Quantum® LeaPs
innovating embedded systems

nano

Application Note:

Event-Driven Arduino Programming with QP™-nano and QM™

Document Revision Q
February 2017

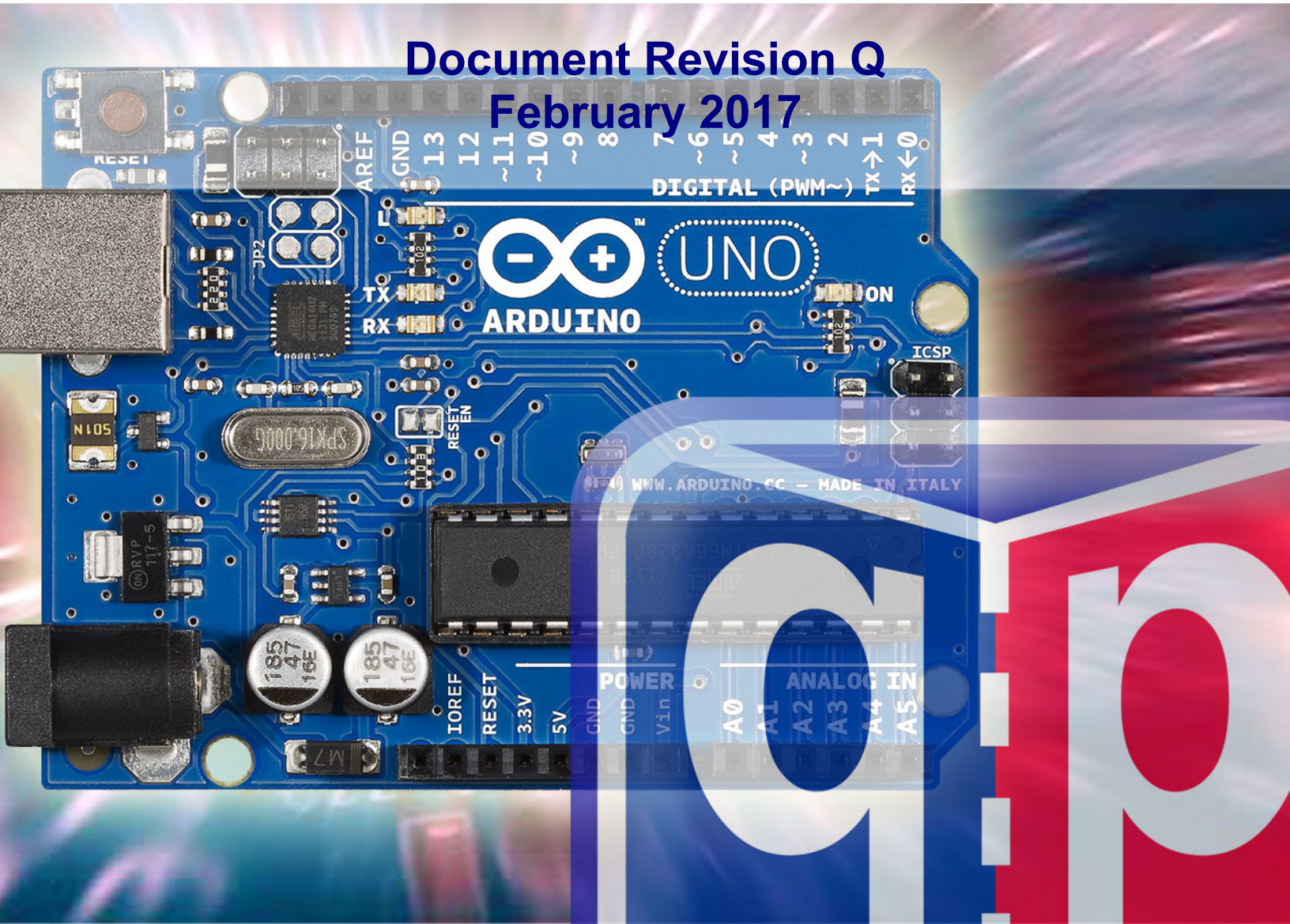


Table of Contents

1 Introduction 1
- 1.1 About Arduino 1
- 1.2 Event-Driven Programming with Arduino 2
- 1.3 QP™-nano Active Object Framework 3
- 1.4 QM™ Graphical Modeling Tool 5

2 Getting Started 6
- 2.1 Software Installation 7
- 2.2 Building the Examples in the Standard Arduino IDE 10
- 2.3 The Blinky Example 11
- 2.4 The PELICAN Crossing Example 14
- 2.5 The Dining Philosophers Problem Example 16

3 The Structure of an Arduino Sketch for QP™-nano 17
- 3.1 Include files 19
- 3.2 Events 19
- 3.3 Active Object declarations 19
- 3.4 Board Support Package 20
- 3.5 Interrupts 20
- 3.6 QP-nano Callback Functions 21
- 3.7 The Assertion Handler 21
- 3.8 Define the Active Objects (Generate the State Machine Code) 22

4 Working with State Machines 23

5 QP™-nano Library for Arduino 25
- 5.1 The qpn.h Header File 25
- 5.2 The qepn.c, qfn.c, and qvn.c Files 25

6 QM Tools for Arduino 26
- 6.1 Configuring The Environment Variables 27
- 6.2 The build_avr.tcl Build Script 28
- 6.3 Uploading Code to Arduino 29
- 6.4 The rm Utility for Cleaning Up the Build 29
- 6.5 The serialterm Utility for Displaying the Serial Output from Arduino 29

7 Related Documents and References 30

8 Contact Information 31

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 truly concurrent Arduino programs with the open source QP™-nano 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 free graphical QM™ modeling tool to draw state machine diagrams graphically and to **generate** Arduino code **automatically** from these diagrams. The QM™ modeling tool together with the build script provided in the accompanying code to this Application Note allow you to build and upload the Arduino sketches entirely from the QM tool.

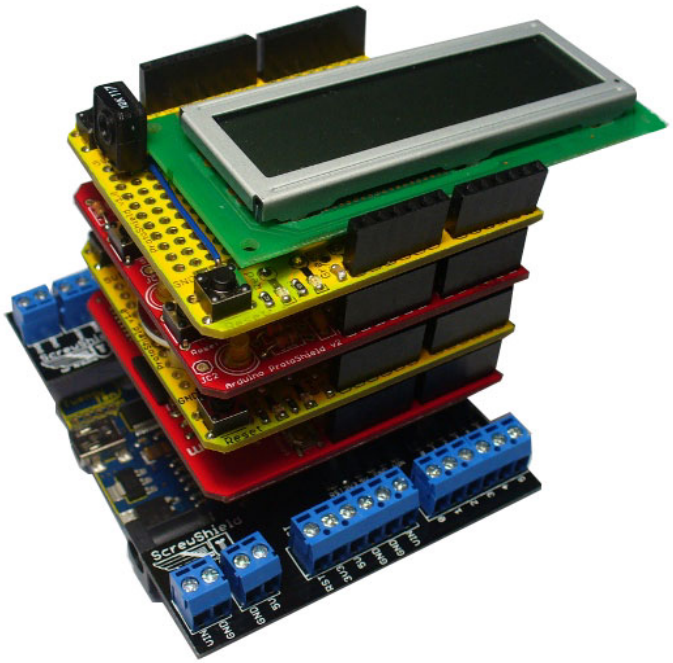
1.1 About Arduino

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++ and C languages (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 (the standard Blink Arduino code)

```
void loop() {
  digitalWrite(13, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);              // wait for a second
  digitalWrite(13, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);              // wait for a second
}
```

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 time delay), it is not doing any other work and is **not responsive** to other events (e.g., button presses).

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 (no waiting!)
    if (event2())          // event2 occurred?
        event2Handler(); // process event2 (no waiting!)
    . . .                  // 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 QP™-nano framework described in the next section allows you to combine the event-driven programming paradigm with modern state machines.

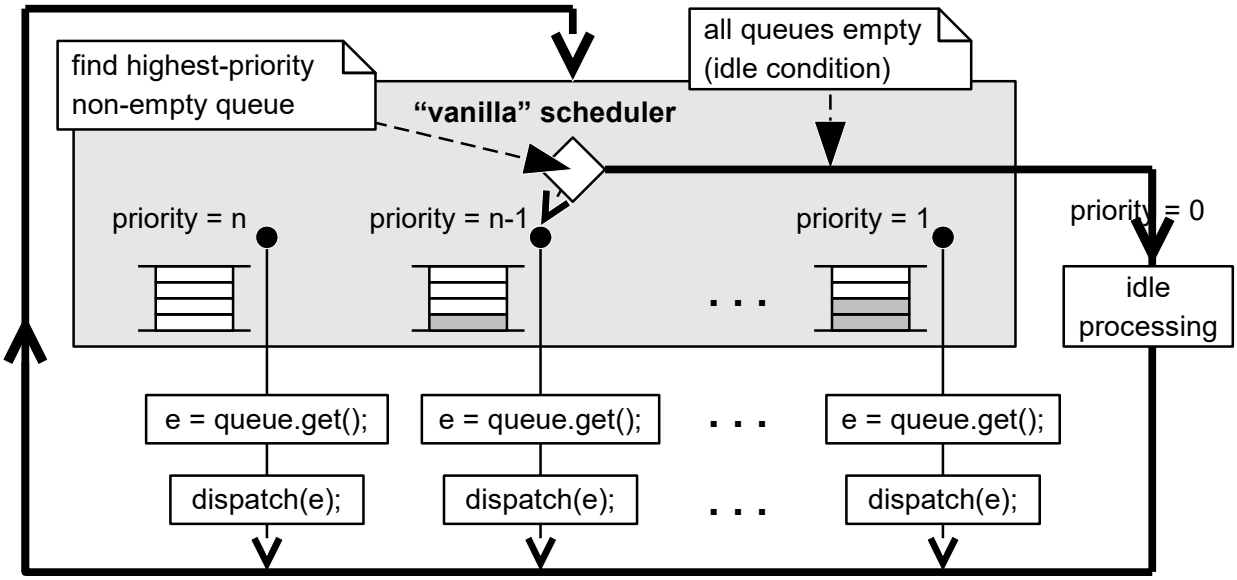
1.3 QP™-nano Active Object Framework

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™-nano framework, 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 *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 + a unique priority is collectively called an **Active Object**.

