



Quantum[®]Leaps
Modern Embedded Software



Application Note: **Event-Driven Arduino Programming with QP[™] and QM[™]**

Document Revision R
February 2020

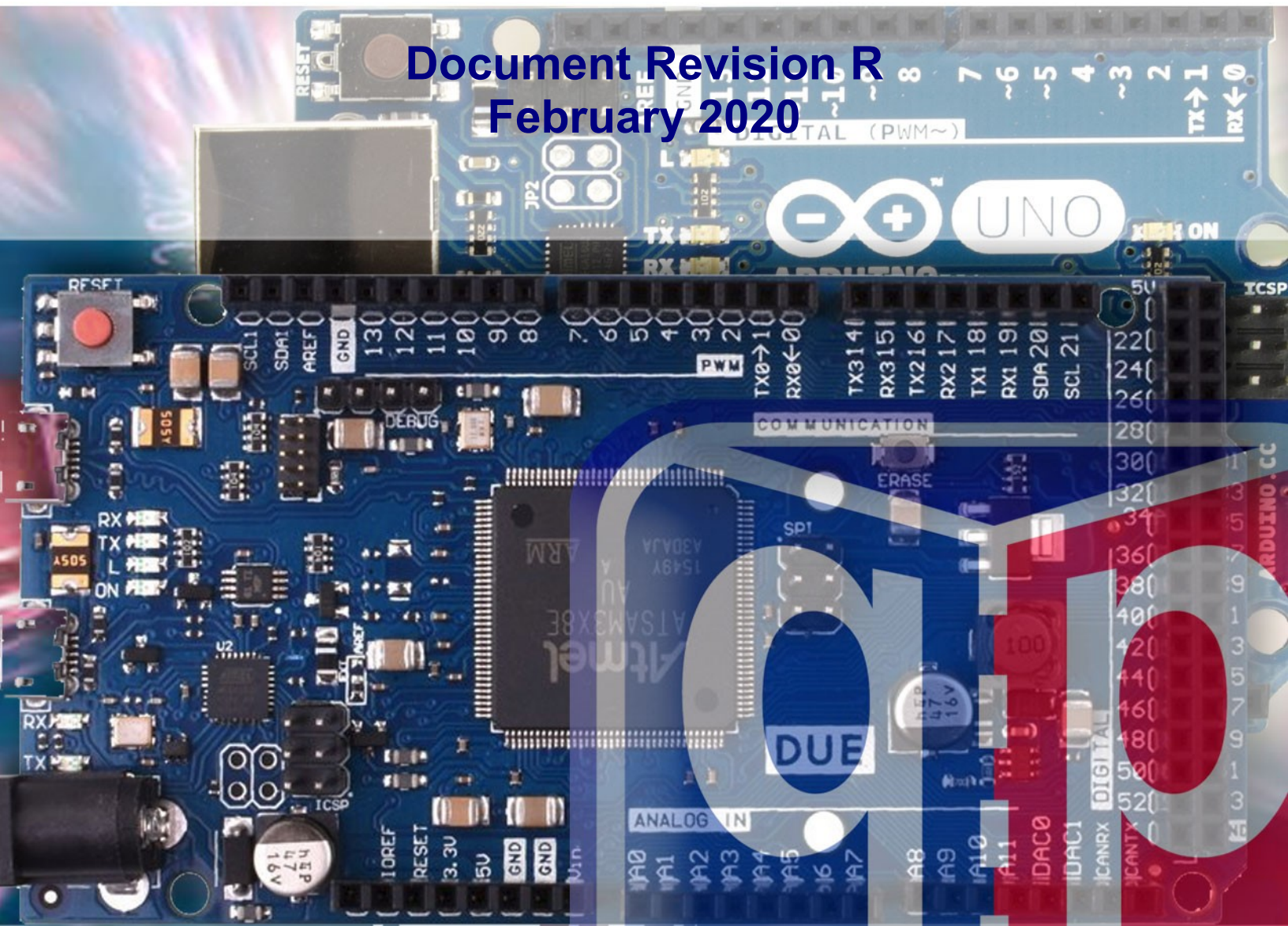


Table of Contents

1 Introduction.....	1
1.1 About Arduino.....	1
1.2 Event-Driven Programming with Arduino.....	2
1.3 QP™ Real-Time Embedded Frameworks.....	3
1.4 QM™ Graphical Modeling Tool.....	5
2 Getting Started.....	6
2.1 Software Installation.....	7
2.2 Modeling and Generating Code.....	10
2.3 Building the Examples.....	10
2.4 The Dining Philosophers Problem Example.....	13
2.5 The PELICAN Crossing Example (qpn_avr Library).....	14
3 The Structure of an Arduino Sketch for QP-nano™.....	15
3.1 Include files.....	17
3.2 Events.....	17
3.3 Active Object declarations.....	17
3.4 Board Support Package.....	18
3.5 Interrupts.....	18
3.6 QP-nano Callback Functions.....	19
3.7 The Assertion Handler.....	19
3.8 Define the Active Objects (Generate the State Machine Code).....	19
4 The Structure of an Arduino Sketch for QP/C++™ (qpcpp_sam Library).....	20
4.1 Include files.....	21
4.2 Miscellaneous Declarations.....	21
4.3 Initialization.....	21
4.4 Starting Active Objects.....	21
4.5 Transferring Control to the QP/C++ Framework.....	21
5 Working with State Machines.....	22
6 Related Documents and References.....	24
7 Contact Information.....	25

Legal Disclaimers

Information in this document is believed to be accurate and reliable. However, Quantum Leaps does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information.

Quantum Leaps reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

All designated trademarks are the property of their respective owners.



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™ real-time embedded frameworks](#), which are like modern **real-time operating systems** (RTOSes) 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.

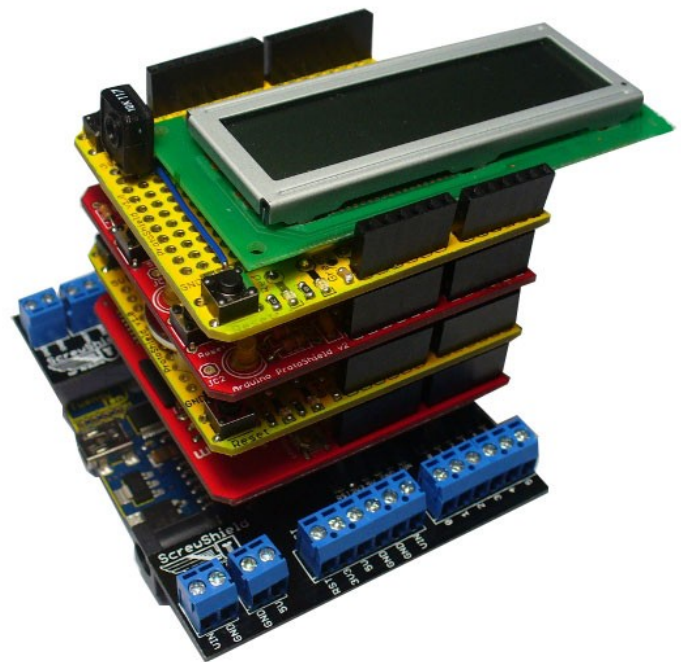
1.1 About Arduino

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 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 Arduino IDE (integrated development environment), called Processing, that runs on a desktop computer (Windows, Linux, or MacOS).

