF::run() doesn't return
}

//............................................................................
#ifdef QK_PREEMPTIVE
void QK_init(void) {
}

                 // device driver signal offset at the top of the signal range
#if (Q_SIGNAL_SIZE == 1)
    #define DEV_DRIVER_SIG  static_cast<QP_ QSignal>(0xFFU - 8U)
#elif (Q_SIGNAL_SIZE == 2)
    #define DEV_DRIVER_SIG  static_cast<QP_ QSignal>(0xFFFFU - 32U)
#elif (Q_SIGNAL_SIZE == 4)
    #define DEV_DRIVER_SIG  static_cast<QP_ QSignal>(0xFFFFFFFFU - 256U)
```

```
#else
    #error "Q_SIGNAL_SIZE not defined or incorrect"
#endif

#endif
```

NOTE: The QP framework provides the generic definition of the Arduino `loop()` function, so that you do **not** need to provide it in your applications. In fact, your Arduino sketch will not build correctly if you define the `loop()` function yourself. All you need to provide is the Arduino `setup()` function.

Additionally, the `qp_port.cpp` source file provides the definition of the C++ `operator delete()`, which somehow the Arduino compiler (WinAVR) uses when virtual functions are present. You should not edit the `qp_port.cpp` source file.

5.3 The `qp.cpp` File

The `qp.cpp` source file contains the platform-independent code of the QP library, which contains the facilities for executing state machines, queuing events, handling time, etc. You should not edit the `qp.cpp` source file.

6 QP Tools for Arduino

The QP/Arduino ZIP file comes with some special tools, which, among others, enable you to build the Arduino projects (sketches in the Arduino talk) and upload the code to the Arduino board directly from the QM tool. This section descirbes the additional command-line tools for Arduino contributed to the Arduino community in the QP/Arduino ZIP file.

NOTE: The tools included in the QP/Arduino ZIP file are usable even outside the QM tool, because they work from the **command line**.

Listing 12: The tools provided in the qp_arduino_<var>.zip file

```
+-tools/               ==> goes to the <Arduino>/tools/ folder
| +-qtools/              - tools contributed Quantum Leaps
| | +-avrmake.tcl        - Generic TCL script for building Arduino/AVR sketches
| | +-arduiniodude.exe - Arduino programming utility (calls avrdude.exe)
| | +-rm.exe             - Command-line utility for removing files (cleanup)
```

6.1 The avrmake.tcl Build Script

The `avrmake.tcl` build script implements the standard Arduino build process, as documented in http://arduino.cc/en/Hacking/BuildProcess, in a portable TCL script.

NOTE: The TCL shell interpreter is included in the standard Arduino distribution for Windows (`<Arduino>\hardware\tools\avr\bin\tclsh84.exe`) and on other platforms, such as Linux, the TCL interpreter is available by default.

The `avrmake.tcl` build script is designed to be called from the directory containing the Arduino sketch (`.ino` file). From this directory, you call the script as follows:

```
tclsh avrmake.tcl <BARD> "<LIBS>" ["<DEFINES>"]
```

For example, to build a sketch for the UNO board, using the QP library as well as the Ethernet library and providing the definitions for the symbols `BAR` and `FOO=25`, you call the `avrmake.tcl` build script as follows:

