Lab 4: Real Time Clock

Taylor Okel

10/6/2018

Update: Correcting previous submission with custom RTC controller, descriptions of the controller design, and clearer graphics

Table of Contents

Table of Figures

Problem Statement

Description

Create a design that implements the real time clock (RTC) to send periodic time updates. The design should provide a mechanism for updating the date and time.

In addition to this loop, a higher-priority task of providing 1024 clock cycles on an external signal should be included. (Stakem and Crum n.d.)

Requirements

The following requirements were derived from the Lab 4 description (Stakem and Crum n.d.)

- Use the Real Time Clock to accurately determine time
- On power-on, print a message (via Serial) with current date and time
- Print a message (via Serial) that displays the time every second
- On external signal, provide 1024 clock cycles (highest priority)

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
- Part 3 – 1024 clocks – is a little unclear of what the acceptance criteria is. It is assumed that the acceptance criteria is simply 1024 clock cycles generated on demand

Equipment

Hardware

- Olimexino-STM32 Development Board (henceforth "board")
    - STM32F103RBT6 Arm Processor (Olimex 2016)
- USB programming cable
- Programming Workstation (henceforth "computer")
    - Windows 10 operating system
    - Arduino IDE
- Pushbutton
- Yellow LED
- Various jumper cables
- Breadboard

Architecture

An external interrupt and some method to observe the clock is needed. Therefore, a pushbutton to control the interrupt and Yellow LED to display the clock signal are to be wired to the board.

In this design, the Interrupt pin is pulled up to a known 3.3V state. The pushbutton is connected to this junction and to ground. When pressed, the pushbutton shorts the pullup to ground, driving the interrupt pin to zero. When this port, D0, is attached to an interrupt on the falling edge this configuration generates an interrupt on button press.

Port D1 is then connected to an LED, which is then connected to ground. When driven high, it will light; when driven low, it will turn off. Digital writes to this pin will therefore model a clock signal.

These considerations are shown in the wiring diagram (Figure 1).