Arduino boards can be purchased preassembled 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-known 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*. The QP[™] frameworks described in the next section allow you to combine the event-driven programming paradigm with modern state machines.

1.3 QP[™] Real-Time Embedded Frameworks

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*.

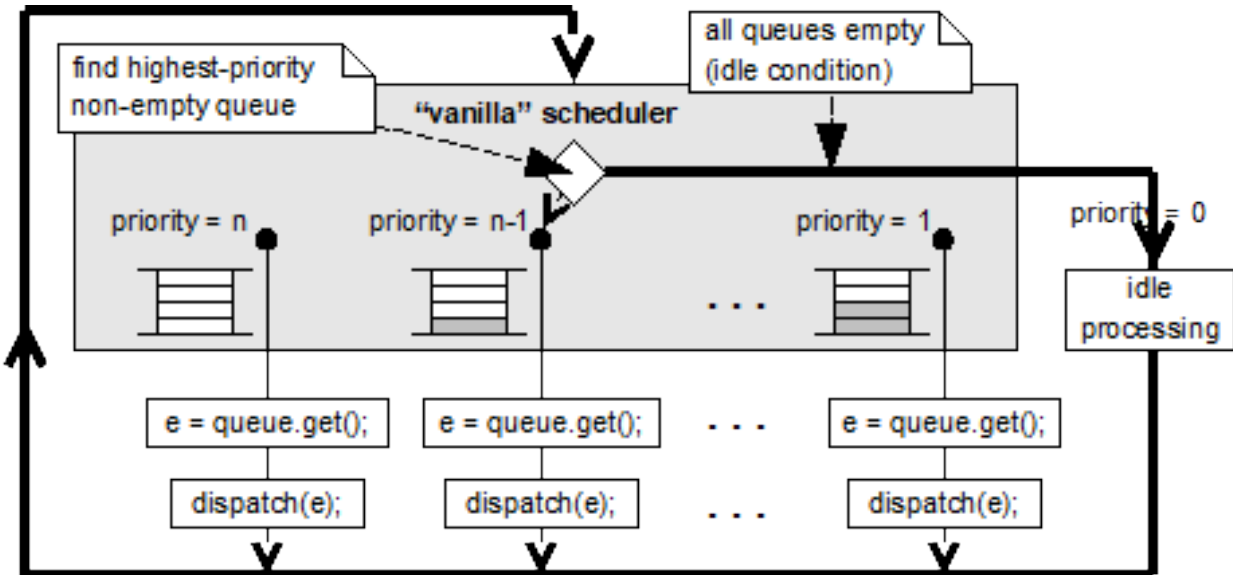
QP[™] real-time embedded frameworks provide a more complete event-driven software infrastructure for programming Arduino. Currently, the QP-nano framework supports AVR-based Arduino boards (such as Arduino UNO) and QP/C++ supports SAM-based Arduino boards (such as Arduino DUE).

Figure 2 shows the structure of the cooperative scheduler inside the QP[™] frameworks, 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 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), 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 framework contains also a more advanced, fully **preemptive** real-time kernel called QK. However, for simplicity, the QP development kit for Arduino currently does not provide the QV 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

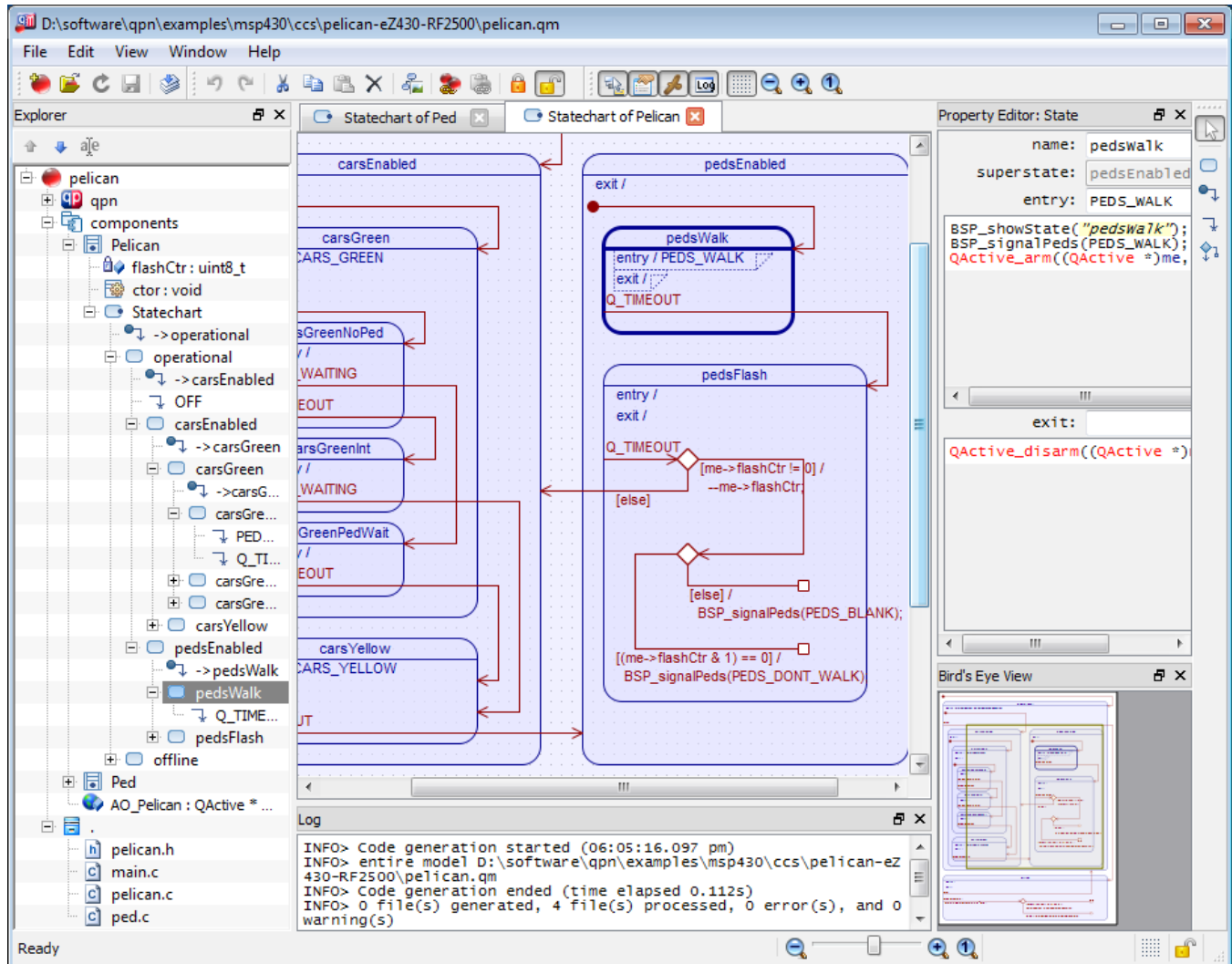
QM™ (Quantum Modeler) is a free, cross-platform, graphical modeling tool for designing and implementing real-time embedded applications based on the QP™ state machine frameworks. QM™ is available for Windows 32/64-bit, Linux 64-bit, and MacOSX.