```
tclsh avrmake.tcl UNO "QP Ethernet" "BAR FOO=25"
```

NOTE: The `avrmake.tcl` build script requires the environment variable **`ARDUINO_HOME`** to be defined and point to the installation directory of the standard Arduino software.

The `avrmake.tcl` build script works as follows:

1 Builds the specified Arduino libraries using the `avr-gcc` compiler included in the standard Arduino distribution. The libraries must exist in the `<Arduino>\libraries\` directory and must be structured according the the Arduino standard. The source code in the libraries can be located in the library directory as well as the `utility\` sub-directory. The compiled libraries (`.a` files) are created in the `lib\` sub-directory of the project directory.

2 Generates the dependency files for all `.ino`, `.cpp`, and `.c` source code files found in the project directory. The dependency files (`.d` files) contain the information of all files included a given source file, so that if any of these files changes, the dependent source file can be recompiled. The dependency files are generated in the `bin\` sub-directory of the project directory.

3 Parses the <Arduino>\hardware\arduino\boards.txt to determine the exact MCU type, the AVR core, the Build variant, etc. needed to build the code. This information is extracted from the boards.txt file based on the Arduino board parameter provided to the avrmake.tcl build script.

4 Builds the object files for all .ino, .cpp, and .c source code files that need recompilation. The object files are generated in the bin\ sub-directory of the project directory.

5 Links the object files (.o files) in the bin\ sub-directory and the libraries (.a files) in the lib\ sub-directory to form the <project>.elf file in the bin\ sub-directory of the project directory, where <project> is the name of the project directory. The naming of the compiled image is chosen for compatibility with the standard Arduino build process in the Arduino IDE.

6 Generates the HEX file (.hex file) from the .elf file in the bin\ sub-directory, which is used for uploading the image to the Arduino board.

6.2 The arduinodude Utility for Uploading Code to Arduino

The arduinodude.exe utility is a thin wrapper around the standard avrdude.exe utility for uploading the code to the Arduino boards.

NOTE: The avrdude.exe utility is provided in the standard Arduino distribution in the directory <Arduino>\hardware\tools\avr\bin\avrdude.exe).

The purpose of the arduinodude.exe utility is to shield the user from the complexity of the parameters requried by the avrdude.exe, which can be extracted from the <Arduino>\hardware\arduino\-boards.txt file based only on the Arduino boad name.

NOTE: The arduiodude.exe utility requires the environment variable **ARDUINO_HOME** to be defined and point to the installation directory of the standard Arduino software.

The arduinodude.exe utility is designed to be called from the directory containing the Arduino sketch (.ino file). From this directory, you call the script as follows:

```
arduinodude.exe <BARD> <COM-port>
```

For example, to upload a bliky image to the Arduino UNO port connected to COM5, you call the arduinodude.exe utility as follows:

```
arduinodude.exe UNO COM5
```

The image loaded to the Arduino board is taken from the bin\<project>.hex file, where <project> is the name of the project directory. The naming of the compiled image is chosen for compatibility with the standard Arduino build process in the Arduino IDE.

6.3 The rm Utility for Cleaning Up the Build

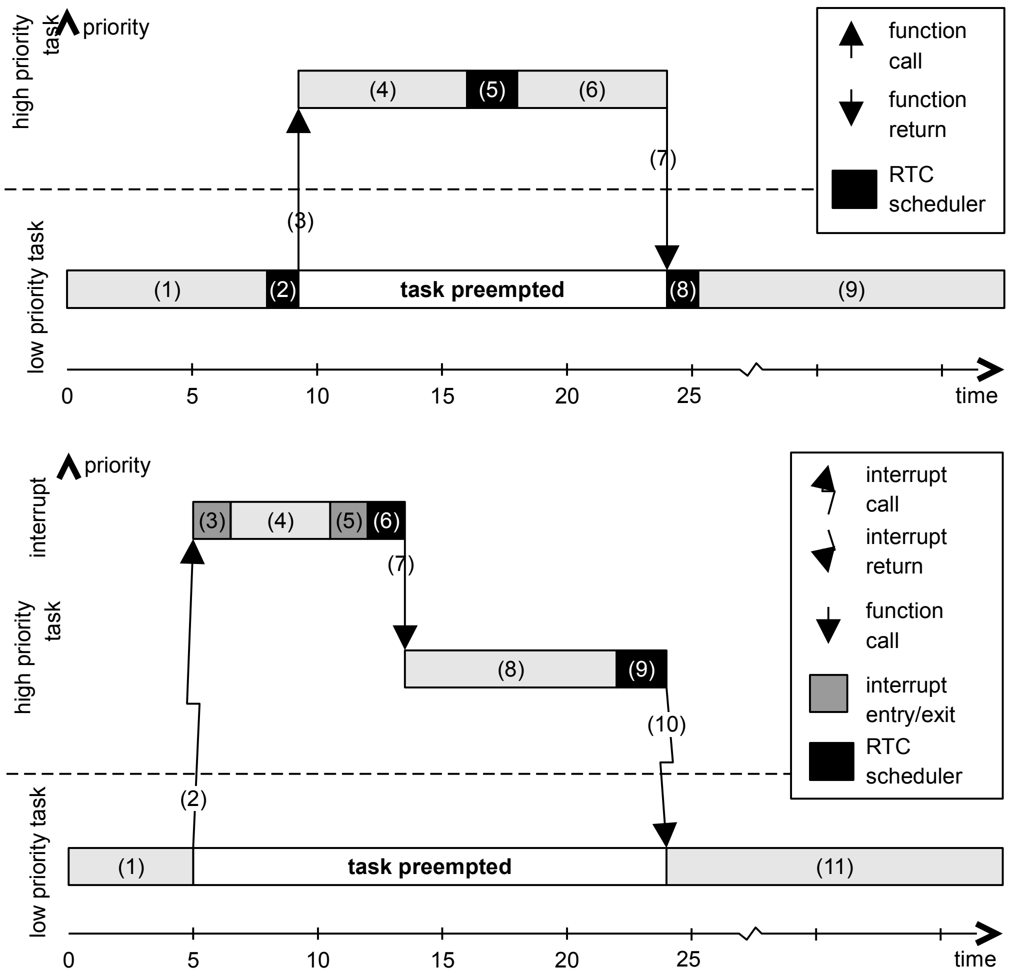
For convenience of cleaning your builds, the qtools\ directory contians the rm.exe utility for deleting files. The rm.exe utiliy on Windows is designe to work the same way as the rm utility in Unix. The utility takes a list of files to be deleted. The files can be also specfied using wild-card, such as bin*.o or bin*.d.

7 Using The Preemptive QK Kernel

The structure of the QP/C++ framework for Arduino discussed in Section 1.3 corresponds to the simple cooperative scheduler called "Vanilla". However, the QP framework can also execute Arduino applications using the **preemptive QK kernel**. The difference between non-preemptive kernel (like "Vanilla") and preemptive kernel (like QK) is shown in Figure 13.

NOTE: A **kernel** manages computer time by assigning the processor to various tasks.

Figure 13: Execution profiles of a non-preemptive kernel (top) and a preemptive kernel (bottom).



A non-preemptive kernel (panel (a) in Figure 13) gets control only after completion of the processing of each event. In particular, Interrupt Service Routines (ISRs) always return to the same task that they preempted. If an ISR posts an event to a higher-priority task than currently executing, the higher-priority task needs to wait until the original task completes. The task-level response of a non-preemptive kernel is not deterministic, because it depends when other tasks complete event processing. The upside is a much easier sharing of resources (such as global variables) among the tasks.

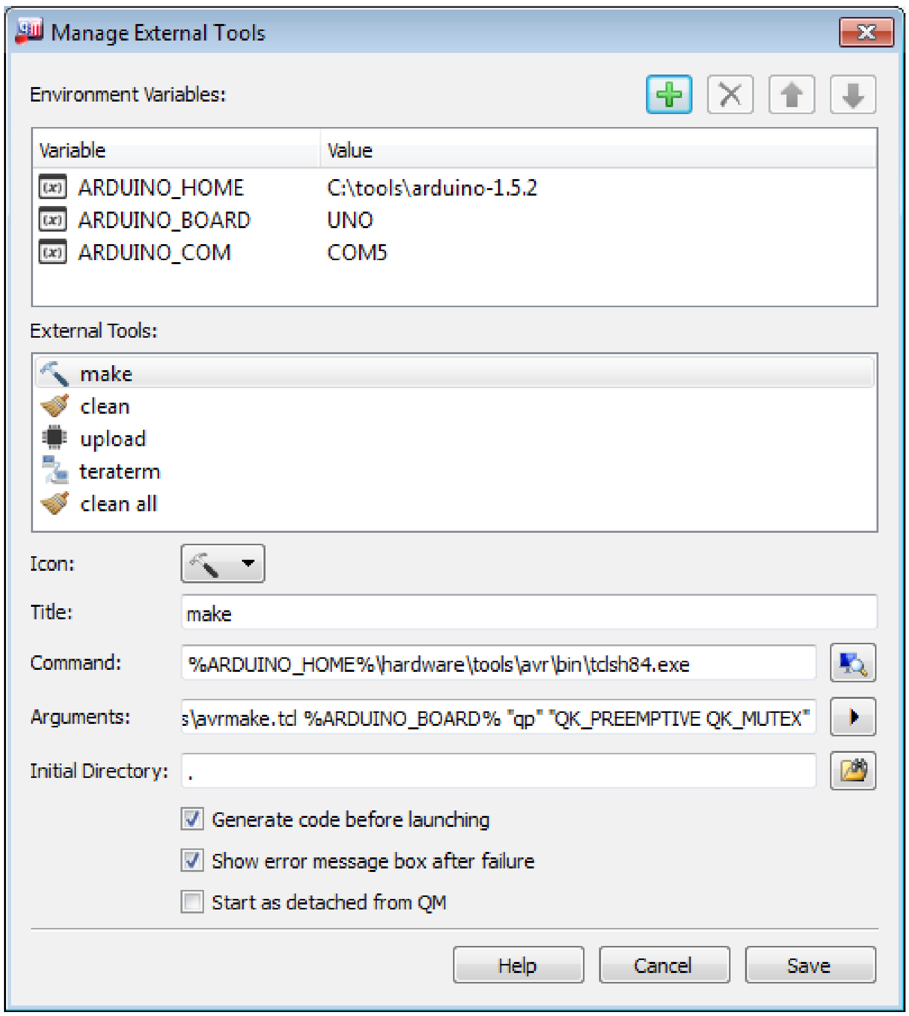
A preemptive kernel (panel (b) in Figure 13) gets control at the end of every ISR, which allows a preemptive kernel to return to a **different** task than the originally preempted one. If an ISR posts an event to a higher-priority task than currently executing, the preemptive kernel can return to the higher-priority task, which will service the event without waiting for any lower-priority processing.