Figure 2: Event-driven QP-nano framework with multiple event queues and state machines (Active Objects)



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.

> **NOTE:** The scheduler shown in Figure 2 is an example of a **cooperative** scheduler (called QV-nano), because state machines naturally cooperate to implicitly yield to each other at the end of each run-to-completion step. The full-version of the QP-nano framework contains also a more advanced, fully **preemptive** real-time kernel called QK-nano. However, for simplicity, the QP-nano development kit for Arduino currently does not provide this kernel.

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 `QV_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.

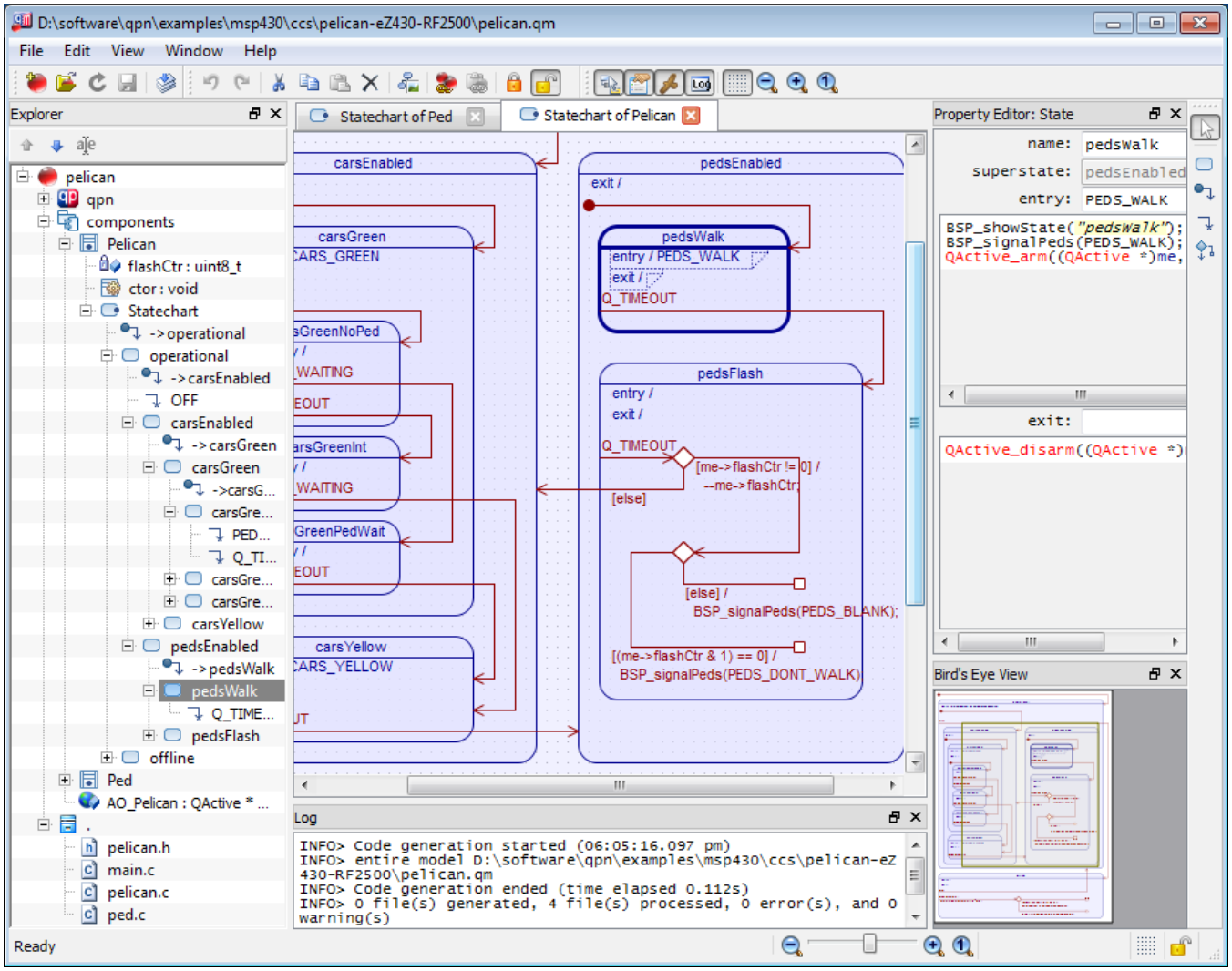
Finally, please also note that the framework shown in Figure 2 can achieve good real-time performance, because the individual run-to-completion (RTC) steps of each state machine are typically short (execution time counted in microseconds).

1.4 QM™ Graphical Modeling Tool

QM™ (Quantum Modeler) is a free, cross-platform, graphical modeling tool for designing and implementing real-time embedded applications based on the QP™ state machine frameworks. QM™ is available for Windows 64-bit, Linux 64-bit, and Mac OS X.

QM™ provides intuitive diagramming environment for creating good looking hierarchical state machine diagrams and hierarchical outline of your entire application. QM™ eliminates coding errors by automatic generation of compact C or C++ code that is 100% traceable from your design. Please visit state-machine.com/qm for more information about QM™.

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



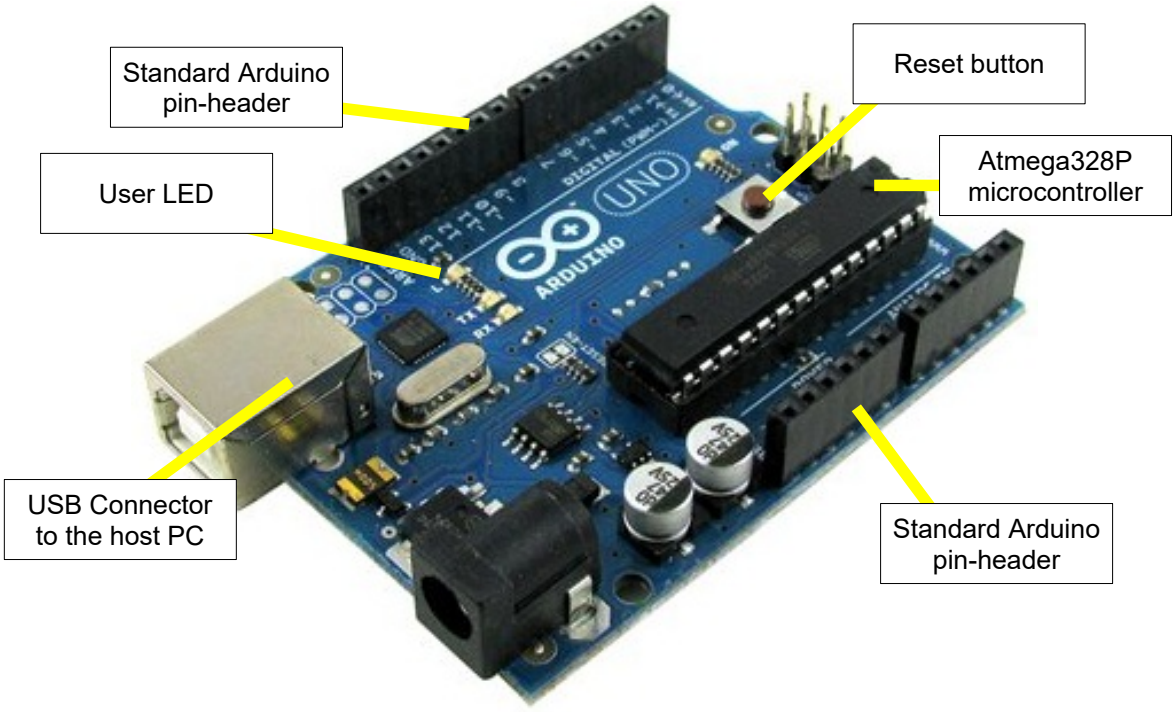
2 Getting Started

The example code has been built entirely from the **QM modeling tool** (version 4.x or higher), but it uses the compiler and libraries in the standard **Arduino 1.8.x** (the latest as of this writing), which is available for a free download from the Arduino website.

NOTE: The tools and procedures for building and uploading Arduino sketches **will not work with the Arduino 1.0.x** toolset, because it has a very different structure than Arduino 1.8.x.

To focus the discussion, this Application Note uses the **Arduino UNO** board based on the Atmel Atmega328p microcontroller (see Figure 4).

Figure 4: Arduino UNO board



NOTE: The following discussion assumes that you have downloaded and installed both the standard Arduino software for **Windows** and the QM tool on your computer. The free QM tool can be downloaded from: https://sourceforge.net/projects/qpc/files/QM.