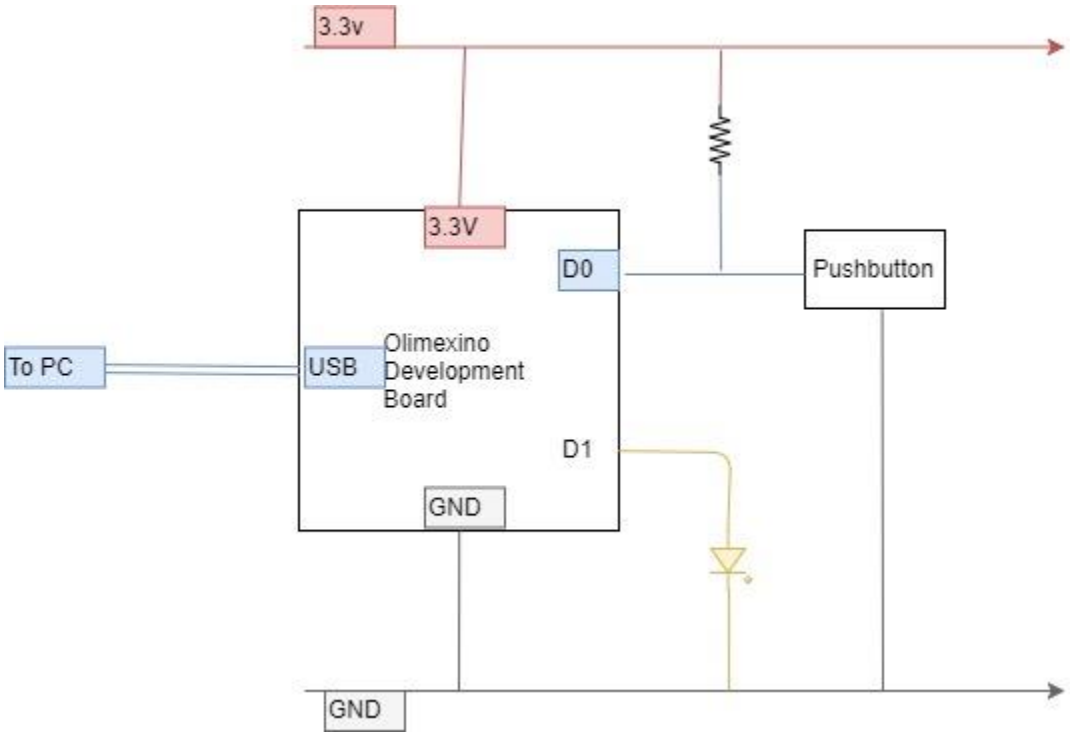

Figure 1: Wiring Diagram

Design

The Real Time Clock (RTC)

Working with the RTC requires register manipulation, which is abstracted through the custom RTC Controller. This design requires a collection of functions to achieve the goal of using the RTC to accurately measure time and perform interrupts. To this effect, the following methods were created:

- rtcInit – Initializes the RTC core and begins operation
- rtcGetCount – gets the time value as a uint32
- rtcSetCount – sets the time value as a uint32
- rtcAttachSecondsInterrupt – enables the second interrupt and assigns a function as the handler
- rtcSetPrescaler – sets the prescaler to the specified value
    - A value of 0x7fff with the LSE (32.768KHz) will cause the second interrupt to fire at exactly one second

In addition, a custom register map for the RTC functions must be created. This was built using the STM32 Reference Manual's section on the RTC in section 18 (ST 2018). Several inline functions that abstract sync/config completion polling and configuration mode enter/exit further aid in abstraction of the underlying process.

These are featured in the RTCController header file (Okel 2018). The implementations of the following controller are in the RTCController code file.

LibMaple

To help accomplish these functions, the Maple Library (Leaf Labs 2014) helper definitions and functions were used. The headers for these functions are included as part of the Arduino STM 32 installation.

- bitband - provides the bb_peri_[get/set]_bit helper functions for writing to registers
    - This simplifies the register manipulation process to a known-good method
- bkp - provides addresses to the backup registers and functions to initialize the system
    - The register map structure provided a useful pattern that will be modeled in future labs to abstract the addresses in
- libmaple_types - provides type definitions, including a void function pointer used for interrupt handler assignment and a __IO type that essentially marks variables as volatile
- nvic – register map, a function to initialize the nested vector interrupt controller (NVIC), and a function to enable internal interrupt vectors
- pwr – provides addresses for enabling power to sections of the chip (bkp in particular) and an initialization function
- rcc – (Reset and Clock Control) a function to initialize the LSE clock
- util – Macros to simplify specific bit manipulation in registers

rtcInit

To use the RTC, several actions must first be performed. First, the APB1 bus needs to be enabled along with the backup registers. This will allow the RTC to use the Low-Speed External (LSE) clock, running at 32.768 KHz, to increment its internal counter representing time since epoch (1970). Once this is accomplished, the RTC itself needs to be enabled.

ST provides an application note (ST 2009) that aided in the definition of this process. In section 2 of this document, the following list of steps is defined:

On startup, follow the steps below to configure the RTC to generate an interrupt every second:

- Enable the APB1 backup domain and power interface clocks by writing the BKPEN and PWREN bits to '1' in the RCC_APB1ENR register
- Enable access to backup domain by writing the DBP bit to '1' in the PWR_CR register
- Enable the LSE clock by writing the LSEON bit to '1' (also write LSEBYP to "1" when the external clock has to be bypassed)
- Poll the LSERDY flag in the RCC_BDCR register until the LSE clock is ready (if the external crystal is used as the clock source).
- Select LSE as the RTC clock source by writing '01' to the RTCSEL bits in the RCC_BDCR register.
- Enable the RTC clock by setting the RTCEN bit in the RCC_BDCR register
- Poll the RSF bit in the RTC_CRL register until the RTC registers are synchronized (if a 50/60 Hz external clock source is used this step may take up to a minute to complete)
- Poll the RTOFF bit in the RTC_CRL register until the last operation on the RTC registers is over
- Enable the RTC second global interrupt by setting the SECIE bit in the RTC_CRH register
- Wait for the last task to complete
- Set the RTC prescaler value using the following formula:

  $f_{TRCLK} = f_{RTCCLK}/(PRL[19:0]+1)$, where:
  - f_{RTCCLK} is the input clock frequency
  - f_{TRCLK} is the time base generated from the prescaler block

  For example, if an external 32.768 kHz (32 kHz) crystal oscillator is used, set the prescaler to 32767. If an external 50 Hz supply is used set the prescaler value to 49.

