Lab 3: Quadrature Encoder

Taylor Okel

9/25/2018

Table of Contents

Table of Figures ii
Problem Statement 1
  Background 1
  Purpose and Procedure 1
  Requirements 1
Assumptions 1
Equipment 1
Architecture 2
Design 2
  Decoding the Quadrature Encoder 2
  I/O Handling – Interrupt or Poll? 4
  Program Design 4
    Main Loop 4
    ISR 6
Implementation 8
Test Plan 9
Test Results 10
  Demonstration 10
Suggestions 10
  Future Improvements to this Implementation 10
  Future Improvements to Lab Description 10
Code ..... A
Discussion ..... B
References ..... D

Table of Figures

Figure 1: Wiring Diagram 2
Figure 2: Quadrature Encoder P/N Breakdown 3
Figure 3: Signal Graphs 3
Figure 4: Main Program Flow 5
Figure 5: Common Interrupt Service Routine (ISR) 7
Figure 6: Schematic On-board 8

Problem Statement

Background

We have been asked by a major petroleum producer to design a better flow monitor for gas pumps. Our flow rate sensor will be a small in-line turbine, connected to a quadrature encoder. Each revolution of the encoder represents 1 ounce of product. (Stakem and Crum n.d.)

Purpose and Procedure

This lab focuses on implementing a quadrature encoder peripheral as if it were measuring the flow rate of gasoline. As the encoder dial is turned, the board should convert these revolutions to flow rate; 1 revolution equals 1 ounce of product (Stakem and Crum n.d.). These values shall be sent to the host PC via serial connection for display and/or computations. In addition, the board shall control two LEDs that light up when the flow rate exceeds a maximum ("overflow"), or the encoder is turned in the wrong direction ("backflow").

Requirements

The following requirements were derived from the Lab 3 description (Stakem and Crum n.d.)

- The board shall interface with the provided encoder to read the states off the A and B pins
- The board should communicate this data to the host PC.
- The host PC and/or board shall use this data to determine rotation rate
- The host PC and/or board shall use this data to determine the total amount dispersed, in gallons
- The board shall control two LEDs to signal cases of overflow or backflow

Assumptions

- All equipment functions as advertised on their respective datasheets
- The board has a high enough frequency to reasonably complete all tasks assigned to it
- The host PC is not require to perform any tasks *besides* display
    - "This **can** be done on the host PC"

Equipment

- Olimexino-STM32 Development Board (henceforth "board")
    - STM32F103RBT6 Arm Processor (Olimexino-STM32 development board User's manual 2016)
- USB programming cable
- Programming Workstation (henceforth "computer")
    - Windows 10 operating system
    - Arduino IDE

- 858-EN12-HN22AF25 Quadrature Encoder

Architecture

The center pin (“common” or “C” pin) of the quadrature encoder is wired to ground, while the left pin (“A”) and right pin (“B”) are wired to inputs D2 and D4, respectively, of the board with 10KΩ pullup resistors. Additionally, pins D0 and D14 of the board are wired to red and yellow LEDs, respectively, with the negative end wired to ground. Meanwhile, the 3.3V and GND pins are connected to the vertical strips on the breadboard for common use. Figure 1 is a graphical depiction of the schematic.