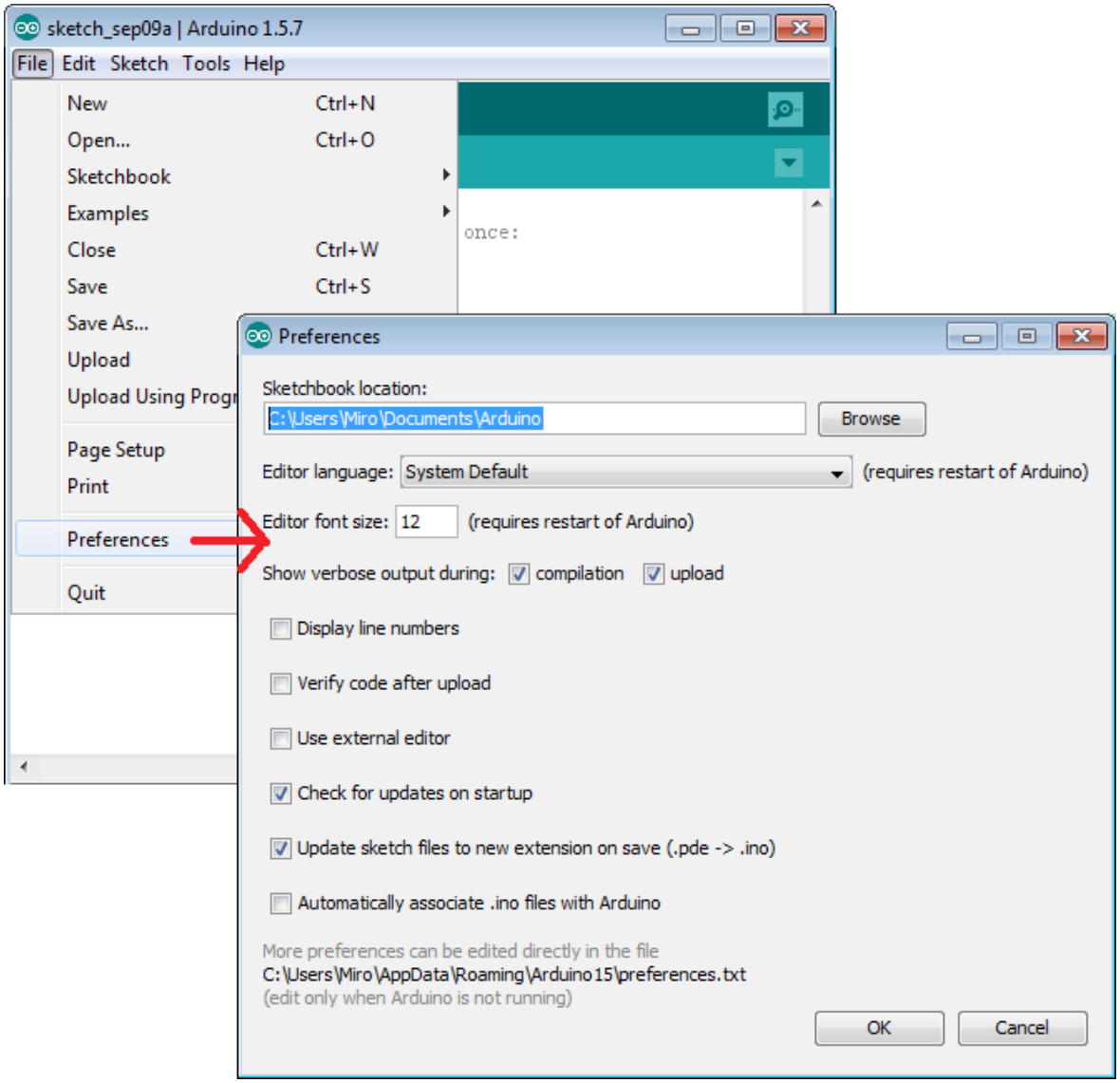
NOTE: Currently, the provided software has been tested only on **Windows**.

2.1 Software Installation

QP-nano for Arduino is distributed in a single ZIP archive `qpn-<ver1>_arduino-<ver2>.zip`, where `<ver1>` stands for the QP-nano version and `<ver2>` for the version of the Arduino software (e.g., 1.8.x).

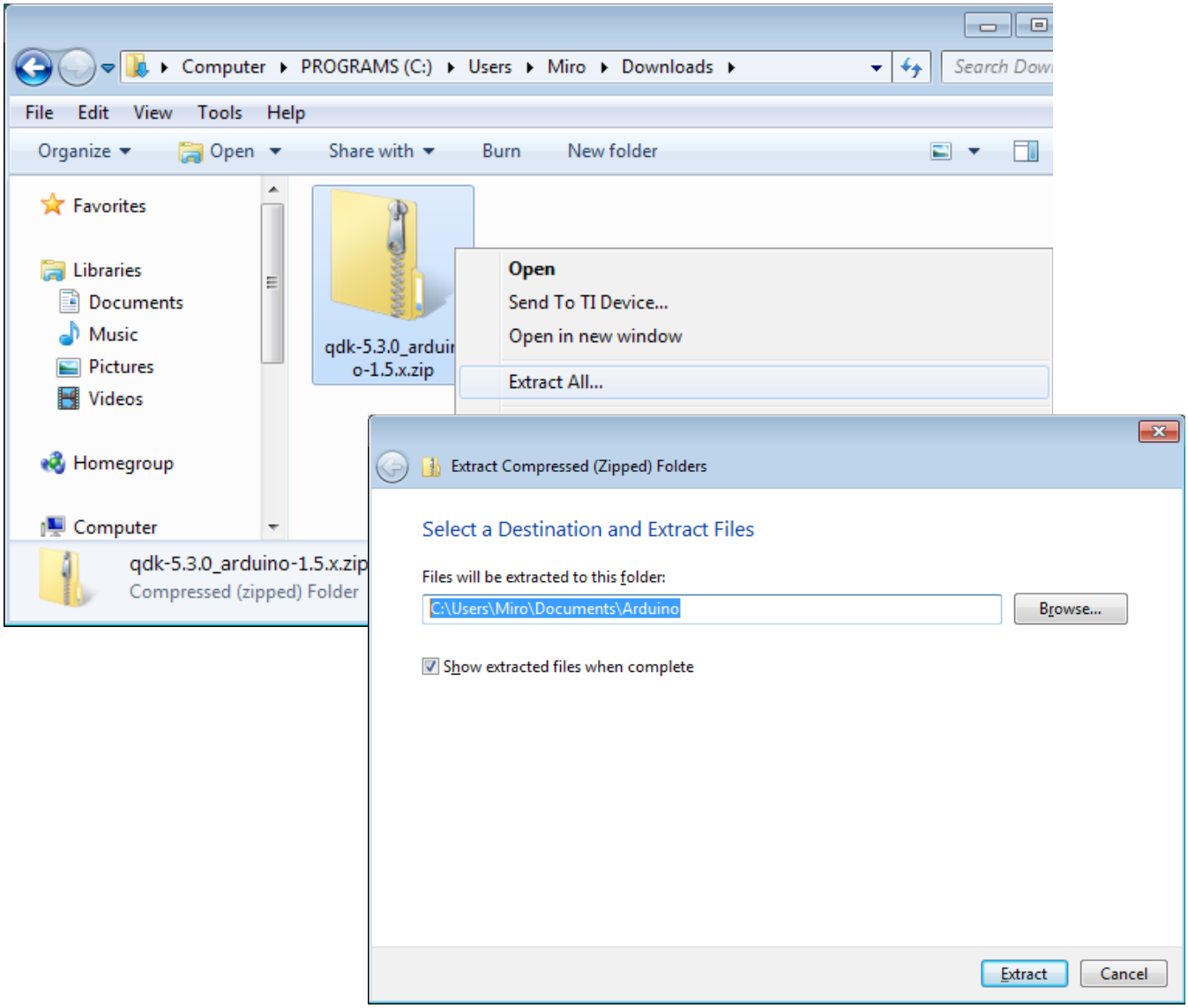
You need to unzip the `qpn-<ver1>_arduino-<ver2>.zip` archive into your **Arduino-Sketchbook folder**. To find out where your Sketchbook folder is, or to configure a different location for your Sketchbook folder, you need to open the Arduino IDE and select **File | Preferences** menu (see Figure 6). The sketchbook location will be shown at the top of the **Preferences** dialog box.

Figure 5: Finding/configuring Arduino-Sketchbook location



Once you identify the Sketchbook folder, you simply unzip the whole archive to your Sketchbook (see Figure 6).

Figure 6: Unzipping `qpn-<ver1>_arduino-<ver2>.zip` into the Arduino Sketchbook.



The contents of the archive unzipped into your Sketchbook folder is shown in Listing 3.