Figure 2: RTC Init Steps from App Note

For the steps enabling the backup domain and clock, the LibMaple functions bkp_init and bkp_enable_writes simplify this process. Register and bit definitions from the RCC function are used to enable the RCC clock. Finally, the RCC is set to enable the RTC. Once configured, wait for clock synchronization and any RTC configuration write operations to finish.

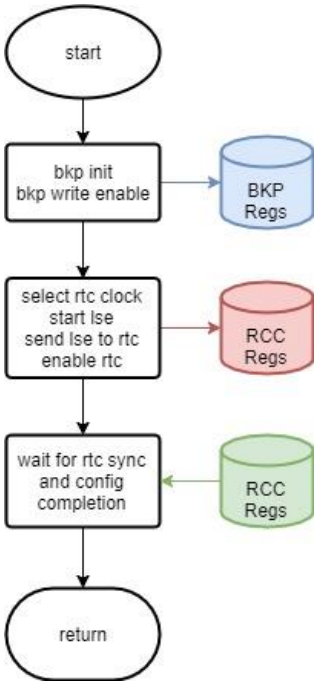

Figure 3: RTC Init Flowchart

rtcGetCount

The RTC must be initialized prior to this function. The RTC's count registers (note that there are two 16-bit registers that combine into a 32-bit representation of time. To get the count, first confirm that the clock is in sync and wait for any configuration operations to finish. Once these conditions are satisfied, combine the upper (CNTH) and lower (CNTL) register values by shifting the upper by 16 and bitwise-or'ing the lower register. This must be returned as at least a 32-bit data structure; a uint32 is used in this case.

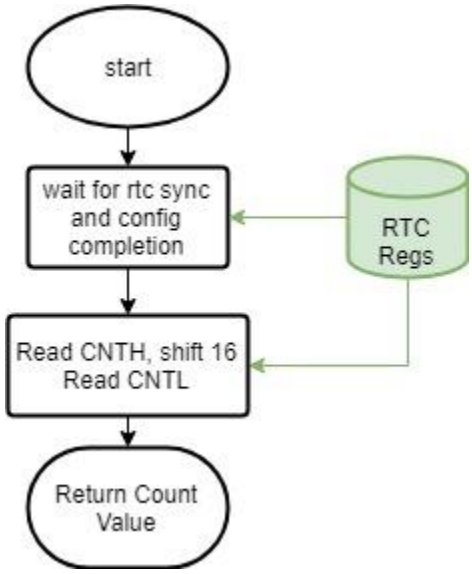

Figure 4: rtcGetCount Flowchart

rtcSetCount

The RTC must be initialized prior to this function. The RTC's count registers (note that there are two 16-bit registers that combine into a 32-bit representation of time. To set the count, the RTC must be synchronized and any previous configurations must be complete. The controller then sets the RTC into config mode and writes the values appropriately into CNTH and CNTL. The RTC then exits config mode, writing the count value. Wait for the write operation to finish before returning.