A preemptive kernel can guarantee **deterministic** event responses posted to high-priority tasks, because the lower-priority tasks can be preempted. But this determinism comes a price of increased complexity and possibility of corrupting any shared resources among tasks. A preemptive kernel can perform a context switch at any point where interrupts are not disabled (so essentially between most machine instructions). Any such context switch might lead to corruption of shared memory or other shared resources, and a preemptive kernel usually provides special mechanisms (such as a mutex) to guarantee a mutually exclusive access to any shared resources.

7.1 Configuring the QP/C++ Library to Use Preemptive QK Kernel

Re-configuring the QP to use the preemptive QK kernel, instead of the simple “Vanilla” kernel, is very easy. You need to define the symbol `QK_PREEMPTIVE` to the `avrmake.tcl` build script. Additionally, if you want to use the QK mutex, you need to define the symbol `QK_MUTEX`. The following Listing 13 highlights the changes.

Figure 14: Defining the QK_PREEMPTIVE symbol in the arguments of the `avrmake.tcl` script (DPP-QK example)



7.2 Changes to the BSP for the Preemptive QK Kernel (dpp-qk Example)

The example `dpp-qk` located in the Arduino examples directory (see Listing 1) demonstrates the DPP application described in Section 2.5, running under the preemptive QK kernel. This example demonstrates the bsp.`cpp` for the preemptive QK kernel. The following Listing 13 highlights the changes.

> **NOTE:** Compared to the simple non-preemptive "Vanilla" kernel, the preemptive QK kernel requires **more stack** space. For more information about the QK kernel, please refer to Chapter 10 in the "Practical UML Statecharts" book and to the ESD article "Build the Super-Simple Tasker" (see Section Related Documents and References)

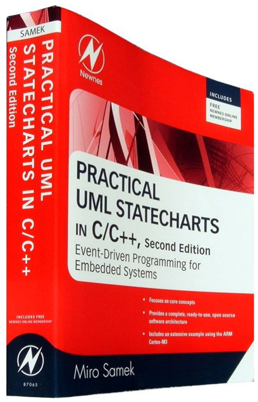
Listing 13: The bsp.cpp file for the dpp-qk example

