ECE 153B – Winter 2024

Sensor and Peripheral Interface Design

Lab 2 – Step Motor, Interrupts, and RTC

Deadline: **Jan 29, 2024, 10:00 PM**

Objectives

1. Understand full and half stepping to control the speed and position of a stepper motor.
   a) Use GPIO pin in output mode
   b) Control the stepper motor through the GPIO pin
2. Understand I/O interrupts
   a) Understand the basic procedure of interrupt handling
   b) Understand auto stacking and unstacking of the interrupt handling process
3. Understand the advantages of using Real Time Clock (RTC)
   a) Program RTC and use it to generate periodic alarm interrupts
   b) Perform clock calibration
   c) Use RTC alarm interrupts to update an LED

Grading

Part	Weight
Part A	20 %
Part B	20 %
Part C	30 %
Checkoff Questions	30 %

You must submit your code to the submission link on Gradescope by the specified deadline. In the week following the submission on Gradescope, you will demo your lab to the TA.

Please note that we take the Honor Code very seriously, do not copy the code from others.

Contents

1 Necessary Supplies

- STM32L4 Nucleo Board
- Type A Male to Mini B USB Cable
- Breadboard & Jumper Wires
- 28BYJ-48 5v stepper motor + ULN2003 driver board

2 Lab Overview

A. Rotate the stepper motor shaft exactly 360 degrees clockwise or counter-clockwise via both half and full stepping, and adjust the speed of rotation.

B. In Lab 1 you used polling to toggle the green LED for user button click. However, you will use interrupts instead of polling.

C. Understand how the RTC is configured and how date/time information is stored. Calculate the current time and use RTC alarms to toggle LED at set times.

3 Part A - Stepper Motor

In this part of the lab, you will perform the 4-step sequence of full-stepping or the 8-step sequence of half-stepping to rotate the 28BYJ-48 5v stepper motor. This is accomplished by using output GPIO pins to control the current through the two coils of the motor, through the ULN2003 driver board. The motor and driver board are shown below.

Figure 1: Stepper Motor and Control Board (Model: 28BYJ-48).

The following is the hardware information.

Motor Model	**28BYJ-48**	Number of phases	2
Rated voltage	5V DC	Gear reduction ratio	1/64
DC resistance per phase	$50\Omega \pm 7\%(25°C)$	Pull in torque	>300gf.cm / 5VDC 100pp

3.1 Introduction

A stepper motor rotates by moving a rotor to evenly-spaced positions in a fixed sequence of steps that form a complete rotation. The rotor is moved to different positions by adjusting the current in two coils in fixed locations around the rotor. The two most common and preferred sequences (also known as driving methods) are called *Full-Stepping* and *Half-Stepping*, which split the rotation into 4 and 8 steps respectively. These sequences are shown in Figures 2 and 3 below.

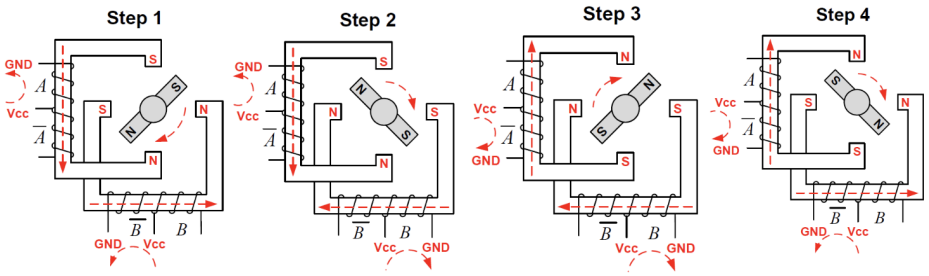


Figure 2: Full-Stepping Sequence

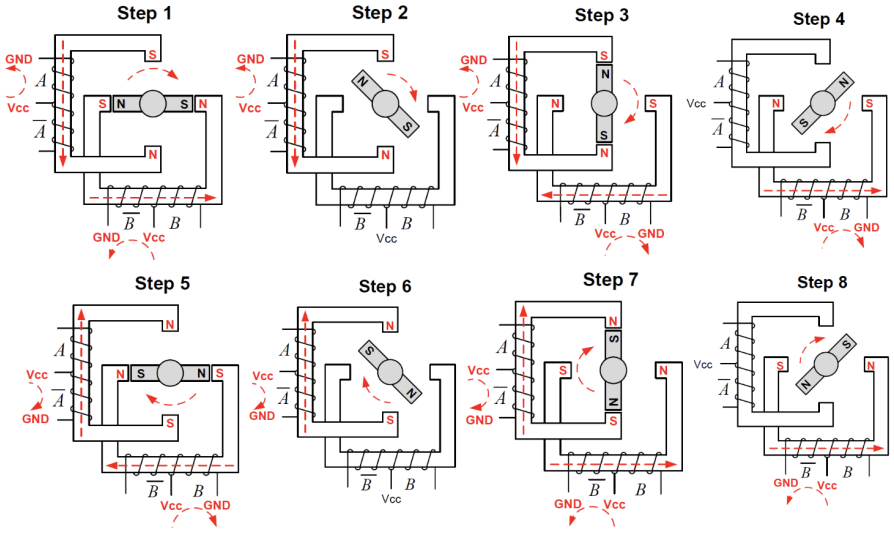


Figure 3: Half-Stepping Sequence

Note that it doesn't matter where you actually start each sequence, you just have to complete a full rotation. The sequences can also be different combinations of rotor angles, so long as they are evenly spaced (based on the number of steps) and form a complete rotation.