Listing 3: Contents of the qpn-<ver1>_arduino-<ver2>.zip archive for Arduion 1.8.x

```
<Arduino_Sketchbook>    - Your Arduino Sketchbook folder
+-doc/                  - documentation
| +-AN_Event-Driven_Arduino_QP-nano.pdf - this document
| +-AN_DPP.pdf          - Dining Philosopher Problem example application
| +-AN_PELICAN.pdf      - PEdestrian LIght CONtrolled (PELICAN) crossing example
|
+-libraries/
| +-qpn_avr/            - QP-nano library for AVR-based Arduinos
| | +-examples/
| | | +-blinky/         - Very simple "blinky" example
```

```
| | | | +-.blinky      - The QM session file for the Blinky project
| | | | +-blinky.qm    - The QM model of the Blinky application
| | | +-dpp/           - Dining Philosophers Problem (DPP) example
| | | | +-.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
| | |
| | +-qepn.c           - QEP-nano component platform-independent implementation
| | +-qfn.c            - QF-nano  component platform-independent implementation
| | +-qpn.h            - QP-nano API (the file to be included in applications)
| | +-qvn.c            - QV-nano cooperative kernel implementation
|
+-tools/
| +-scripts/           - tools contributed Quantum Leaps
| | +-build_avr.tcl    - Generic TCL script for building Arduino/AVR sketches
| | +-upload_avr.bat   - Batch file for uploading sketches to the Arduino/AVR board
| +-share/             - Folder for the TCL interpreter
| | +-tcl8.4/          - facilities for TCL 8.4
| | +-. . .
| +-utils/             - tools contributed Quantum Leaps
| | +-COPYING.txt      - terms of copying this code
| | +-GPL2.TXT         - GPL version 2 open source license
| | +-TCL_LICENSE.TXT  - License for the TCL interpreter
| | +-cp.exe           - Command-line utility for copying files
| | +-rm.exe           - Command-line utility for removing files (cleanup)
| | +-serialterm.exe   - Command-line serial terminal
| | +-tcl84.dll        - DLL for the TCL interpreter
| | +-tclpip84.dll     - DLL for the TCL interpreter
| | +-tclsh84.exe      - TCL Shell to execute TCL scripts
| | +-tclsh.exe        - TCL Shell to execute TCL scripts (alias)
|
+-GPLv3.txt            - GNU General Public License version 3
+-QPn-Arduino_GPL_Exception.txt - GPLv3 exception for Arduino
+-README.txt           - README file with basic instructions
```

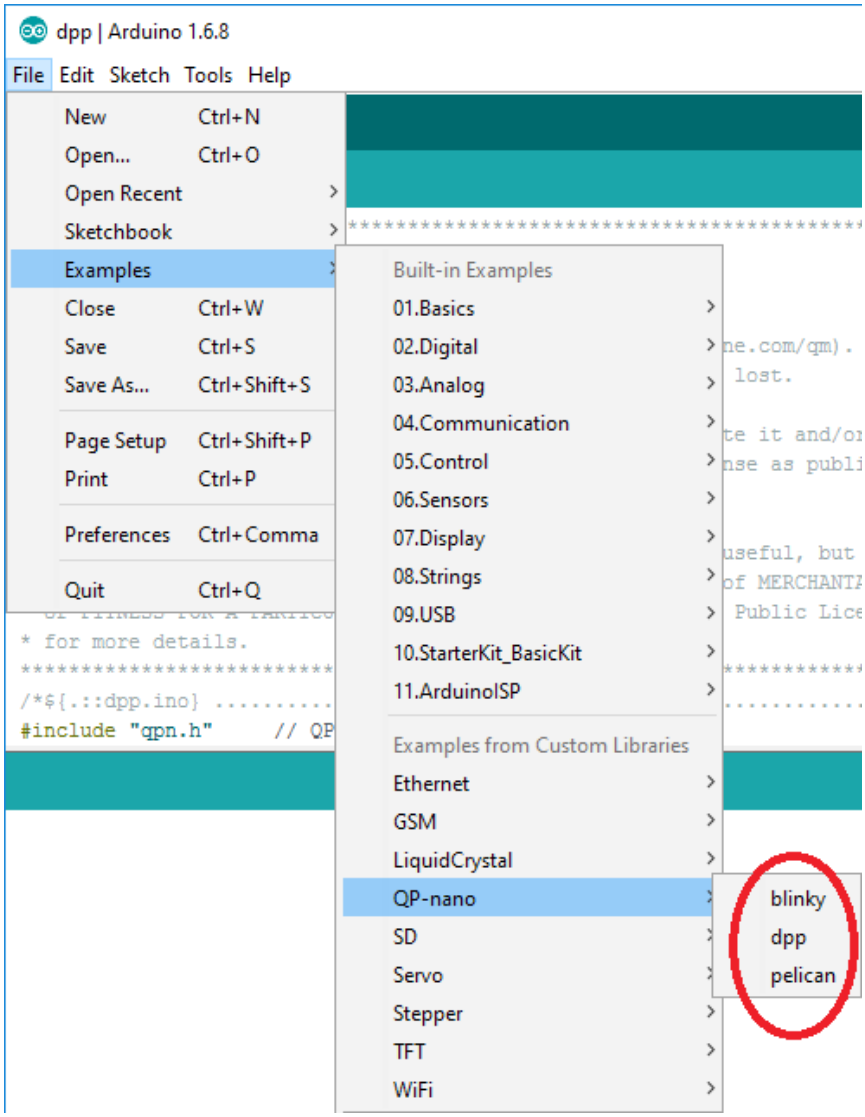
Each QP-nano example for Arduino (in the `library\qpn_avr\examples` folder) contains a QM model, which is a file with the extension `.qm`., such as `<Sketchbook>\library\qpn_avr\examples\blinky\blinky.qm`, see Listing 3). These models and the QM modeling tool take Arduino programming to the next level. 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 8).

> **NOTE:** To start working with the QM™ modeling tool, you need to download the tool from **SourceForge.net repository** . 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 the Examples in the Standard Arduino IDE

The examples accompanying the QP-nano library 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. The following screen shot shows how to open the examples.

Figure 7: Opening the QP-nano Examples from the Standard Arduino IDE



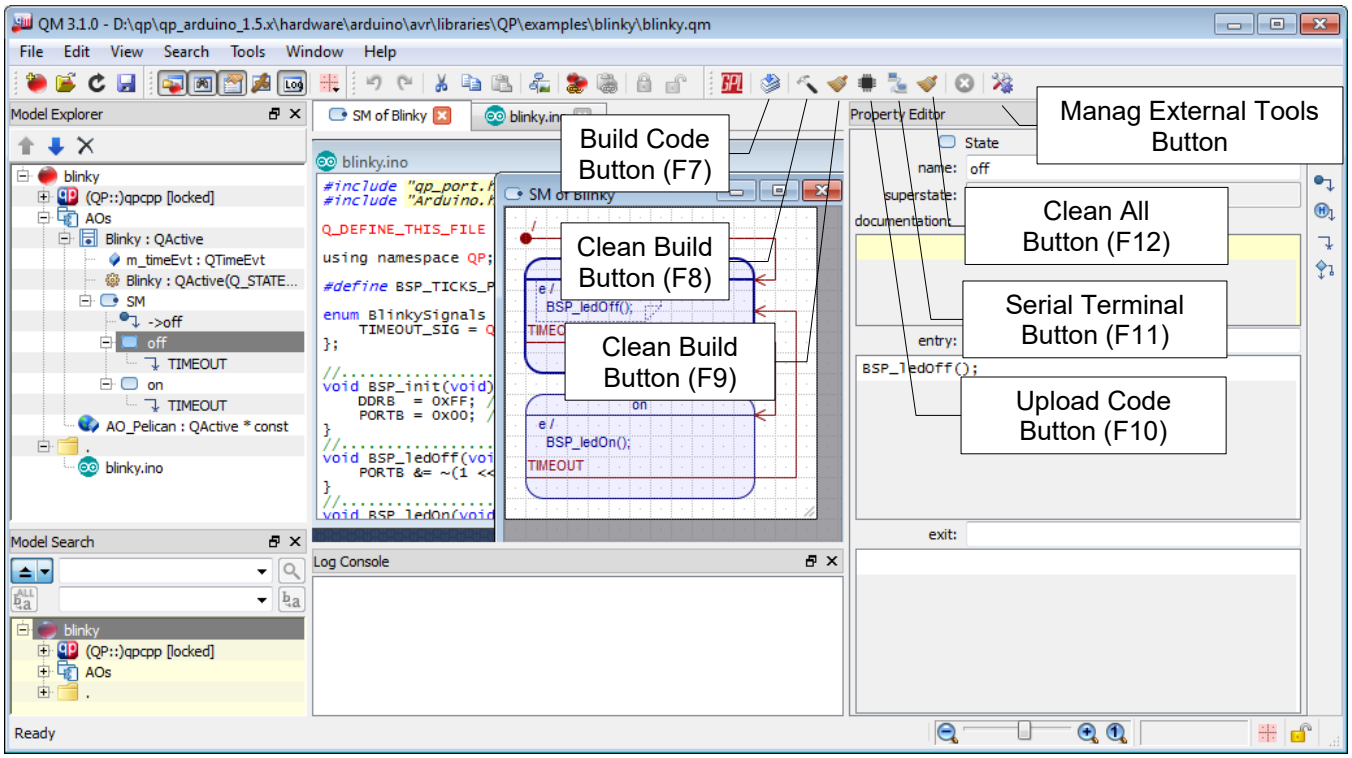
NOTE: The Standard Arduino IDE is **not** the most convenient way of managing automatically generated code, because you need to re-generate the sketch code (the `.ino` file) every time you change anything in the grahpical model. Therefore the following sections describe how to build and download the code to Arduino **directly from the QM modeling tool**.

2.3 The Blinky Example

To build the provided Blinky example, change to the `<Arduino_Sketchbook>\library\qpn_avr\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 8. **You build and upload the QP examples directly from the QM modeling tool**, without the need to launch the standard Arduino IDE.