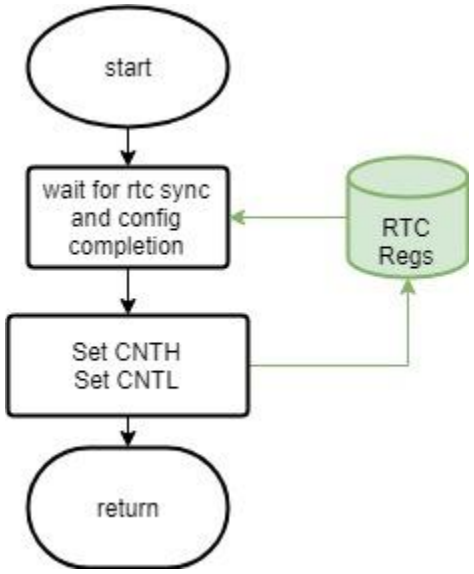

Figure 5: rtcSetCount Flowchart

rtcAttachSecondInterrupt

The RTC must be initialized prior to this function. The RTC's control register high (CRH) provides bits to enable interrupts. By setting the second bit, the RTC will fire an interrupt every second (note that this will match a real second if and only if the prescaler is set to an appropriate value). With the interrupt enabled, the interrupt must be handled; the __irq_rtc function, which originally responds to an interrupt, can be overloaded to achieve this by calling the handler function, and clearing the interrupt bit when the operation is complete.

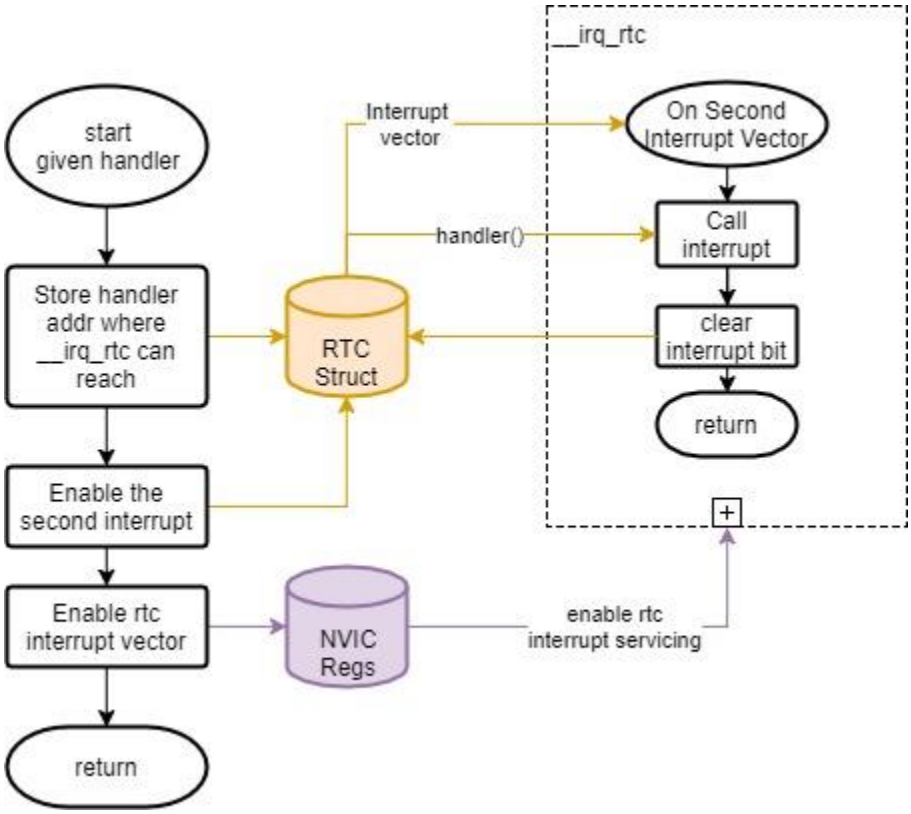

Figure 6: rtcAttachSecondInterrupt Flowchart

rtcSetPrescaler

This process is the same as rtcSetCount but targets the prescaler register. Note that the prescaler is only 20 bits, so the upper 12 bits of a 32-bit value will be discarded.