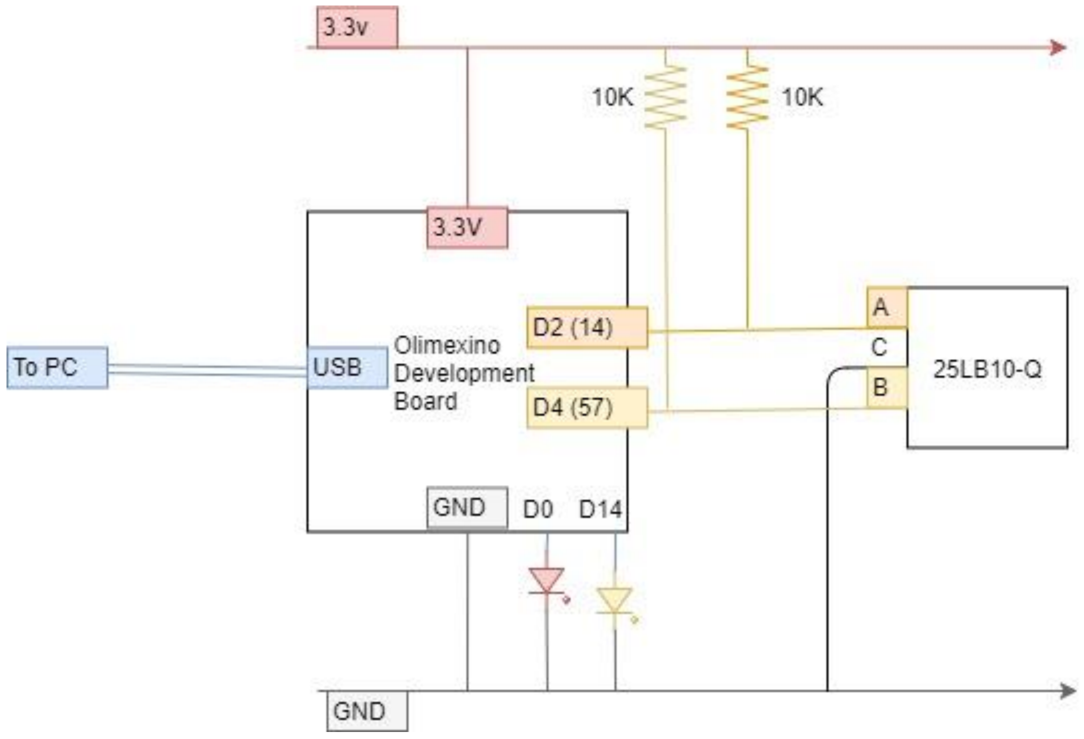

Figure 1: Wiring Diagram

Design

Decoding the Quadrature Encoder

Using the part number of the given encoder, the datasheet for the device was obtained. Using this datasheet, two conclusions can be made: the number of pulses per revolution, and the bit change pattern.

The following snips of the datasheet (TT Electronics EN12-HN22AF25 2018) provide information on the Part Number breakdown and signal graphs for the data pins.