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™.



NOTE: QM for Windows is included in the QP-Arduino download, as described in Section 2.1.

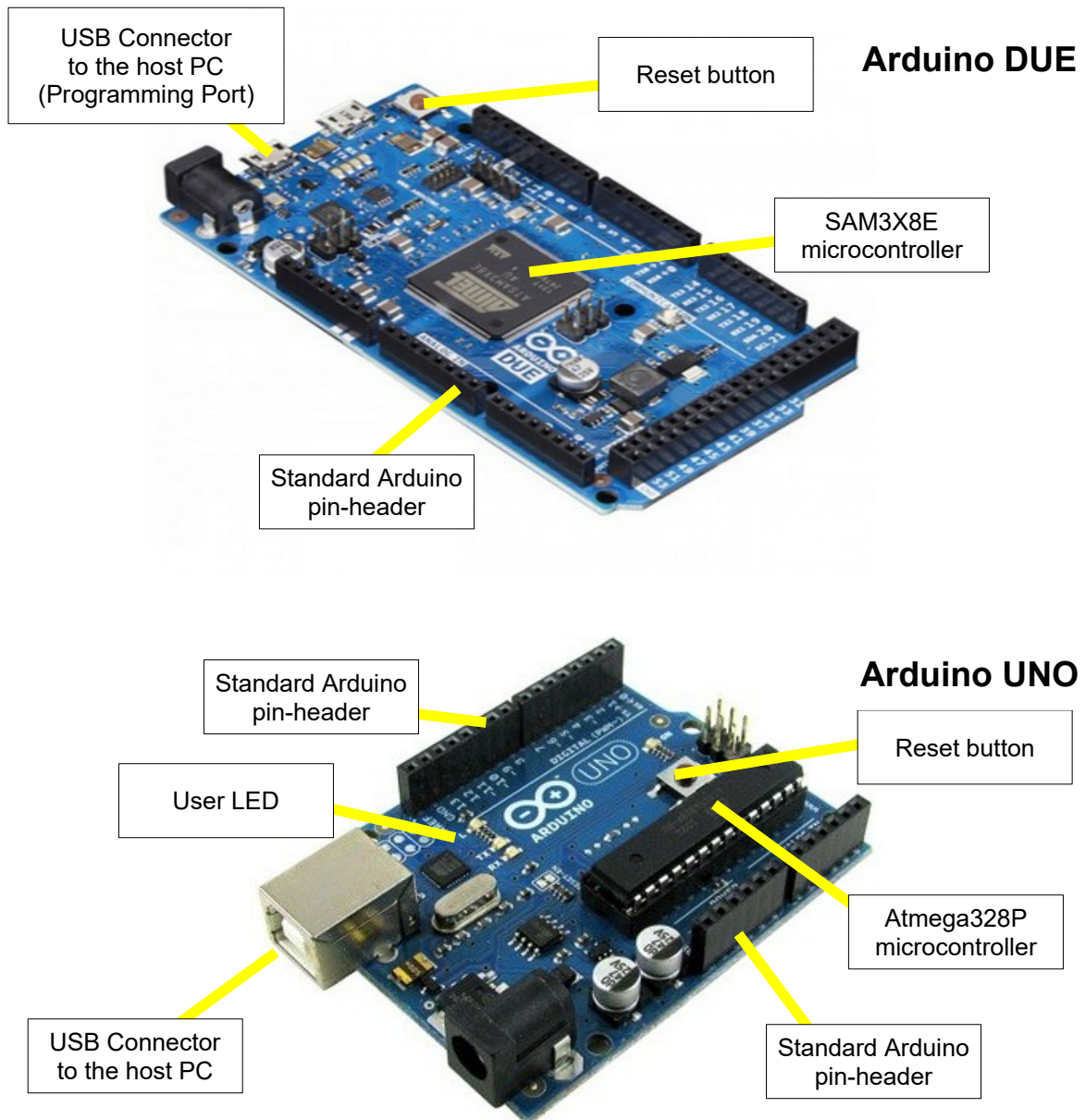
Figure 3: The PELICAN example model opened in the QM™ modeling tool



2 Getting Started

QP for Arduino is available for free download from [GitHub](#). QP for Arduino is distributed in a ZIP archive `qp-<ver1>_arduino-<ver2>.zip`, where `<ver1>` stands for the QP version and `<ver2>` for the version of the Arduino software (e.g., 1.8.x). Additionally to this QP-Arduino ZIP file, you need to download and install the standard **Arduino 1.8.x** (the latest as of this writing), which is available from the [Arduino website](#). To focus the discussion, this Application Note uses the **Arduino DUE** board based on the ARM Cortex-M SAM microcontroller and the **Arduino UNO** board based on the Atmega328p microcontroller (see [Figure 4](#)).