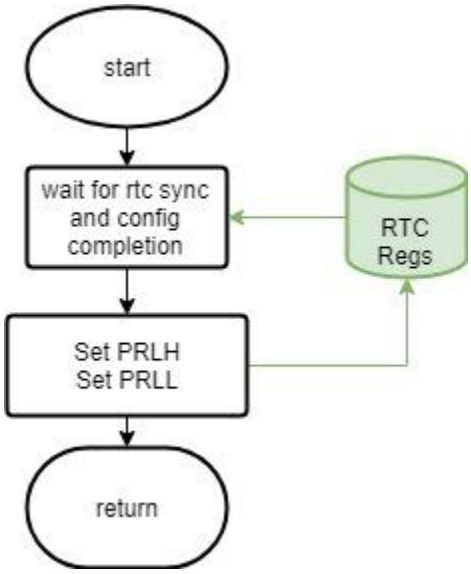

Figure 7: rtcSetPrescaler Flowchart

Program Design

The bulk of the program will be controlled by two ISRs and one polling loop. The first ISR will activate every second and display time. The second ISR will activate on button press and send the clock signal. Finally, the polling loop will monitor the serial line.

This is designed based on real-time need. The Serial line should have associated buffers and is a User IO method. Therefore, the timeliness is not important. The RTC has an interrupt on second option, making an ISR an easy to implement choice; since it is not time-critical, it is interruptible. Finally, the button press was designated as high priority. It will therefore need an ISR and is non-interruptible to meet its timing requirements. Figure 8 demonstrates this design in flowchart form. Note that dashed lines denote interrupt attachments. The print time and clock generator handlers are discussed in more detail below.

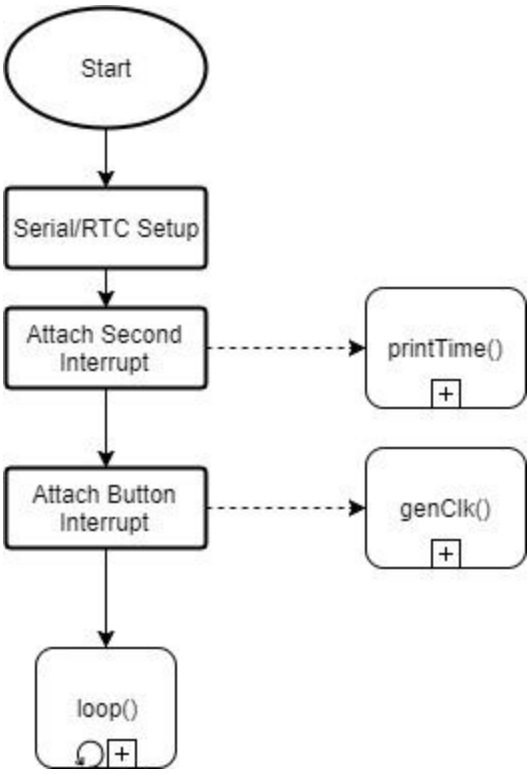

Figure 8: Main Flowchart

The RTC Setup uses the rtcInit function to enable the RTC, then rtcSetPrescaler to set the prescaler to fire interrupts at exactly one second, based on the LSE rate, and rtcSetCount to set the time to the current time. After this setup, the second interrupt is attached to the handler via

rtcAttachSecondInterrupt and the clock generator is attached to the external interrupt, fired by a button, using Arduino's attachInterrupt method (Arduino 2018). At this point, it enters the main loop.

Main Loop: User Input

As the lowest-priority task, user input will be polled in the main loop. As the data flows in, it is buffered; when processor time is available, this data is read as characters. Each of these characters must be converted into integers and accumulated into a single value that represents time since epoch, an unsigned 32-bit integer (uint32). This value can then be written into the RTC's counter to change the current time representation. After communicating this data, the controller returns to polling the serial interface. This design is depicted in Figure 9.