Splitting the rotation into just 4 or 8 steps means the rotations are not very smooth, and the motor can only be in specific positions. To fix both of these problems, stepper motors have significant gear reduction to decrease the number of rotations of the output shaft for each rotation of the input shaft. This means the differences between steps of the sequences are very small at the output, leading to smooth rotation and precise control of the position of the output shaft.

We can find the gear reduction ratio by counting the number of teeth in each of the gears between the input and output in the stepper motor. The gears of the 28BYJ-48 5v stepper motor are shown in Figure 4 below (note that some gears are out of place for better visual clarity).

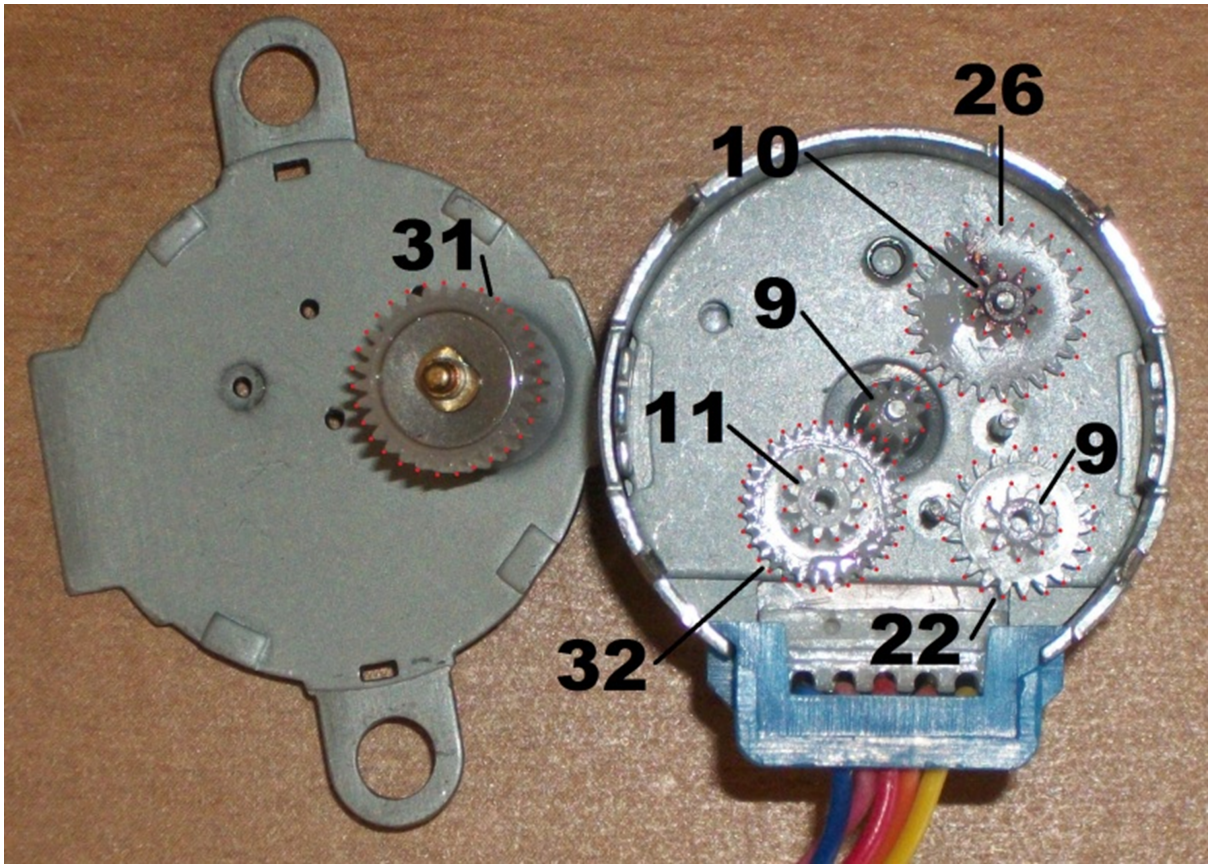


Figure 4: Internal gears of the stepper motor (image from forum.arduino.cc)

In this case, the gear ratio is:

$$\frac{31 * 32 * 26 * 22}{11 * 10 * 9 * 9} = 63.68395$$

If the output shaft rotates 1 resolution (gear with 31 teeth in the figure), the internal shaft (gear with 9 teeth in the middle) must rotate approximately 64 resolutions.