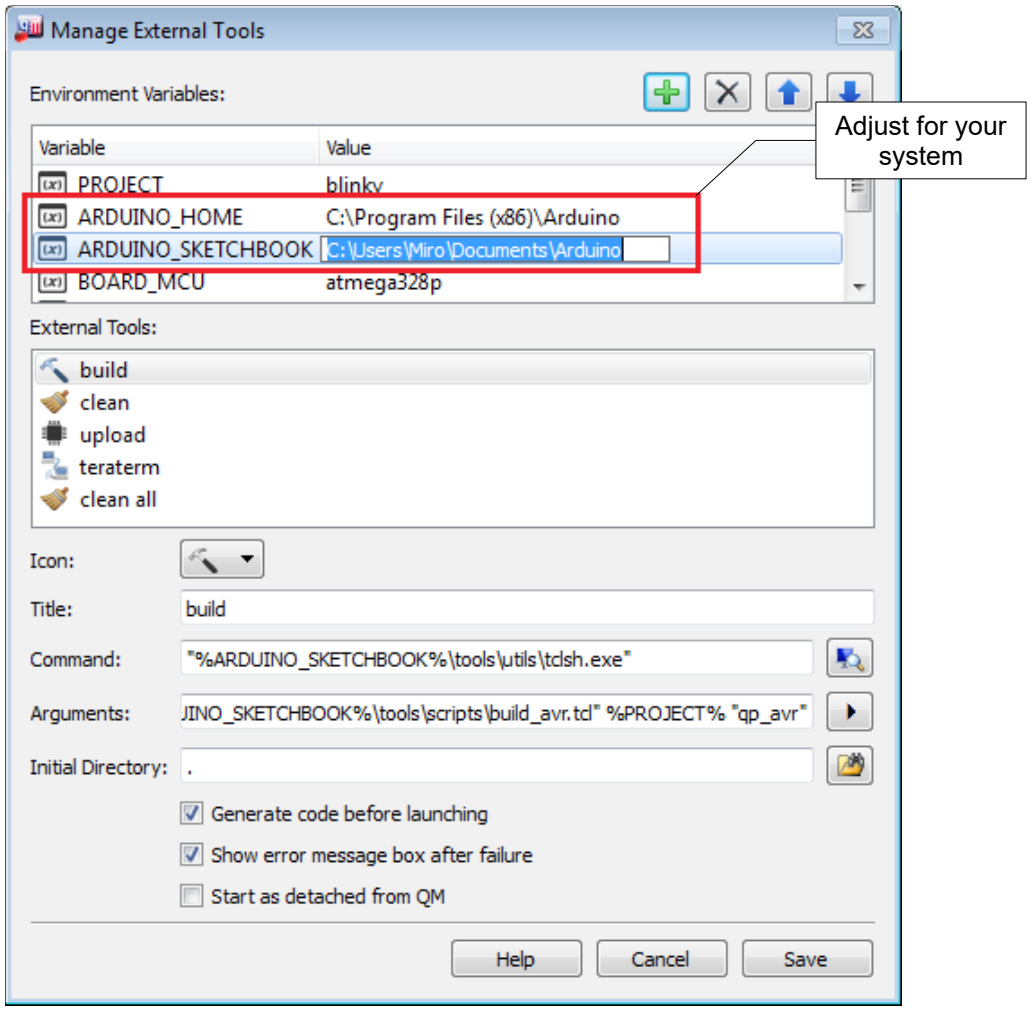
NOTE: Even though you are not using the Standard Arduino IDE in this case, you still need to install it, because to build your software the QM tool needs the compiler, linker, and the standard Arduino libraries to build your code.

Figure 8: 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 8) and edit the environment variables `ARDUINO_HOME` and `ARDUINO_SKETCHBOOK` to the location of the standard Arduino software and your Arduino-Sketchbook, respectively. You can also edit the parameters of your Arduino board (`BOARD_MCU`, `BOARD_VARIANT`, and `BOARD_F_CPU`) as well as the COM port used by your board (`BOARD_COM` and `BOARD_BAUD`). Please refer to Section 6 for more details.

Figure 9: Adjusting the ARDUINO_HOME and ARDUINO_SKETCHBOOK environment variables in the Manage External Tools dialog box

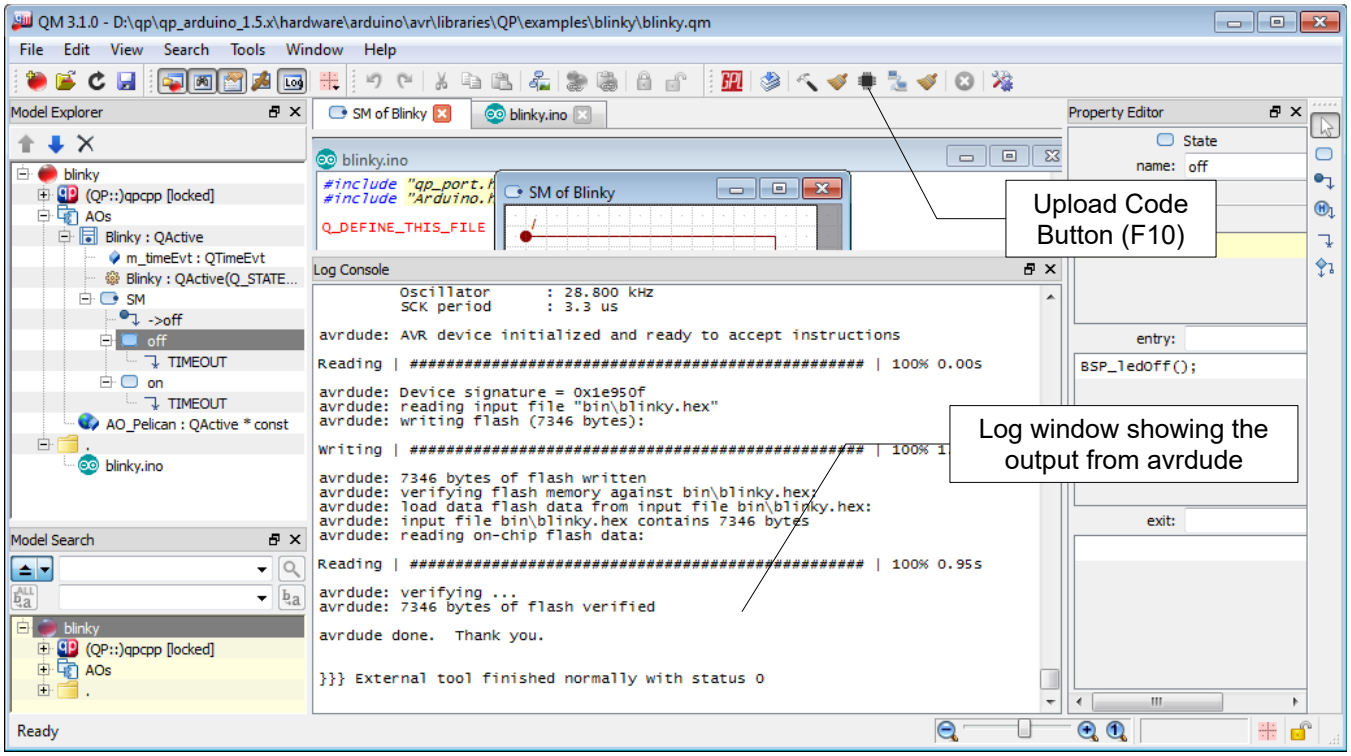


NOTE: The Manage External Tools dialog box allows you to configure up to five different external tools. The described configuration is included in the QP examples for Arduino, but you can modify the tools to suit your needs at any time

After you've adjusted the environment variables, click on the **Build** button (see Figure 8). 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-nano framework for AVR (`"qpn_avr"`). 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 clicking on 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 4).

Figure 10: Uploading the code to the Arduino board



2.4 The PELICAN Crossing Example

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_Sketchbook>\library\qpn_avr\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 11) 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).

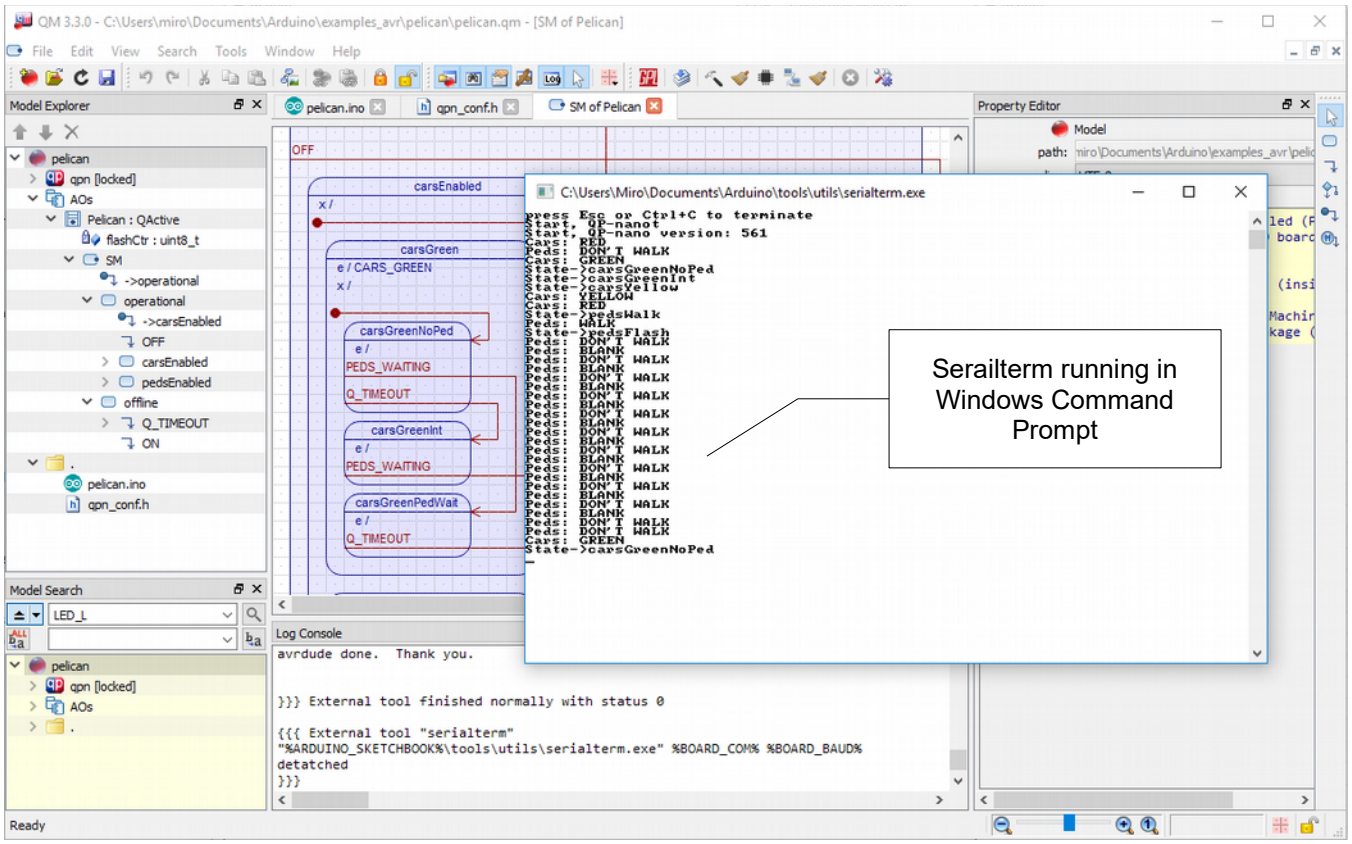
Figure 11: Pedestrian Light CONtrolled (PELICAN) crossing

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 8). 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 12.