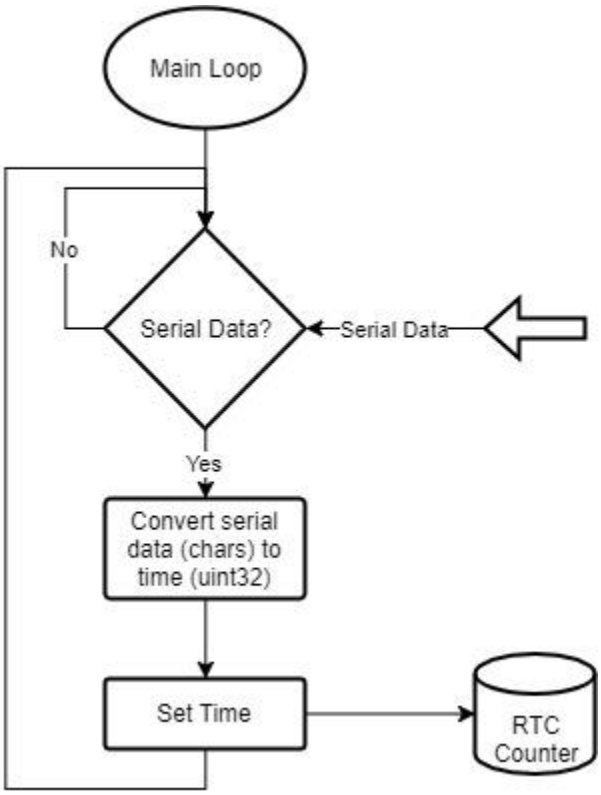

Figure 9: Main Loop (Serial Input)

For this portion, a custom RTC Controller module is used to write the RTC (explained in rtcSetCount) and Arduino's Serial library (Arduino 2018) to read serial input. The Arduino Serial API provides a Read method, which returns a single character from the serial buffer. As stated before, this character value must be converted to integer value.

The drawback of this design is the potential to starve this loop due to the higher priority interrupts. For example, the button could be pressed at a high enough frequency to never return to this loop. The serial buffer would then overflow, and some data would be lost. If only partial data is lost, there would be no way to

recover. However, since this is user input, the data will be relatively slow to arrive, so this situation would be difficult to achieve. In addition, the user would be able to correct the failure when it became apparent.

The Second Interrupt: printTime()

This function is called on the RTC's second interrupt. It immediately gets a uint32 representation of time from rtcGetCount. This representation is converted to a human-readable format through a switch statement, which assigns integers based on character value. This is then sent via Serial connection (Arduino 2018) to the host PC for display. Figure 10 depicts this process.

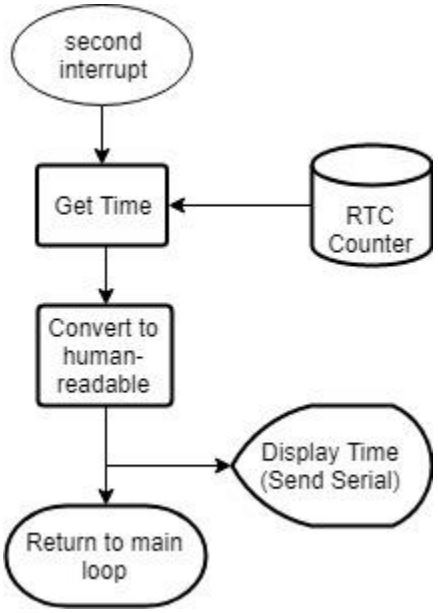

Figure 10: Second Interrupt Handler

The Button Interrupt: clkGen()

This interrupt is fired by the user pressing a button, connected to the board as described in Architecture. As this is a pullup, the interrupt fires on falling edge. The handler routine the generates a clock signal by flipping a bit connected to the output pin. Per the specification in the lab description (Stakem and Crum n.d.), the design needs 1024 clock cycles, so 2048 flips need to occur. Finally, as stated in Program Design, this is the highest priority; to prevent interruption from the second interrupt, the loop is wrapped by the interrupt disable/enable functions provided by Arduino (Arduino 2018). This process is demonstrated via flowchart in Figure 11.