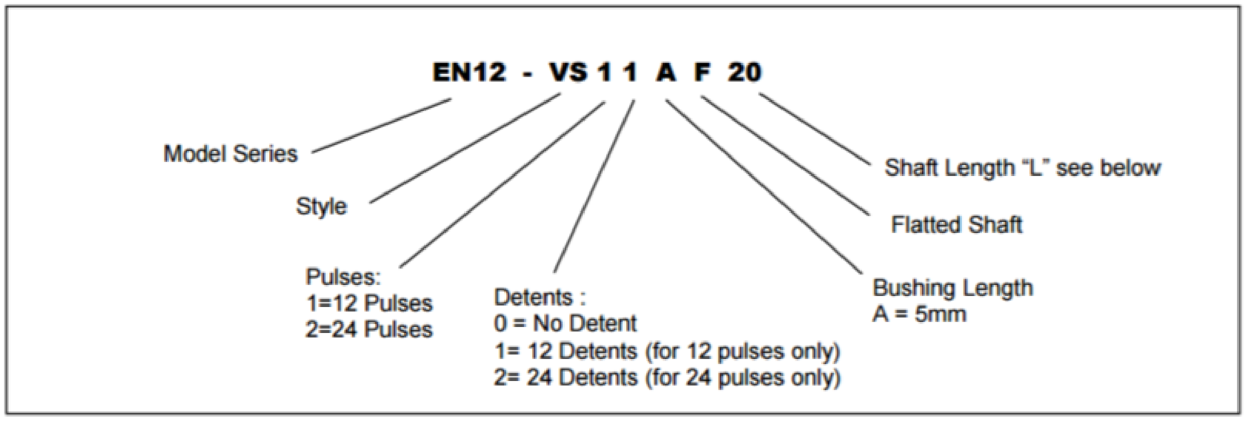

Figure 2: Quadrature Encoder P/N Breakdown

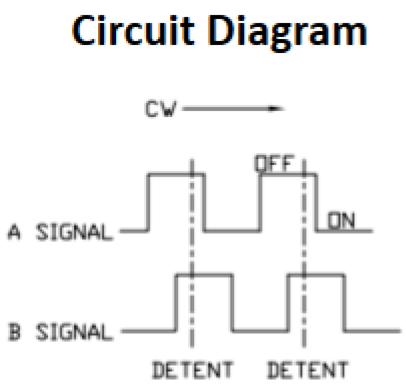

Figure 3: Signal Graphs

Using Figure 2, with our part number of 858-EN12-HN22AF25, concludes that there are 24 pulses per revolution. Figure 3 reveals the pattern that the pins emit. This pattern is digested into the sequence demonstrated in Table 1. This pattern plays a big role in determining the flow direction.

Table 1: Quadrature Encoder Pattern for Clockwise Rotation

A	B
1	1
0	1
0	0
1	0
1	1

Using Table 1, look individually at the changes of A (0→1 or 1→0). Observe that the B values at these changes follow a pattern; when A goes low (1→0), B is high. When A goes high (0→1), B is low. If the opposite is true (for instance, A goes low while B is low), then this pattern is reversed. In other words, this pattern holds true when the device is rotated clockwise but is not when rotated counterclockwise. B holds a similar pattern; B(0→1) : A(1), and B(1→0) : A(0). If this pattern is violated, counterclockwise rotation is occurring.

For completeness, note that no rotation maintains the same state.

I/O Handling – Interrupt or Poll?

When dealing with IO, there are two methods of responding to changes; polling, where the pin is constantly checked for changes, and interrupts, where a signal pauses the running program while the input is handled.

Both methods have pros and cons; interrupts need to be quick, as they can starve the rest of the program, whereas polling can waste resources or miss changes if the update rate is not quick enough.

In this implementation, a lot of value is placed in catching every update from the quadrature encoder. Missing a state means missing rotation direction information. Further, there are easy implementations that efficiently consume the input on interrupt. Therefore, an interrupt model was chosen.

Program Design

The design can be broken into two sub-topics: the main loop, and a common ISR.

Main Loop

The main flow of the control program is depicted in Figure 4. Two interrupt service routines (ISRs) are attached to the D2 and D4, or the A and B pin of the quadrature encoder. The main program then launches into an infinite loop that controls the flow calculations and communicates. This loop repeats every second.