Figure 4: Arduino DUE and Arduino UNO boards

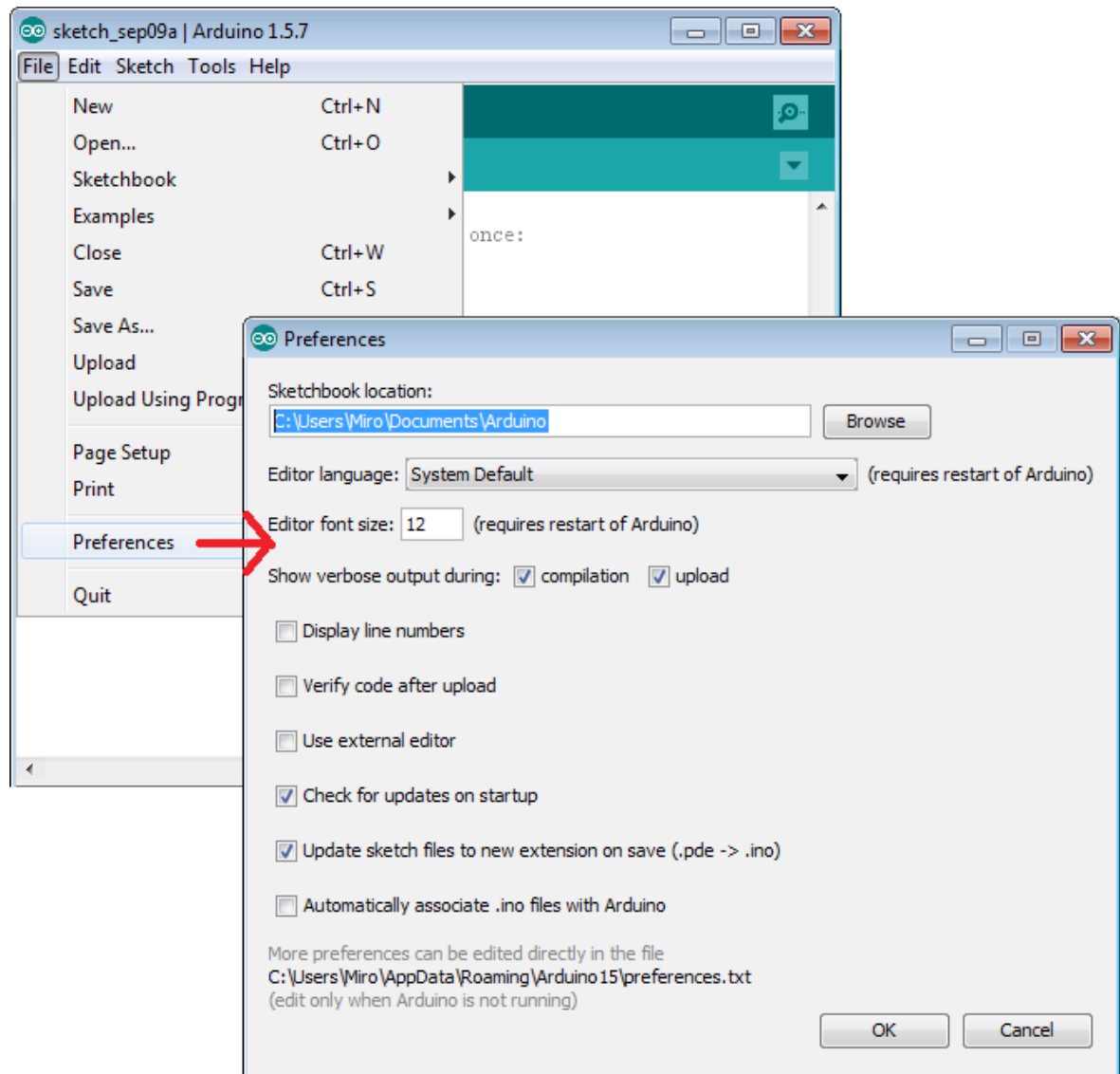


2.1 Software Installation

NOTE: The following discussion assumes that you have downloaded and installed both the standard Arduino software for **Windows** and the **QP-Arduino ZIP file**, as described above.

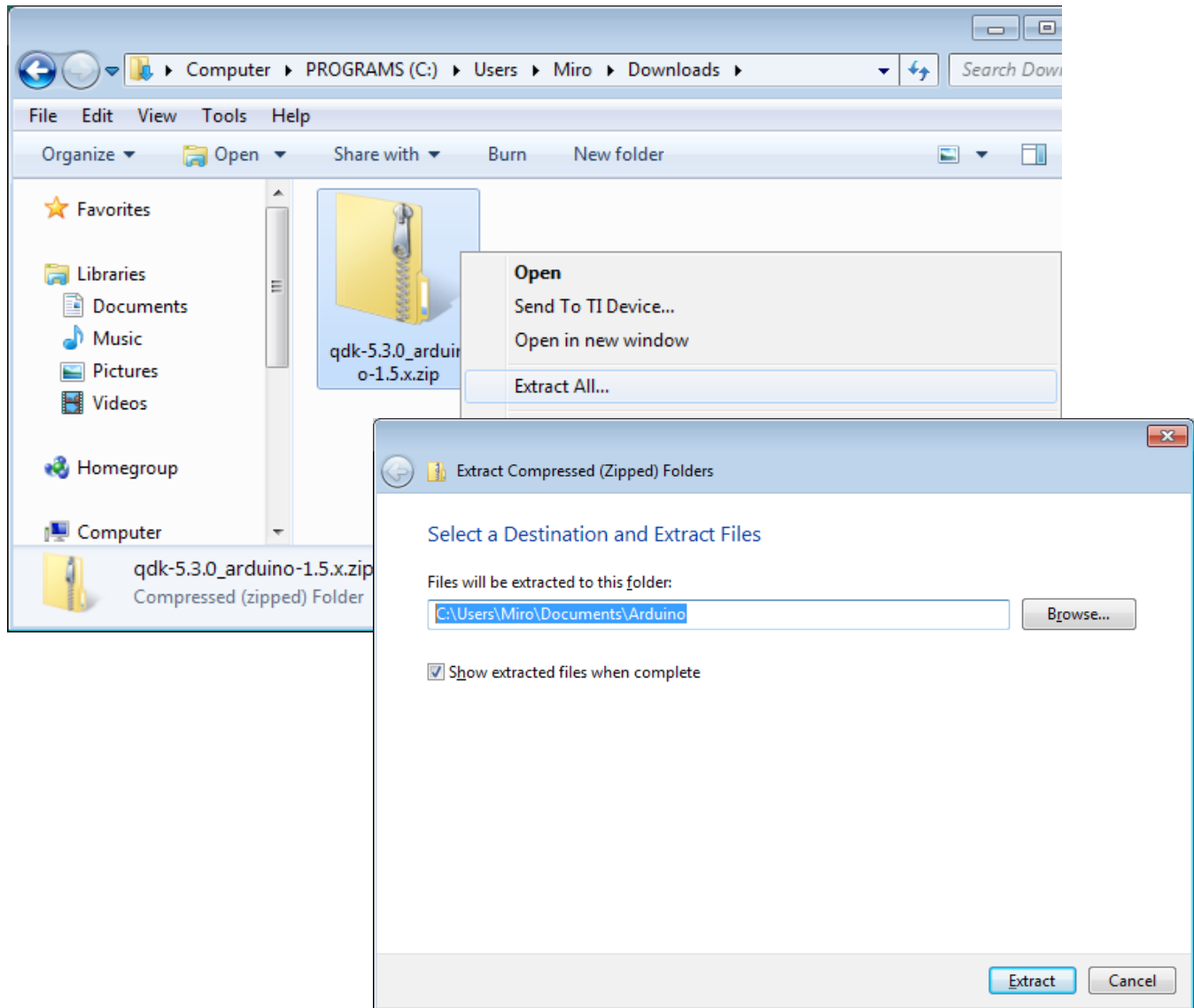
You need to unzip the `qp-<ver1>_arduino-<ver2>.zip` archive into your **Arduino-Sketchbook folder**. To find out where your Sketchbook folder is, or to configure a different location for your Sketchbook folder, you need to open the Arduino IDE and select **File | Preferences** menu (see [Figure 6](#)). 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-QM.pdf - this document
| +-AN_DPP.pdf         - Dining Philosopher Problem example application
| +-AN_PELICAN.pdf     - PEdestrian LIght CONTROLled (PELICAN) crossing example
|
+-libraries/
| +-qpcpp_sam/         - QP/C++ library for SAM-based Arduinos
| | +-examples/
| | | +-blinky/        - Very simple "blinky" example
```