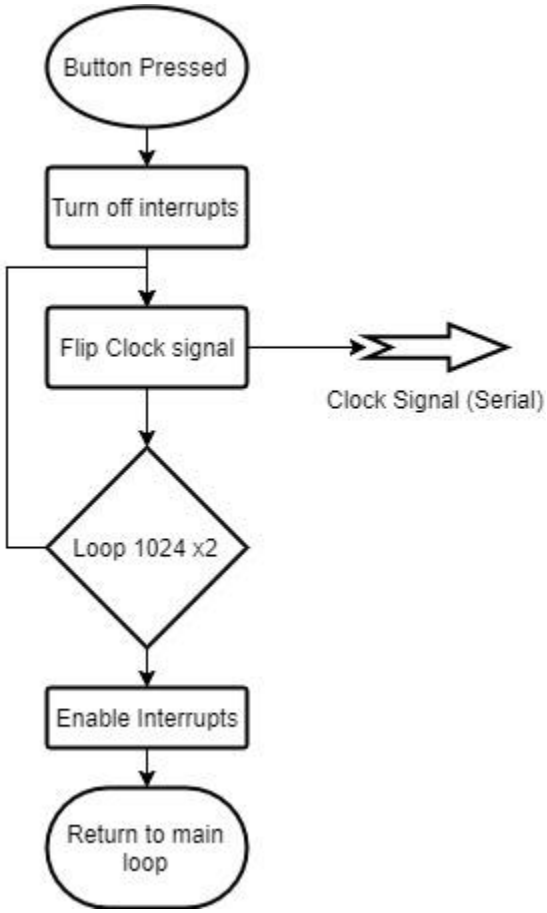

Figure 11: Button Interrupt Handler

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.

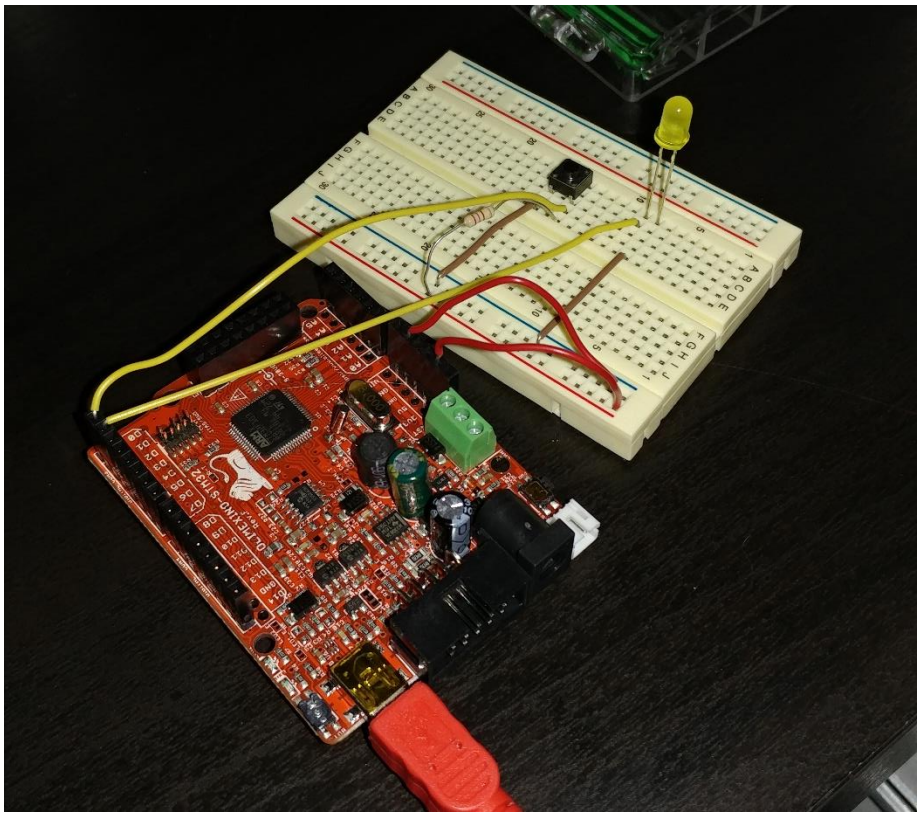

Figure 12: Schematic On-board

The main code is loaded through Lab4.ino, with the controller code provided by RTCController.c and RTCController.h. 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 entering values into the Arduino Serial Monitor or pressing the pushbutton on the breadboard.