NOTE: The Serial Terminal button opens the **serialterm** application, which a freeware command-line serial terminal application for Windows that is provided in the `tools\utils` sub-directory.

Figure 12: Serialterm running in a Windows console with the output from the PELICAN crossing example



NOTE: The **serialterm** application runs "detached" from QM in a separate window. Before uploading your code to Arduino again, you need to close the serialterm window by pressing the **ESC key**, to free the serial connection to the Arduio board.

2.5 The Dining Philosophers Problem Example

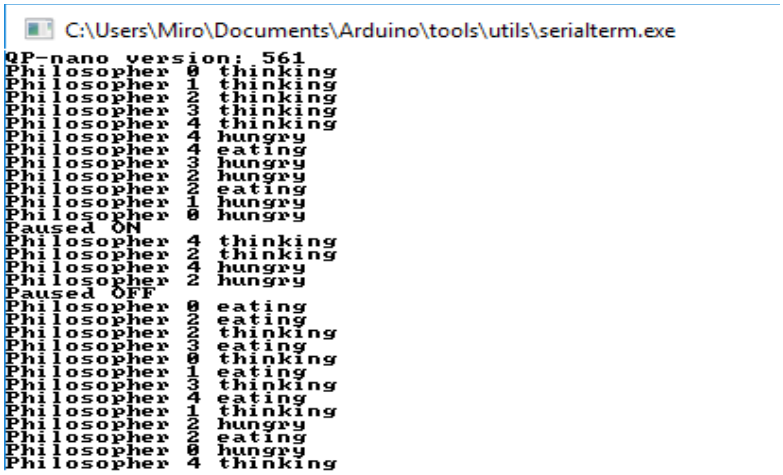
Finally, the example code contains the Dining Philosophers Problem (DPP) example, which demonstrates multiple active objects This example is located in the directory `<Arduino_Sketchbook>\library\qpn_avr\examples\dpp`.

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 13). 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 13: Dining Philosophers Problem

You build and upload the DPP example to the Arduino UNO board the same way as the PELICAN example. Just like in the PELICAN example, the User LED in DPP 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 example, you need to open a **Serial Terminal** by pressing the Serial Terminal button on the QM toolbar (see Figure 8). For the Arduino UNO board, the Serial Terminal should be configured to **115200 baud rate**.



3 The Structure of an Arduino Sketch for QP™-nano

Every Arduino sketch (the `.ino` file) for QP™-nano typically consists of seven groups:

1. Include files,
2. Events,
3. Active Object declarations (generated code),
4. Board Support Package (including the Arduino `setup()` and `loop()` functions),
5. Interrupts
6. QP™-nano callback functions, and
7. Active object definitions (generated code)

NOTE: An QP-nano application can be structured in many different ways and can be broken down into multiple files. The structure described here is only a recommendation for a typical Arduino "sketch" of low to moderate complexity.

To focus the discussion, the following Listing 4 shows the PELICAN sketch as an example (see Section 2.4), but the discussion applies to all sketches for QP™-nano. The explanation sections immediately following Listing 4 discuss the specific code groups in greater detail.

NOTE: The following listing shows the sketch code as it is viewed in the QM™ modeling tool. The actual sketch file saved by QM on disk is much bigger, because it contains all original code plus the code **generated** by QM™.

Listing 4: Typical Arduino sketch for QP™-nano (file `pelican.ino`)

```
[1] #include "qpn.h"      // QP-nano framework
[2] #include "Arduino.h" // Arduino API

    //============================================================================
[3] enum PelicanSignals {
        PEDS_WAITING_SIG = Q_USER_SIG,
        OFF_SIG,
        ON_SIG
    };

    //============================================================================
    // declare all AO classes...
[4] $declare(AOs::Pelican)
    //...

    // define all AO instances and event queue buffers for them...
[5] Pelican AO_Pelican;
[6] static QEvt l_pelicanQSto[10]; // Event queue storage for Pelican
    //...

    //============================================================================
    // QF_active[] array defines all active object control blocks ----------------
[7] QMActiveCB const Q_ROM QF_active[] = {
        { (QMActive *)0,           (QEvt *)0,        0U                          },
        { (QMActive *)&AO_Pelican, l_pelicanQSto,    Q_DIM(l_pelicanQSto)        }
```

```
    };

    //============================================================================
[8] // Board Support Package (BSP)
    // various other constants for the application…
    enum {
        BSP_TICKS_PER_SEC = 100, // number of system clock ticks in one second
        LED_L             = 13,  // the pin number of the on-board LED (L)
        PHILO_0_PRIO      = 1,   // priority of the first Philo AO
        THINK_TIME        = 3*BSP_TICKS_PER_SEC, // time for thinking
        EAT_TIME          = 2*BSP_TICKS_PER_SEC  // time for eating
    };
    //...

    //............................................................................
[9] void setup() {

        // initialize the QF-nano framework
[10]    QF_init(Q_DIM(QF_active));

        // initialize all AOs...
[11]    QActive_ctor(&AO_Pelican.super, Q_STATE_CAST(&Pelican_initial));
        //...

        // initialize the hardware used in this sketch...
        pinMode(LED_L, OUTPUT); // set the LED-L pin to output

        Serial.begin(115200);   // set the highest stanard baud rate of 115200 bps
        //...
    }
    //............................................................................
[12]void loop() {
        QF_run(); // run the QP-nano application
    }

    //============================================================================
    // interrupts
[13]ISR(TIMER2_COMPA_vect) {
[14]    QF_tickXISR(0); // process time events for tick rate 0

        if (Serial.available() > 0) {
            switch (Serial.read()) { // read the incoming byte
                case 'p':
                case 'P':
                    QACTIVE_POST_ISR((QMActive *)&AO_Pelican, PEDS_WAITING_SIG, 0U);
                    break;
                //...
            }
        }
    }

    //============================================================================
    // QF callbacks...
[15]void QF_onStartup(void) {
        // set Timer2 in CTC mode, 1/1024 prescaler, start the timer ticking...
        TCCR2A = (1U << WGM21) | (0U << WGM20);
```

```
        TCCR2B = (1U << CS22 ) | (1U << CS21) | (1U << CS20); // 1/2^10
        ASSR  &= ~(1U << AS2);
        TIMSK2 = (1U << OCIE2A); // enable TIMER2 compare Interrupt
        TCNT2  = 0U;

        // set the output-compare register based on the desired tick frequency
[16]    OCR2A  = (F_CPU / BSP_TICKS_PER_SEC / 1024U) - 1U;
    }
    //............................................................................
[17]void QV_onIdle(void) {   // called with interrupts DISABLED
        // Put the CPU and peripherals to the low-power mode. You might
        // need to customize the clock management for your application,
        SMCR = (0 << SM0) | (1 << SE); // idle mode, adjust to your project
[18]    QV_CPU_SLEEP();  // atomically go to sleep and enable interrupts
    }
    //............................................................................
[19]void Q_onAssert(char const Q_ROM * const file, int line) {
        // implement the error-handling policy for your application!!!
        QF_INT_DISABLE(); // disable all interrupts
        QF_RESET();  // reset the CPU
    }

    //============================================================================
    // define all AO classes (state machines)...
[20]$define(AOs::Pelican)
    //...
```

3.1 Include files

[1] Each sketch must include the QP-nano library (the "`qpn.h`" header file).

[2] Each sketch for must include the standard **"`Arduino.h`"** header file. This is necessary to make the sketch compatible with the `build_avr.tcl` build script (see also Section 6.2)

3.2 Events

[3] All event signals used by the application are enumerated. Please note that all the signal names must end with the `_SIG` suffix and that the very first signal must be equal to the constant `Q_USER_SIG`.

3.3 Active Object declarations

[4] All active classes (subclasses of the `QActive` base class) in the application must be declared by means of the **"$declare()"** directives. This directive instructs the QM modeling tool to generate a declaration of the model element specified in the parentheses, such as the `AOs::Pelican` active class in this case. The active class declarations are subsequently used to define active objects (instances of the active classes)

[5] All active objects are defined as instances of the active classes declared in step [4]. Please note that you can instantiate more than one active object from a given class. (The DPP example illustrates this by instantiating five `AO_Philo` active objects from one `Philo` class)

[6] Each active object instance needs an event queue, so you need to provide the properly sized queue for each active object.

[7] The global `QF_active[]` array contains pointers to all active object instances and their event queues in the system. The order of placing the active objects in the array determines their **priority** (from low priorities to high priorities).