```

| | | | +-blinky.ino    - The Blinky sketch generated by QM
| | | | +-blinky.qm    - The QM model of the Blinky application
| | | +-blinky_bsp/    - "blinky" example with board-support package (BSP)
| | | | +-blinky_bsp.ino - The Blinky-BSP sketch generated by QM
| | | | +-blinky.qm    - The QM model of the Blinky application
| | | +-dpp_bsp/       - Dining Philosophers Problem (DPP) example with BSP
| | | | +-dpp_bsp.ino  - The DPP-BSP sketch generated by QM
| | | | +-dpp.qm       - The QM model of the DPP application
| | |
| | +-src/             - Source code for the QP/C++ library for SAM-based Arduinos
| | | +-...
| | +-library.properties - Properties of the QP/C++ library for the Arduino IDE
| |
| +-qpn_avr/           - QP-nano library for AVR-based Arduinos
| | +-examples/
| | | +-blinky/        - Very simple "blinky" example
| | | | +-blinky.ino   - The Blinky sketch generated by QM
| | | | +-blinky.qm    - The QM model of the Blinky application
| | | | +-dpp/         - Dining Philosophers Problem (DPP) example
| | | | +-dpp.ino      - The DPP sketch generated by QM
| | | | +-dpp.qm       - The QM model of the DPP application
| | | | +-pelican/     - PEdestrian LIght CONTROLled (PELICAN) crossing example
| | | | +-pelican.ino  - The PELICAN sketch generated by QM
| | | | +-pelican.qm   - The QM model of the PELICAN crossing application
| | |
| | +-src/             - Source code for the QP-nano library for AVR-based Arduinos
| | | +-...
| | +-library.properties - Properties of the QP-nano library for the Arduino IDE
| |
| +-qm/               - QM modeling tool for Windows
| | +-bin/            - QM binaries (executable and DLLs)
| | | +-qm.exe        - QM executable for Windows
| | +-Resources/      - QM resources
| | | +-...
| |
+-GPLv3.txt           - GNU General Public License version 3
+-QP-Arduino_GPL_Exception.txt - QP GPLv3 exception for Arduino
+-README.md           - README file with basic instructions

```

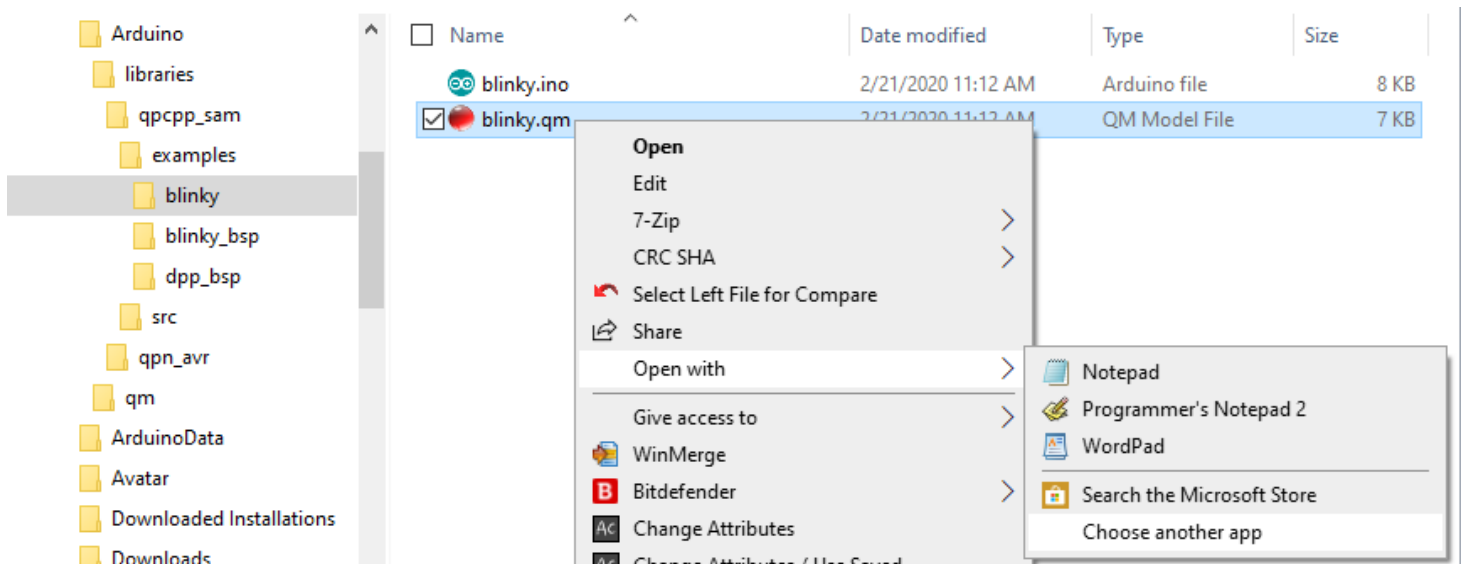
NOTE: The QP-Arduino archive contains **QM for Windows** only. But QM is also available for Linux and MacOS hosts. If you wish to work on those operating systems, you will need to install QM separately, as described at: <https://www.state-machine.com/qm/gs.html>. Please also see [Related Documents and References](#).

2.2 Modeling and Generating Code