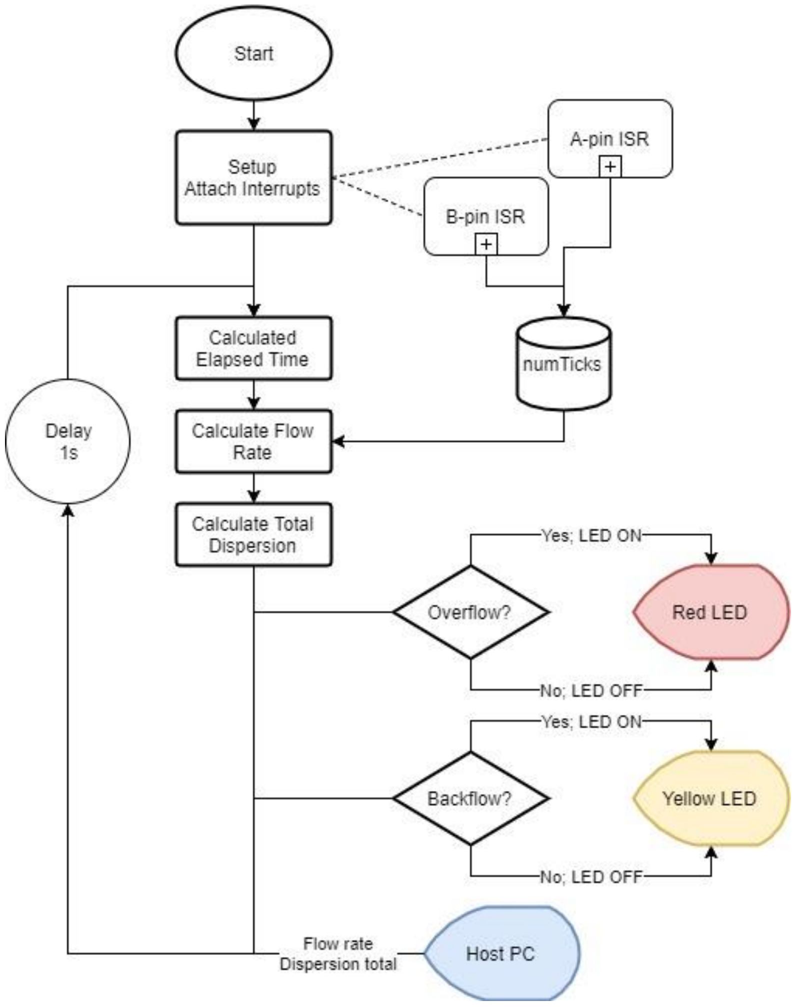

Figure 4: Main Program Flow

To measure elapsed time, the micros API call from Arduino was used. This call returns the number of microseconds since the program launched. By keeping track of the timestamp of the last update, the total elapsed time can be easily calculated.

Equation 1: Elapsed Time Calculation

$$elapsedTime = micros() - lastUpdate; lastUpdate = micros();$$

While it is always about 1 second, due to the delay, this provides a more accurate readout.

To calculate the flow rate, the following equation is used:

Equation 2: Flow Rate Calculation

$$flowRate = \left(\frac{numTicks * ouncesPerTick}{elapsedTime}\right) * 1^{6} \quad \frac{ounces}{second}$$

Where *numTicks* is the number of clockwise ticks, as incremented by the ISR shown in Figure 5: Common Interrupt Service Routine (ISR)Figure 5 (a detailed explanation is given later), *ouncesPerTick* is a conversion ratio calculated using Figure 2 and the lab description (Stakem and Crum n.d.) of 1 oz per revolution,

Equation 3: Ounces Per Tick

$$ouncesPerTick = \frac{1oz}{24ticks}$$

And *elapsedTime* is calculated in Equation 1. The entire result is then multiplied by 1^6 to account for the microsecond unit of *elapsedTime*. Finally, this result can be converted into gallons and accumulated into a *totalDispersed* variable:

Equation 4: Total Dispersed Calculation

$$totalDispersed \mathrel{+}= \left(\frac{flowRate}{128} * elapsedTime\right)\ Gallons$$

With the data in hand, compare the rate to the maximum flow rate; if it exceeds the max flow rate (overflow), light the red LED; otherwise, make sure the LED is off. Then determine if rotation is clockwise (*numTicks,* or *flowRate* by extension, positive) or counterclockwise (negative). If counterclockwise, then light the yellow LED; otherwise, ensure the yellow LED is off. Finally, send *flowRate* and *totalDispersed* to the host PC. *flowRate*'s polarity further communicates the direction of flow to the user at the host PC.

For communication, the Arduino Serial (Arduino Serial Reference 2018) library is utilized, specifically print and println. This sends the strings, or variables represented as character strings, over the USB cable as a serial message. This can be viewed on the computer by any serial console.

ISR

The ISR needs to be designed as efficiently as possible, as it is blocking the main program and cannot be interrupted (by design). This could lead to starvation if the designer is not careful. The ISR's flow is depicted in Figure 5.