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 second interrupt by letting it run for a few seconds
6) Test ability to set
    a) Use epoch converter (Misja.com 2018) to find time since epoch
        i) Enter date and time into "Human date to Timestamp" fields
        ii) Click button
    b) Upload this time
    c) Confirm that the time displayed matches the time in the fields
7) Test incorrect data in Serial
    a) Type non-integer characters into serial and send
    b) Confirm that the time does not change
8) Test button interrupt
    a) Start program and wait a few seconds; observe timer
    b) Press button
        i) Confirm that the LED (very) momentarily flashes
        ii) Confirm that the timer pauses during this timeframe
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 1: Lab 4 Test Results, the successful test matrix, was obtained after some trial and error to get the code in working order.

Table 1: Lab 4 Test Results

Test Number	Pass/Fail	Comments
1	Pass	Visual Inspection
2	Pass	Compilation
3	Pass	
4	Pass	Upload successful
5	Pass	
6	Pass	
7	Pass	
8	Pass	Flashes too quickly to see; however, it is much dimmer signifying that the clock signal is generate.

Suggestions

Future Improvements to this Implementation

There are several improvements that I would make to my implementation. However, the current implementation is submitted, as it fully meets the requirements as set out in the lab description.

- Some input sanitization on serial input – it currently does not detect invalid characters and reject them. This could be further extended by notifying the user.
- Better time input – currently, only an integer value representing time since epoch is accepted. Some method to accept multiple input formats could be implemented.
- RTCController could be further fleshed out – only the functionality needed for this lab was designed. Functions to fully control and abstract the RTC could be implemented for future use
- Better testing of the clock signal – send this to the host PC or other device, along with some known data pattern; have a program ready on the other side that reads this pattern and verifies operation.
    - This could be further extended by replacing the button with a signal from this other device that starts the data transfer.

Future Improvements to Lab Description

- If multiple pdfs are uploaded, but specific constraints apply, call out the PDF by name rather than description. Or, add the constraint to the PDF.
- Some of the improvements I stated above – such as the better clock signal testing – would make an interesting addition to this, or another, lab.

Code

The code can be viewed on the author's class GitHub page.

https://github.com/525-615-81-FA18/labs-okeltw

The code was tagged to provide snapshots into the development process:

- Lab4_Rev0 – Initial design with RTClock library
- Lab4_Rev1 – First pass without RTClock, with some bugs left in it
- Lab4_Rev2 – Final code at time of this document

References

Arduino. 2018. *attachInterrupt.* https://www.arduino.cc/reference/en/language/functions/external-interrupts/attachinterrupt/.

—. 2018. *Interrupts.* https://www.arduino.cc/reference/en/language/functions/interrupts/interrupts/.

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

Leaf Labs. 2014. *LibMaple APIs.* 1 15. Accessed 2018. http://docs.leaflabs.com/static.leaflabs.com/pub/leaflabs/maple-docs/0.0.12/libmaple/apis.html.

Misja.com. 2018. *Epoch Converter.* Accessed 9 30, 2018. https://www.epochconverter.com/.

Okel, Taylor. 2018. *Module 5 Git Repository.* October 5. https://github.com/525-615-81-FA18/labs-okeltw/tree/master/Labs/Module5.

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

ST. 2009. "AN2821 Application Note." *st.com.* April. https://www.st.com/content/ccc/resource/technical/document/application_note/b0/34/9f/35/17/88/43/41/CD00207941.pdf/files/CD00207941.pdf/jcr:content/translations/en.CD00207941.pdf.

—. 2018. "STM32F103xB Reference Manual." STMicroelectronics, April. 34.

Stakem, Pat, and Gary Crum. n.d. "Lab 4: Real Time Clock." *Blackboard.* Accessed 10 2, 2018. https://blackboard.jhu.edu/bbcswebdav/pid-5545665-dt-content-rid-44238577_2/courses/EN.525.615.81.FA18/Module%205/Lab%204.pdf.