Each QP example for Arduino (in the `library\qpcpp_sam\examples` and `library\qpn_avr\examples` folders) 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.

NOTE: You can configure the Windows File Explorer to open the `.qm` model file automatically with QM by right-clicking on a `.qm` file and choosing "Open with"/"Choose another app" option, after which you select the `qm.exe` executable in the `<Sketchbook>\qm\bin\` directory.

Figure 7: Setting the "Open with" property for the *.qm model files in Windows Explorer



2.3 Building the Examples

After you generate the code with QM, you **build and upload** it to the Arduino boards using the **standard Arduino IDE**. Please note that you typically do NOT use the Arduino IDE to actually edit the code, there

NOTE: You typically do **not** use the Arduino IDE to actually edit the code manually. Therefore you should set the Preferences (File | Preferences menu) to **"Use external editor"**. This will cause the Arduino IDE to automatically **refresh** the project files every time the code is re-generated from QM.

Figure 8: Setting the "Use external editor" option in the Standard Arduino IDE

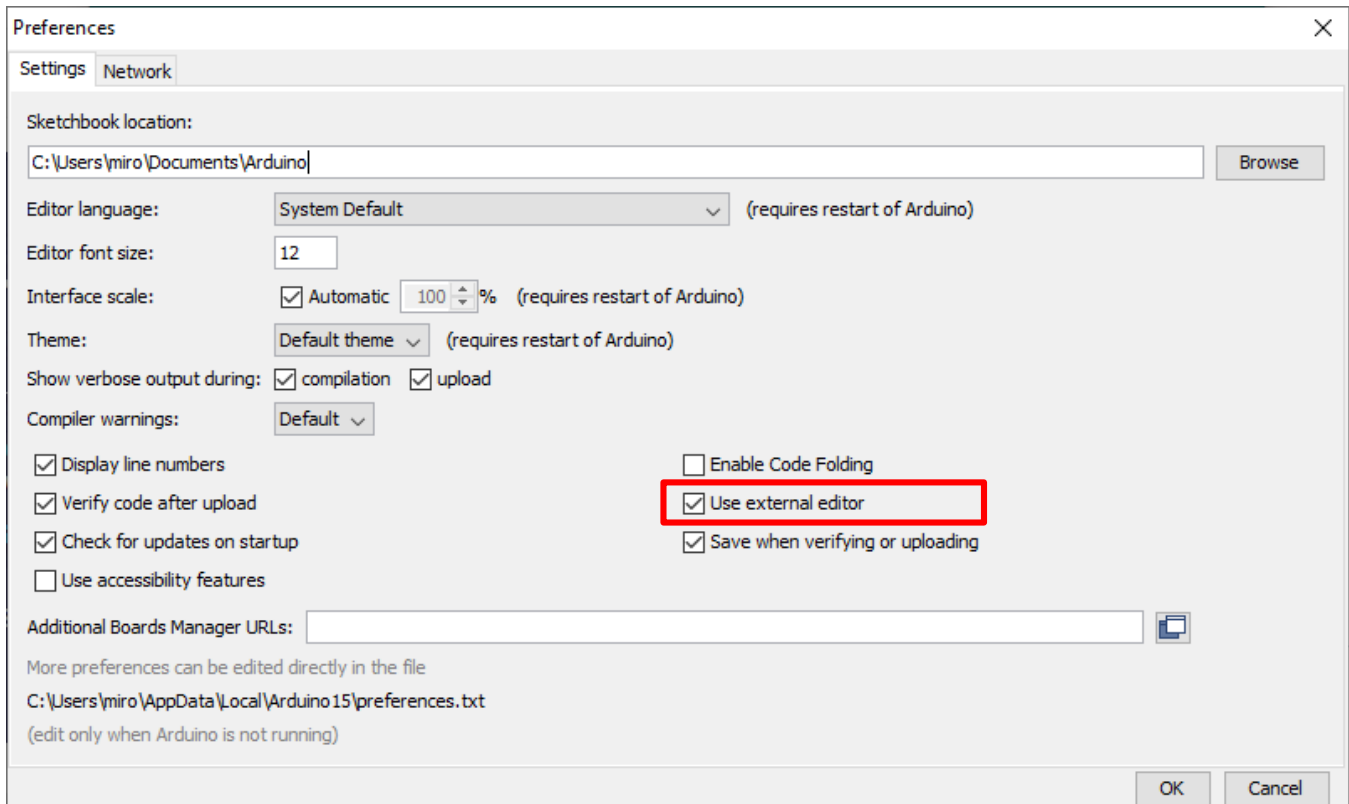
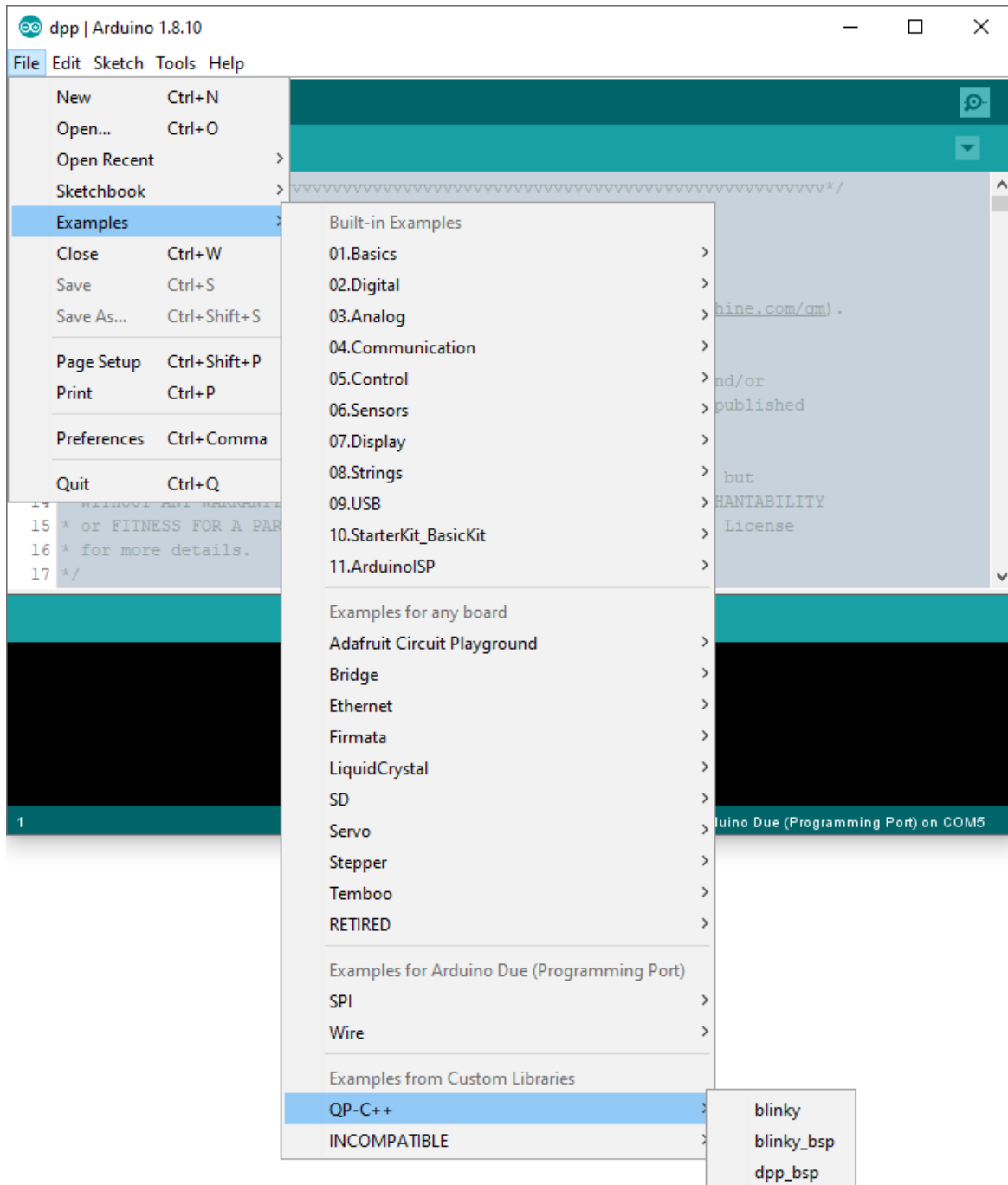


Figure 9: Opening the QP/C++ Examples from the Standard Arduino IDE



NOTE: The available "Examples from Custom Libraries" displayed by the Arduino IDE depend on the currently chosen Arduino board. For the AVR-based Arduino boards, the Arduino IDE displays examples from the "QP-nano" library. For the SAM-based Arduino boards, the IDE displays examples from the "QP-C++" library (see Figure above).

2.4 The Dining Philosophers Problem Example

The Dining Philosophers Problem (DPP) example demonstrates multiple active objects. This example is located in the directory `<Arduino_Sketchbook>\library\ qcppp_sam\examples\dpp` for SAM-based Arduinos and `<Arduino_Sketchbook>\library\ qpn_avr\examples\dpp` for AVR-based Arduinos.

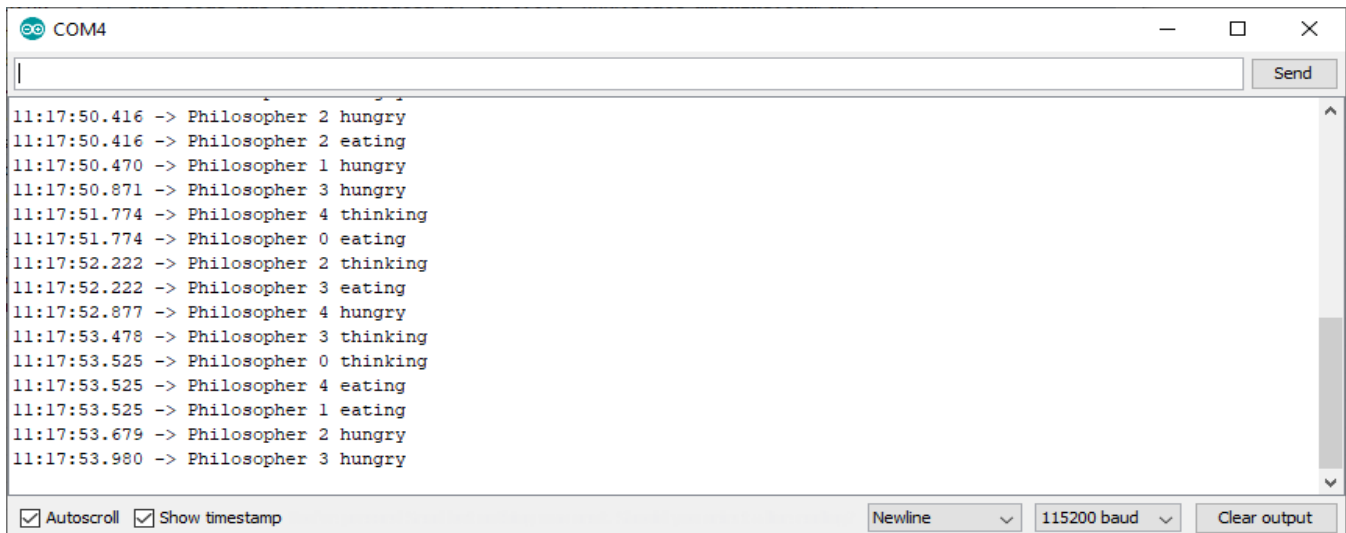
The classical Dining Philosopher Problem (DPP) was posed and solved originally by Edsger Dijkstra in the 1970s and is specified as follows: Five philosophers are gathered around a table with a big plate of spaghetti in the middle (see [Figure 10](#)). Between each two philosophers is a fork. The spaghetti 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 10: 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 Monitor** by pressing the Serial Terminal button on the Arduino IDE toolbar.