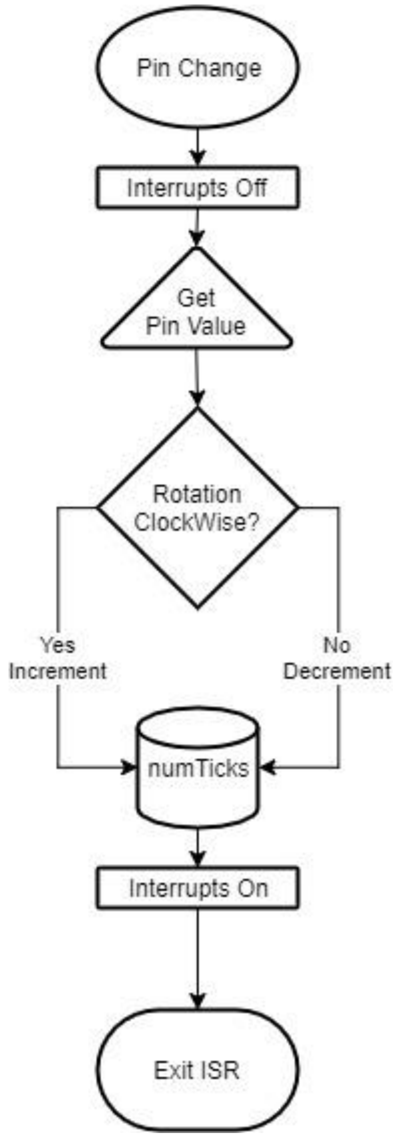

Figure 5: Common Interrupt Service Routine (ISR)

A corresponding ISR is kicked off any time the A or B pin is changed[1]. The logics are generally the same; based on the interrupt, the altered pin's value is sampled and stored. This stored value is the resting state of the event; using Table 1, a logic comparison can be constructed that determines, based on the final value of the pin and its companion's value, the direction of flow. Equation 5: 'A' Event Logic and Equation 6: 'B' Event Logic demonstrate this.

Equation 5: 'A' Event Logic

$$isCW = ((A\ \&\&\ !B)\ \ ||(!A\&\&B))$$

[1] Note that, while the logic is common, there are two differences between the ISRs: the Pin that fires the interrupt, and the logic to determine flow direction.

Equation 6: 'B' Event Logic

$$isCW = ((A \&\& B)\ ||(!A\&\&!B))$$

This can then be efficiently communicated back to the main program by incrementing the *numTicks* variable on clockwise flow and decrementing on counterclockwise flow.

This algorithm has a side effect; the overall “direction” of flow is the direction that had the most flow over the one second period between communications. For example, if 3 clockwise ticks and 1 counterclockwise tick happens in this period, *numTicks* will hold a value of +2. Overall, this is correct, but we lose some of the information within this period. This will reduce noise but may also filter out some bad behavior. The designer can tailor the “noise filtering” effect by decreasing the period on the updates.

Implementation

The schematic depicted in Figure 1 was wired onto the board with the help of the given breadboard. The USB cable was connected to the Host PC.