```
// ISRs -----------------------------------------------------------------
ISR(TIMER2_COMPA_vect) {
    // No need to clear the interrupt source since the Timer2 compare
    // interrupt is automatically cleard in hardware when the ISR runs.

    QK_ISR_ENTRY();                      // inform QK kernel about entering an ISR

    QF::TICK(&l_TIMER2_COMPA);                    // process all armed time events

    QK_ISR_EXIT();                        // inform QK kernel about exiting an ISR
}

//............................................................................
void QK::onIdle() {

    QF_INT_DISABLE();
    USER_LED_ON();      // toggle the User LED on Arduino on and off, see NOTE1
    USER_LED_OFF();
    QF_INT_ENABLE();

#ifdef SAVE_POWER

    SMCR = (0 << SM0) | (1 << SE);  // idle sleep mode, adjust to your project
    __asm__ __volatile__ ("sleep" "\n\t" :: );
    SMCR = 0;                                              // clear the SE bit

#endif
}
. . .
```

As shown in Listing 13, every ISR for the preemptive QK kernel must invoke the macro `QK_ISR_ENTRY()` at the beginning (and before calling any QP function) and must invoke the macro `QK_ISR_ENTRY()` right before the end. These macros give control to the QK kernel, so that it can perform preemptions.

Additionally, the QK kernel handles the idle condition differently (actually simpler) than the non-preemptive "Vanilla" kernel. The idle callback for the QK kernel is called `QK::onIdle()` and the difference is that it is called with interrupts enabled, in contrast to the `QF::onIdle()` callback discussed before.

8 Related Documents and References

Document		Location
	"Practical UML Statecharts in C/C++, Second Edition" [PSiCC2], Miro Samek, Newnes, 2008	Available from most online book retailers, such as Amazon.com. See also: http://www.state-machine.com/psicc2.htm
QP Development Kits for Arduino, Quantum Leaps, LLC, 2011		http://www.state-machine.com/arduino
"Application Note: Dining Philosopher Problem Application", Quantum Leaps, LLC, 2008		http://www.state-machine.com/resources/AN_DPP.pdf
"Application Note: PEDestrian LIght CONtrolled (PELICAN) Crossing Application", Quantum Leaps, LLC, 2008		http://www.state-machine.com/resources/AN_PELICAN.pdf
"Using Low-Power Modes in Foreground/Background Systems", Miro Samek, Embedded System Design, October 2007		http://www.embedded.com/design/202103425
QP Development Kits for AVR, Quantum Leaps, LLC, 2008		http://www.state-machine.com/avr
"QP/C++ Reference Manual", Quantum Leaps, LLC, 2011		http://www.state-machine.com/doxygen/qpcpp/
Free QM graphical modeling and code generation tool, Quantum Leaps, 2011		http://www.state-machine.com/qm
"Build a Super-Simple Tasker", by Miro Samek and Robert Ward, ESD, 2006		http://www.eetimes.com/General/PrintView/4025691 http://www.state-machine.com/resources/samek06.pdf

9 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

Arduino Project

homepage:
http://arduino.cc