```
11:17:50.416 -> Philosopher 2 hungry
11:17:50.416 -> Philosopher 2 eating
11:17:50.470 -> Philosopher 1 hungry
11:17:50.871 -> Philosopher 3 hungry
11:17:51.774 -> Philosopher 4 thinking
11:17:51.774 -> Philosopher 0 eating
11:17:52.222 -> Philosopher 2 thinking
11:17:52.222 -> Philosopher 3 eating
11:17:52.877 -> Philosopher 4 hungry
11:17:53.478 -> Philosopher 3 thinking
11:17:53.525 -> Philosopher 0 thinking
11:17:53.525 -> Philosopher 4 eating
11:17:53.525 -> Philosopher 1 eating
11:17:53.679 -> Philosopher 2 hungry
11:17:53.980 -> Philosopher 3 hungry
```

2.5 The PELICAN Crossing Example (qpn_avr Library)

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 “Don’t 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 Monitor** by pressing the Serial Terminal button on the Arduino IDE. Once the Serial Monitor opens, type 'p' to simulate pressing of the PEDESTRIAN WALK button.

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

This section describes the structure of the Arduino "sketch" file **inside the QM modeling tool** and before the code of the sketch is generated into a file on disk. Such a 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 Error: Reference source not found](#)), 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 "qp.h"          // QP-nano framework

    //=====
[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] QMAActiveCB const Q_ROM QF_active[] = {
        { (QMAActive *)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...

```

```

    . . .

    // 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 "qp_n.h" header file). (For the QP/C++ library for the SAM-based Arduino, you need to use the "qpcpp.h" header file)

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_Philos 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.

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).

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 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 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 The Structure of an Arduino Sketch for QP/C++™ (qpcpp_sam Library)

This section describes the structure of the Arduino "sketch" file **inside the QM modeling tool** and before the code of the sketch is generated into a file on disk. The example sketch (`dpp_bsp.ino`) is a bit more advanced than the simplified `blinky.ino` sketch explained for QP-nano. This is because the `dpp_bsp` project consists of multiple files, including the BSP (Board Support Package). This section discusses only the main `dpp_bsp.ino` sketch, which provides the `setup()` and `loop()` Arduino functions.

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 5: Typical Arduino sketch for QP™/C++ (file `dpp_bsp.ino`)

```
[1] #include "qpcpp.hpp" // QP-C++ framework
    #include "dpp.hpp"   // DPP application
    #include "bsp.hpp"   // Board Support Package

[2] using namespace QP;

[3] Q_DEFINE_THIS_FILE

    //.....
[4] void setup() {
[5]     QF::init(); // initialize the framework
[6]     BSP::init(); // initialize the Board Support Package

    // init publish-subscribe
[7]     static QSubscrList subscrSto[MAX_PUB_SIG];
[8]     QF::psInit(subscrSto, Q_DIM(subscrSto));

    // initialize event pools...
[9]     static QF_MPOOL_EL(TableEvt) smlPoolSto[2*N_PHILO];
[10]    QF::poolInit(smlPoolSto,
                 sizeof(smlPoolSto), sizeof(smlPoolSto[0]));

    // start all active objects...