3.4 Board Support Package

[8] The Board Support Package (BSP) contains all board-related code, which consists of various constants, the board initialization, and all functions that depend on the specific peripherals used by the application.

NOTE: Experience shows that it is very often advantageous to group all board-related functionality in one place (in the BSP) and access it through a well-defined set of functions (called API). That way, the state machine code can be re-used on different boards without any changes. For instance, in the PELICAN example you might connect the signals for cars and pedestrians to different pins of the board.

---

[9] The board initialization is accomplished here with standard Arduino function `setup()`, which initializes the Arduino board for this application.

[19] The `setup()` function must initialize the QP-nano framework by calling the `QF_init()` function. The argument passed to this function is the number of active objects that the framework needs to manage, which is the dimension of the `QF_active[]` 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.

---

[11] The `setup()` function must initialize all active object instances in the project. This initialization is accomplished by calling the QP-nano function `QActive_ctor()`. For every active object instance. The second argument of the function call is always in the form `Q_STATE_CAST(&<Active-Class>_initial)`, where `<Active-Class>` is the active class of the active object instance being initialized.

[12] The standard Arduino function `loop()` simply passes control the QP-nano framework by calling `QF_run()`. `QF_run()` runs your application and never returns.

3.5 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-nano. At the minimum, you must provide the **system clock tick** interrupt, which allows the QP-nano 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-nano 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.

[13] This example uses the Timer2 as the source of the periodic clock tick interrupt (Timer1 is already used to provide the Arduino `milli()` service). The Interrupt Service Routine is not a regular C

function and therefore must be defined by means of the macro `ISR()` provided in the C compiler for AVR.

[14] The ISR calls the QP-nano service `QF_tickXISR(0)`, which processes all time events of the QP-nano framework.

NOTE: Only very specific QP-nano services are allowed to be called from the ISR context. Besides `QF_tickISR()`, the other service is `QACTIVE_POST_ISR()`. Please note the "ISR" suffix in each case.

3.6 QP-nano Callback Functions

The QP-nano framework provides a few "callback functions", which the framework calls, but that are application-specific, so they need to be defined in **your** code.

[15] The callback function `QF_onStartup()` is called from `QF_run()` right before the QP-nano framework enters its event loop (see Figure 2). At this point, all active objects and BSP are already initialized and ready to go, so the application is ready to accept interrupts. So, the main purpose of the `QF_onStartup()` callback is to configure and start interrupts. Here the Timer2 peripheral is configured to provide a periodic interrupt.

[16] The `OCR2` register is loaded with a value that will case the desired number of interrupts per second `BSP_TICKS_PER_SEC`.

[16] The `QF_onCleanup()` callback register is loaded with a value that will case the desired number of interrupts per second `BSP_TICKS_PER_SEC`.

[17] When no events are available, the non-preemptive QV kernel invokes the platform-specific callback function `QV_onIdle()`, which you can use to save CPU power, or perform any other "idle" processing.

[18] When all event queues are empty, no active object will run, so only an external interrupt can provide new events. Therefore, it is an ideal time to put the CPU to a low-power sleep mode, from which it will be woken up by an external interrupt. The transition to a low-power sleep mode is accomplished atomically by means of the `QV_CPU_SLEEP()` macro (provided in QP-nano port to AVR).

NOTE: The article "Using Low-Power Modes in Foreground / Background Systems" (http://www.embedded.com/design/202103425) explains why a transition to low-power sleep mode should be atomic.

3.7 The Assertion Handler

As described in Chapter 6 of the book "Practical UML Statecharts in C/C++, Second Edition" [PSiCC2] (see Related Documents and References), all QP-nano 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.

[19] The callback function `Q_onAssert()` is called when the QP-nano framework encounters a program error. The function gives you the last chance to control the damage, but it should **not return**, because the program is not capable to continue. Typically, the last action performed in `Q_onAssert()` is to **reset** the CPU to start from the beginning.

3.8 Define the Active Objects (Generate the State Machine Code)

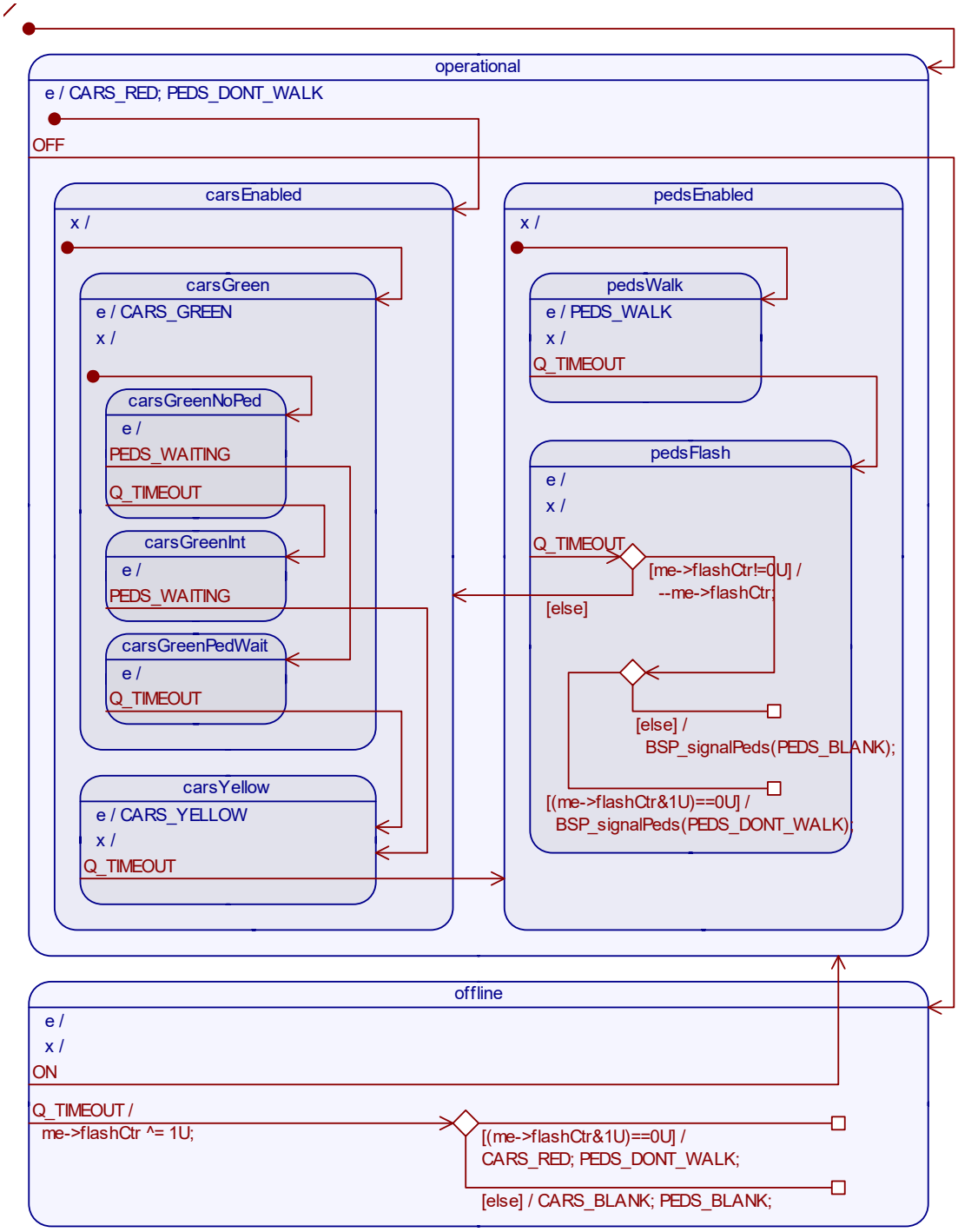
In the last section of your Arduino "sketch" you need to define all your active object classes. ("Define" means to provide the actual code). You accomplish this by means of the QM directive **`$define()`**, which generates the code for the specified model element.

[20] Use the `$define()` directive for all active objects in your project to generate code for them.

4 Working with State Machines

The QP-nano framework allows you to work with the modern **hierarchical** state machines (a.k.a., UML statecharts). For example, Figure 14 shows the HSM diagram for the PELICAN crossing.

Figure 14: 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-hierarchical 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.

5 QP™-nano Library for Arduino

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

5.1 The `qpn.h` Header File

When you import the library (Arduiono IDE, menu Sketch | Import Library | QP-nano), the Arduino IDE will insert the line "`#include "qpn.h"`" into your currently open sketch file. The `qpn.h` file contains the adaptations (port) of QP-nano to the AVR processor followed by the platform-independent code for QP-nano. Typically, you should not need to edit this file, except perhaps when you want to change some configuration parameters of the framework.

5.2 The `qepn.c`, `qfn.c`, and `qvn.c` Files

These source files in C contain the platform-independent code of the QP-nano framework, which contains the facilities for executing hierarchical state machines, queuing events, handling time, etc. You should not edit these source files.