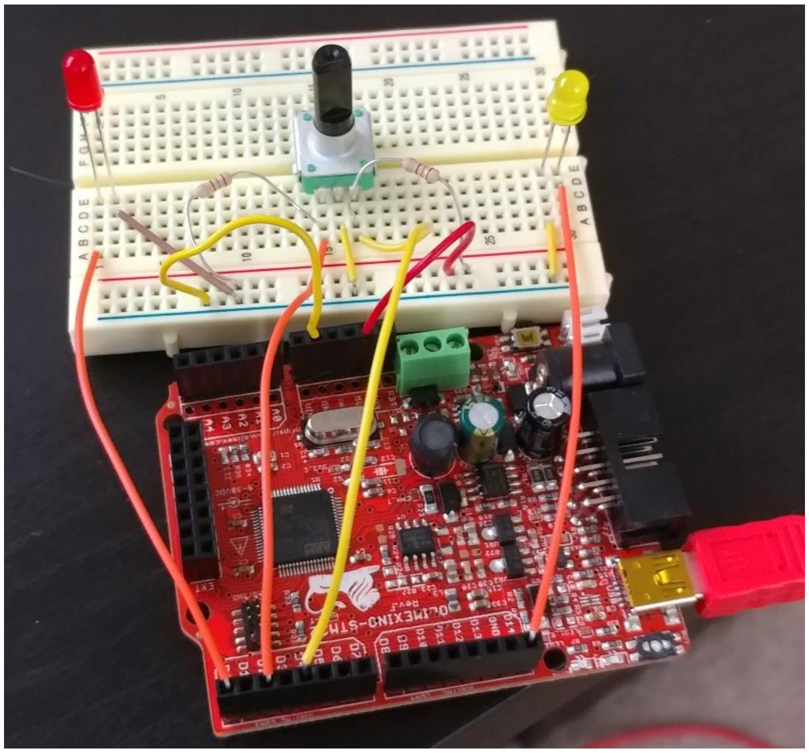

Figure 6: Schematic On-board

All the code is contained within one file, LAB3.ino. Arduino libraries are included by default in the Arduino IDE. After the code is compiled by the Arduino IDE, the IDE then uploads the binary to the board, which begins running after a short delay. The user then interacts by turning the quadrature encoder via the black pin in the center.

Test Plan

1) Wire the schematic
2) Compile the test Payload.
3) Plug the USB into the computer and board.
4) Upload the test payload
5) Test normal operation
    a) Rotate clockwise at various rates below max rate
    b) Confirm the following:
        i) The red LED is off
        ii) The yellow LED is off
        iii) The flow rate appears to match the rate of spin
        iv) The accumulated total makes sense
6) Test backflow operation
    a) Rotate counterclockwise at various rates below max rate
    b) Confirm the following:
        i) The red LED is off
        ii) The yellow LED is on
        iii) The flow rate appears to match the rate of spin
        iv) The accumulated total makes sense
7) Test overflow operation
    a) Rotate clockwise at rates above max rate
    b) Confirm the following:
        i) The red LED is on
        ii) The yellow LED is off
        iii) The flow rate appears to match the rate of spin
        iv) The accumulated total makes sense
8) Test overflow and backflow operation
    a) Rotate counterclockwise at rates above max rate
    b) Confirm the following:
        i) The red LED is on
        ii) The yellow LED is on
        iii) The flow rate appears to match the rate of spin
        iv) The accumulated total makes sense
9) If needed, adjust operation or add more debugging printouts and run again
10) Repeat this process until both tests are (a) fully implemented and (b) passing on the test board.

Test Results

Table 2: Lab 3 Test Results, the successful test matrix, was obtained after some trial and error to get the code in working order.

Table 2: Lab 3 Test Results

Test Number	**Pass/Fail**	**Comments**
1	Pass	Visual Inspection
2	Pass	Compilation
3	Pass	
4	Pass	Upload successful
5	Pass	Flow below Max Rate, positive for clockwise, LEDs off
6	Pass	Flow below Max Rate, negative for CCW, Yellow LED on.
7	Pass	Flow above max rate, positive for CW, Red LED on.
8	Pass	Flow above Max Rate, negative for CCW, both LEDs on.

Demonstration

A demonstration of the design was posted to YouTube (unlisted). Please use the following link:

https://youtu.be/CMOjtDIYD_U

Suggestions

Future Improvements to this Implementation

- Create macro that switches between board/PC calculations
    - Decision to offload can be made with a single line
- Experiment with period of update, find the best balance between noise cancellation/ability to report minor backflows

Future Improvements to Lab Description

- Correct part number for quadrature encoder
- Guidance on how to read this encoder's datasheet
- An encoder with a better datasheet in general

Code

The code can be viewed on the author's GitHub page, https://github.com/okeltw/EmbeddedMicrosystems/tree/master/Labs/Module3/lab2.