    // start Philos
[11]    static QP::QEvt const *philoQueueSto[10][N_PHILO];
    for (uint8_t n = 0U; n < N_PHILO; ++n) {
[12]        AO_Philos[n]->start((uint_fast8_t)(n + 1U), // priority
                             philoQueueSto[n], Q_DIM(philoQueueSto[n]),
                             (void *)0, 0U);
    }
    // start Table
[13]    static QP::QEvt const *tableQueueSto[N_PHILO];
[14]    AO_Table->start((uint_fast8_t)(N_PHILO + 1U), // priority
                   tableQueueSto, Q_DIM(tableQueueSto),
                   (void *)0, 0U);
}

    //.....
```



```
[15] void loop() {  
[16]     QF::run(); // run the QF/C++ framework  
    }
```

4.1 Include files

- [1] Each sketch must include the QP/C++ library (the "qpcpp.hpp" header file), as well as other header files from the project.

4.2 Miscellaneous Declarations

- [2] The declaration `using namespace QP;` allows you to reference QP/C++ facilities without the `QP::` prefix, which simplifies the code. (NOTE: the `QP::` namespace can be useful to allow name conflicts in larger projects).
- [3] The macro `using Q_DEFINE_THIS_FILE` defines the file name for this module, which can be subsequently used in QP assertions. It is typically a good idea to provide such a module name for each implementation file.

4.3 Initialization

- [4] The Arduino `setup()` function is used to initialize the QP/C++ framework and the services used by this application.
- [5] The QP/C++ `QF::init()` function must be called before any other QP/C++ service is used.
- [6] The `BSP::init()` function initializes the board (Board Support Package)
- [7] The DPP example uses the QP/C++ publish-subscribe service, so it needs to first allocate memory for the "subscriber lists"
- [8] And next, the call `QF::psInit()` initializes the publish-subscribe service.
- [9] Also, the DPP example uses the QP/C++ dynamic events, so it needs to first allocate memory for the "event pools"
- [10] The QP/C++ `QF::poolInit()` initializes the "event pool" service.

4.4 Starting Active Objects

- [11] Before starting any active object, the memory for the event-queue buffer must be allocated. Here event-queue buffers are allocated for the `Philo` active objects.
- [12] The active object `start()` function starts a given active object with the provided priority, event-queue buffer, and other parameters.
- [13] Event-queue buffer for the `Table` active object is allocated.
- [14] The active object `start()` function starts `Table` active object with the provided priority, event-queue buffer, and other parameters.

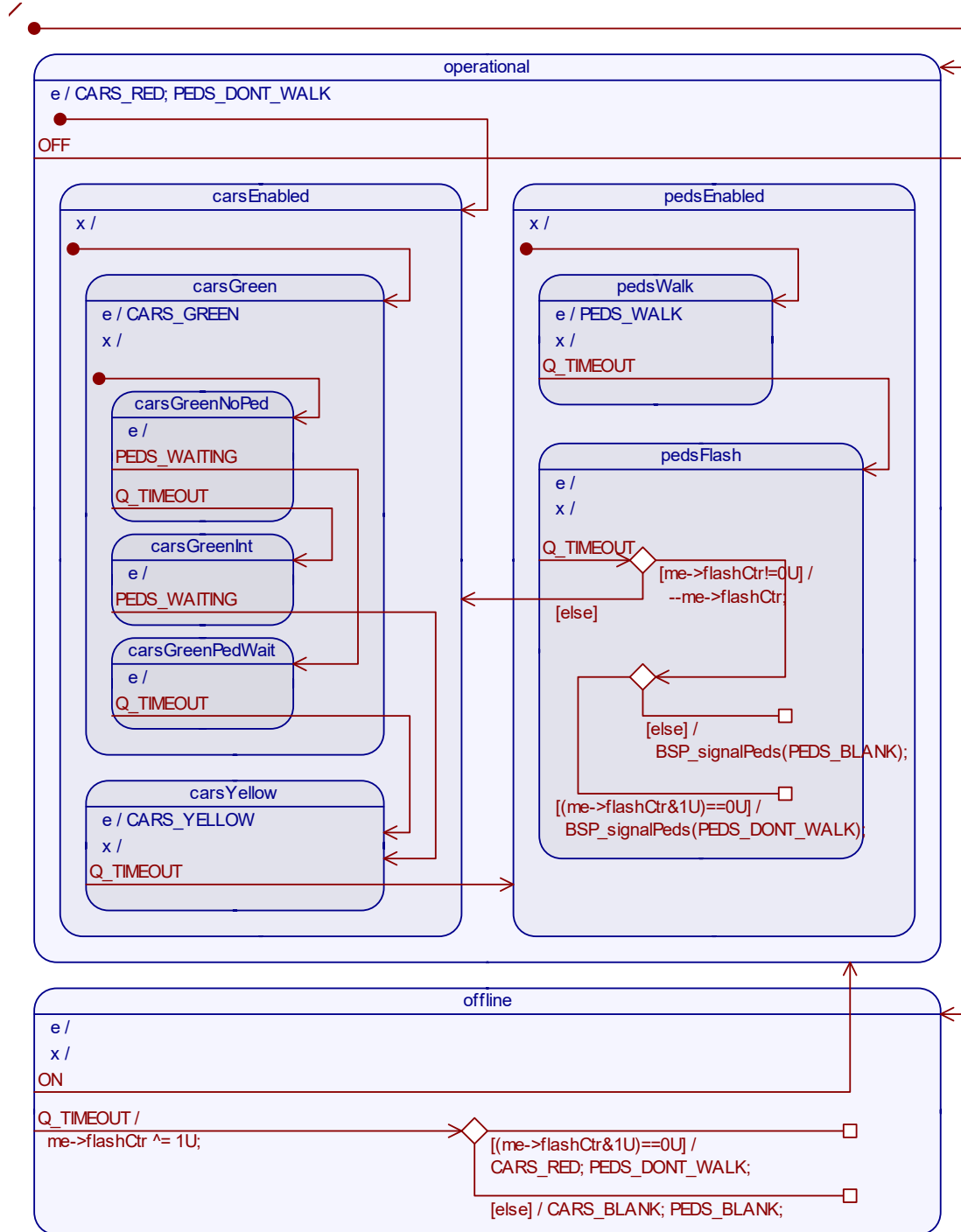
4.5 Transferring Control to the QP/C++ Framework

- [15] The standard Arduino function `loop()` simply passes control the QP/C++ framework.
- [16] The QP/C++ `QF::run()` function runs you application and does **not** return.

5 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 12](#) shows the HSM diagram 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-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.

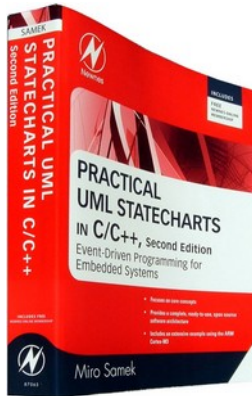
NOTE: The state machine concepts, state machine design, and hierarchical state machine implementation in C are not specific to Arduino and are out of scope of this document. The design and implementation of the DPP example is described in the separate Application Note “Dining Philosophers Problem” (see [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 framework makes particularly straightforward, or you can use the QM tool to generate code automatically.

6 Related Documents and References

Document

Location



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

<https://www.state-machine.com/psicc2>

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

<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>

7 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>