6 QM Tools for Arduino

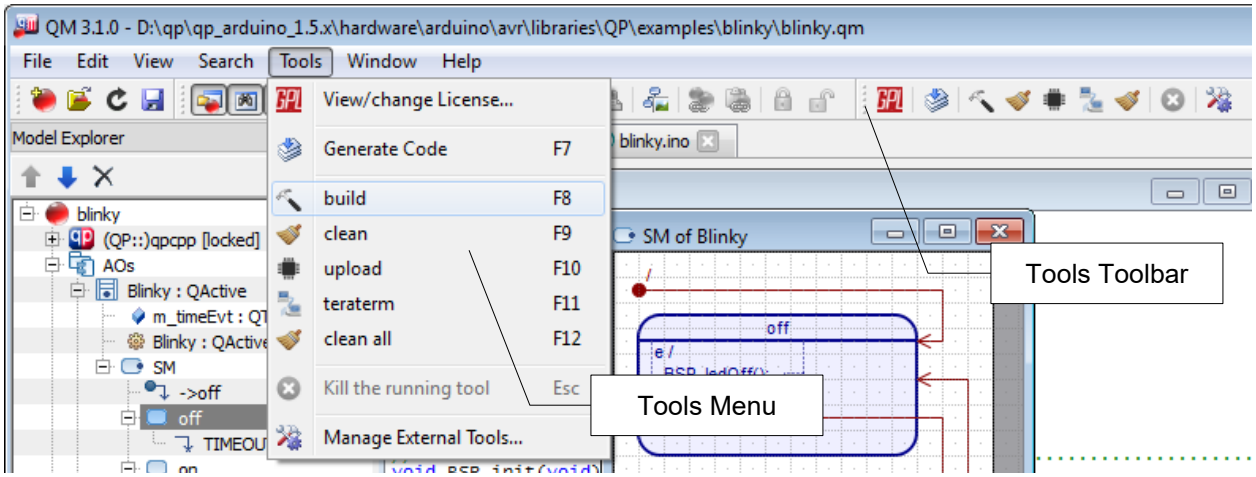
The QDP/Arduino ZIP file comes with some special tools, which among others allow you to build the Arduino projects (sketches in the Arduino talk) and upload the code to the Arduino board directly from the QM tool. This section describes how to use the `build_avr.tcl` build script as well as the `upload_avr.bat` utility directly from the QM modeling tool.

Listing 5: The tools provided in the qp_arduino_<var>.zip file

```
<Arduino_Sketchbook>    - Your Arduino Sketchbook folder
+-tools/                - special tools for QM
| +-scripts/            - scripts
| | +-build_avr.tcl     - TCL script for building Arduino/AVR sketches
| | +-upload_avr.bat    - Batch file for uploading AVR sketches
```

The QM modeling tool supports executing external programs directly from QM by means of the the "Tools" menu and the Tools-toolbar. As shown in Figure 15, the Tools are customized for each model and all example projects for Arduino contain the pre-configured "Tools" designed specifically for Arduino.

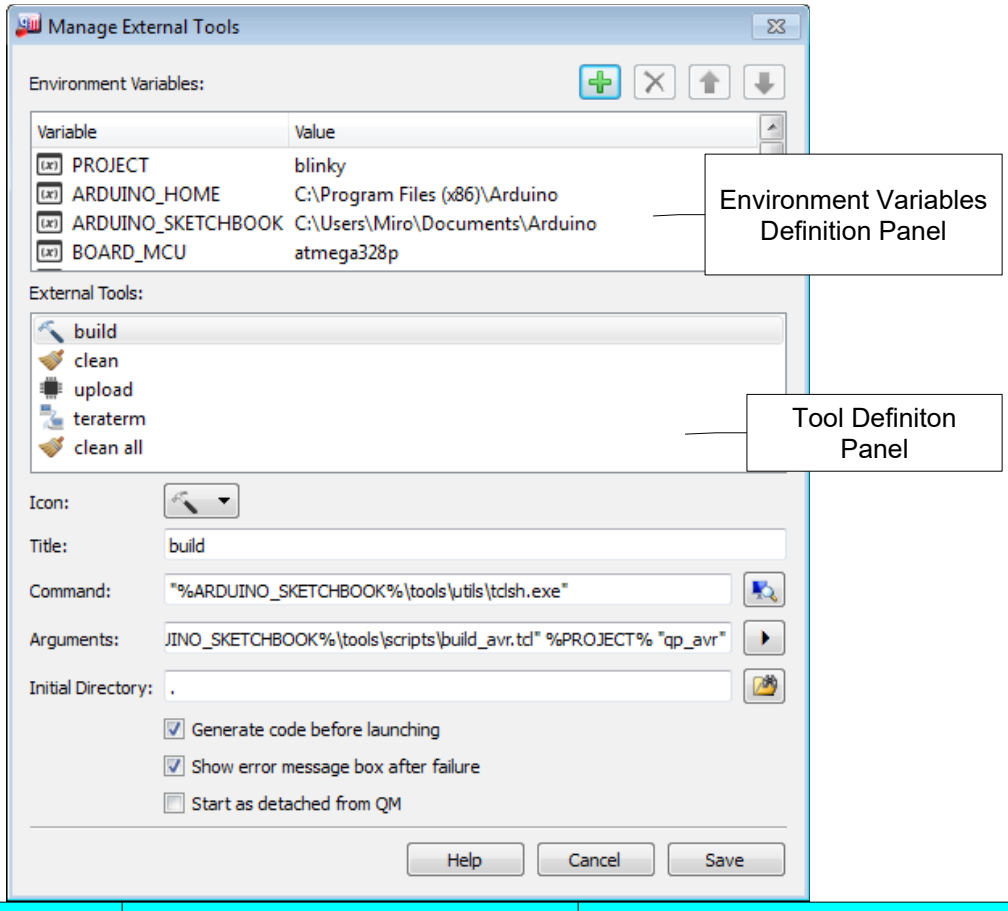
Figure 15: The Tools menu and Tool toolbar in QM



6.1 Configuring The Environment Variables

The the `build_avr.tcl` build script requires defining several environment variables, which you need to adjust to your system through the QM dialog box "**Manage External Tools...**".

Figure 16: The "Manage External Tools" dialog box



Environment Variable	Description	Example (Arduino-UNO board)
`PROJECT`	Project name	`blinky`
`ARDUINO_HOME`	Installation folder of Arduino IDE	`C:\Program Files (x86)\Arduino`
`ARDUINO_SKETCHBOOK`	Your Arduino Sketchbook folder	`C:\Users\Miro\Documents\Arduino`
`BOARD_MCU`	The MCU type used on your board	`atmega328p`
`BOARD_VARIANT`	Variant of the board	`standard`
`BOARD_F_MHZ`	Frequency of the oscillator on the board in MHz	`16`
`BOARD_COM`	COM port (needed only for code upload and external terminal)	`COM5`
`BOARD_BAUD`	Baud rate of the COM Port (needed for code upload & external terminal)	`115200`

NOTE: You can find the correct setting for all the Environment Variables starting with `BOARD_...` by looking inside the **`boards.txt` file** located in `<Arduino>\hardware\arduino\avr\` directory.

6.2 The `build_avr.tcl` Build Script

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

> **NOTE:** The TCL shell interpreter is included in the tools directory inside the QDK-Arduino (`<Arduino_Sketchbook>\tools\utils\tclsh.exe`)

The `build_avr.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 build_avr.tcl <PROJECT> "<LIBS>" ["<DEFINES>"]
```

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

```
"%ARDUINO_SKETCHBOOK%\tools\scripts\build_avr.tcl" %PROJECT% "qp_avr Ethernet"
"BAR FOO=25"
```

> **NOTE:** The `build_avr.tcl` build script requires the Environment Variables are set correctly, as described in the previous section.

The `build_avr.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 or in your Sketchbook `<Arduino_Sketchbook>\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. 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.
4. 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.
5. 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.

6.3 Uploading Code to Arduino

The batch script `upload_avr.bat`, located in `<Arduino_Sketchbook>\tools\scripts\` directory uses the standard `avrdude.exe` utility for uploading the code to the Arduino boards.

NOTE: The `avrdude.exe` utility cannot be called directly, because it requires the CYGWIN dynamic libraries, which won't be found unless the `<Arduino>` installation directory is in the PATH. The `upload_avr.bat` batch script encapsulates this dependency without a need for adding anything to your global PATH.

The `avrdude.exe` utility takes many quite complex parameters, but all of them are controlled by the Environment Variables, so the "upload" tool is already configured and you don't need to edit the `avrdude` arguments directly (in the "Manage External Tools..." dialog box).

The image loaded to the Arduino board is taken from the `bin\%PROJECT%.hex` file, where `%PROJECT%` is the name of the project defined in the environment variable.

NOTE: If you wish to remain compatible with the standard Arduino build process in the Arduino IDE, you should define the `PROJECT` environment variable to be the same as the directory name holding the project.

6.4 The rm Utility for Cleaning Up the Build

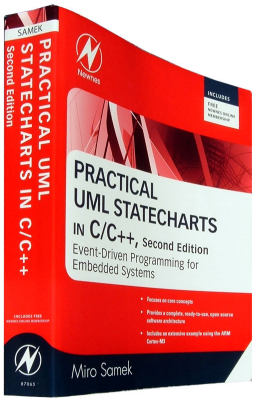
For convenience of cleaning your builds, the `tools\utils\` directory contains the `rm.exe` utility for deleting files. The `rm.exe` utility on Windows is designed to work the same way as the `rm` utility in Unix. The utility takes a list of files to be deleted. The files can be also specified using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

6.5 The serialterm Utility for Displaying the Serial Output from Arduino

The `tools\utility\` folder contains also a simple console-based `serialterm.exe` application, which you can launch directly from QM. This will open the Windows Command Prompt, in which you can view the textual output coming from your Arduino board. You can also send characters to the Arduino board by typing while the serial-terminal has the keyboard focus. You exit `serialterm` utility by pressing the **Esc** key, or **Ctrl-C** key combination.

NOTE: You need to exit the serial terminal to be able to download new code to your Arduino board.

7 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

8 Contact Information

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

WEB : http://www.state-machine.com
Support:sourceforge.net/projects/qpc/forums/forum/668726

Arduino Project

homepage:
http://arduino.cc