This release of the code was tagged as 'Lab2-Rev2', to preserve this code in case of future updates as noted in the Suggestions section.

Discussion

The following question was posed at the end of the lab description (Stakem and Crum n.d.):

> *Here's a real-world application ripped from the world news of Fall, 2015. The "Press" says this required sophisticated software. I say it only takes a couple of lines of code.*
>
> *Here's the scenario, A major German auto manufacturer wants to bypass emissions testing in a clever manner. They speculate they could have two tables of engine control data, one for operation of the car, giving max performance and fuel mileage, and another to meet the emissions specs. What they need is a way to determine when an emissions test is being conducted. A clever auto mechanic suggests that since the emissions test is done on a dynamometer, only the front (drive) wheels are turning. In normal operation, all four wheels are turning. So, using the car's quad encoders, present at each wheel, we could read 1 and 2 (front wheels) and 3 and 4 (back wheels). These are normally used for [antilock] braking and traction control. But our algorithm is, when the front wheels are turning, but not the back, switch to "special" engine control table 1. Otherwise use standard engine control table 2.*
>
> *Of course, this is highly illegal, and will result in massive fines, not to mention bad air... So,* ***how many lines of code (LOC) would it take you to do this****?*

Since it is difficult to judge how large the program is to begin with, I will interpret this question as *additional* SLOC; my answer (speculative and without testing) is one.

The "technically correct" approach, for discussion's sake, says that any number of functions can appear on one line; many languages, e.g. C, don't require any newline. So, *technically*, we could put all the source in this one line – including the change for the two lookup tables.

…but this answer is boring. Instead, assume that there is a pointer variable to the array that holds the table entries. In such a case, all the math for the array could remain the same; we only need the initial address to change, ensuring that the "fixed" data resides at the same offsets from this address. Therefore, I propose the following addition:

```
tablePtr = (backWheelR.numTicks==0)&&(backWheelL.numTicks==0) ?
           &emissionsTable : &performanceTable;
```

This logic uses the same logic my lab solution does in terms of *numTicks*. If both read 0 ticks, assuming this point in the code it is known the front wheels are moving (or append the appropriate logic to the check), then the wheels aren't moving[2], and the tablePointer is updated correctly. Subsequent code, which works exactly the same, grabs the fixed data for the emissions test.

[2] Exception: they move backwards and forwards the same amount, for a tick offset of zero. But, this is an edge case and hard to test for, so this risk could be accepted.

References

2018. *Arduino Micros Reference.* Accessed Septermber 24, 2018. https://www.arduino.cc/reference/en/language/functions/time/micros/.

2018. *Arduino pinMode Reference.* https://www.arduino.cc/reference/en/language/functions/digital-io/pinmode/.

2018. *Arduino Serial Reference.* Accessed September 23, 2018. https://www.arduino.cc/reference/en/language/functions/communication/serial/.

2016. "Olimexino-STM32 development board User's manual." Olimex Ltd, September.

2012. "PM0075 STM32 Flash Programming Manual." STMicroelectronics, August.

Stakem, Pat, and Gary Crum. n.d. "Module 4 Assignment." *JHU Blackboard.* Accessed September 23, 2018. https://blackboard.jhu.edu/bbcswebdav/pid-5545654-dt-content-rid-44238260_2/courses/EN.525.615.81.FA18/lab%203%20module%204%20quad%20encoder%20flow%20rate%20sensor.pdf.

2015. "STM32F103xB Reference Manual." STMicroelectronics, August. 34.

2018. "TT Electronics EN12-HN22AF25." *Mouser Electronics.* Accessed September 23, 2018. https://www.mouser.com/datasheet/2/414/Datasheet_RotaryEncoder_EN12Series-1132176.pdf.