Full-stepping

- Internal motor: 32 steps per revolution
- Gear reduction ratio: $1/63.68395 \approx 1/64$
- Thus, it takes $32 * 64 = 2048$ steps, or 512 repeats of the 4-step pattern, for the output shaft to complete a rotation

Half-stepping

- Internal motor: 64 steps per revolution
- Gear reduction ratio: $1/63.68395 \approx 1/64$
- Thus, it takes $64 * 64 = 4096$ steps, or 512 repeats of the 8-step pattern, for the output shaft to complete a rotation

To rotate a specific degree, your program should do the math to convert the value in degrees to a number of steps, and then do the rotation.

3.2 Lab Exercise

Interfacing the stepper motor requires four pins, excluding +5V and ground. We will be using GPIO **PC 5**, **PC 6**, **PC 8**, and **PC 9**.

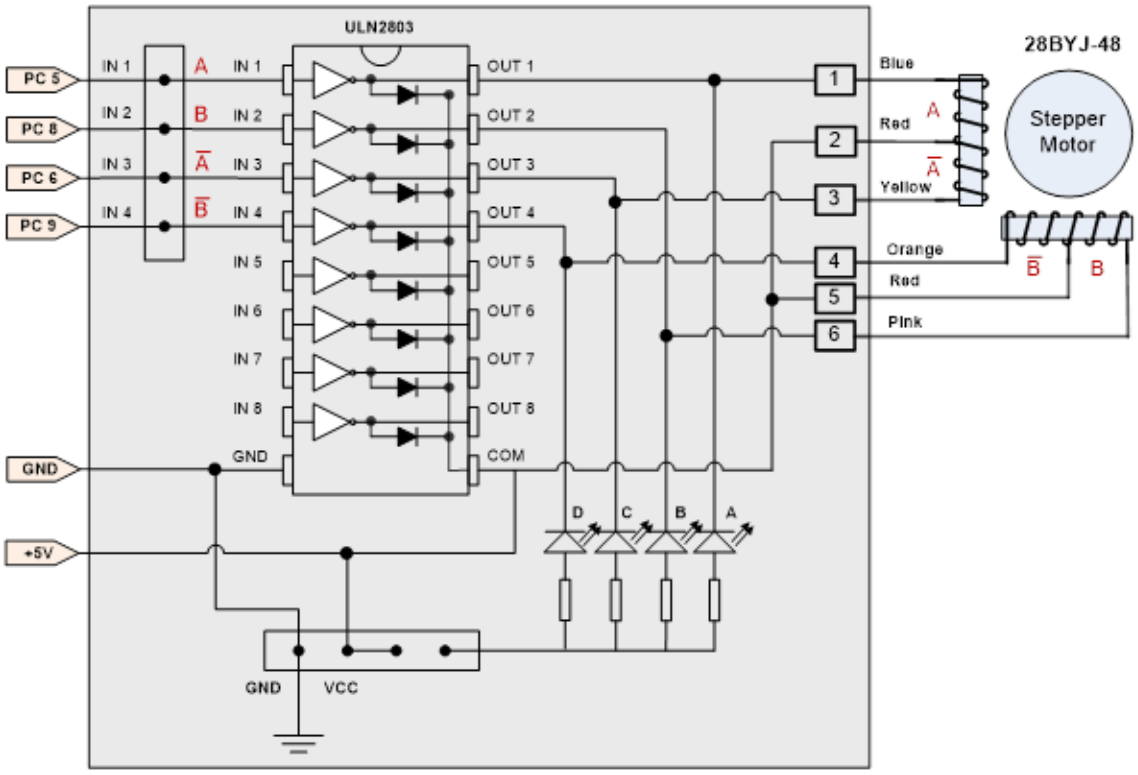
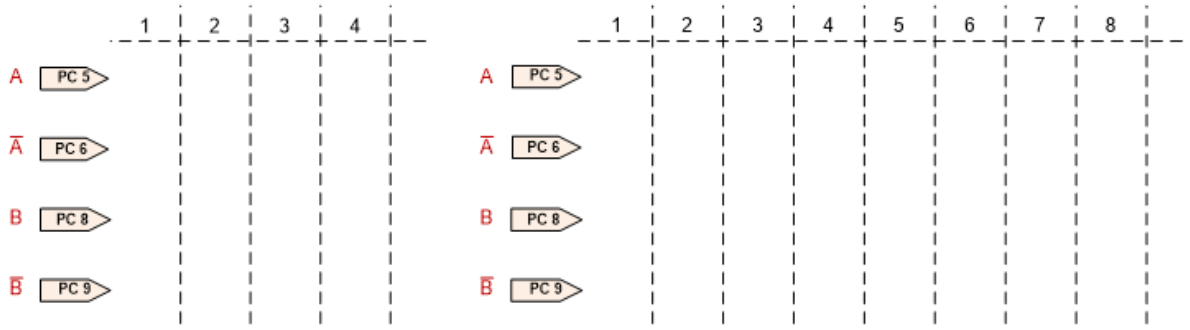


Figure 5: Connection diagram of the motor driver board (Model: 28BYJ-48).

Use Figure 5, as well as Figures 2 and 3, to sketch out below what the waveform should be on the pins for a full-stepping sequence and a half-stepping sequence.



Rotation Direction and Speed

a. How would you change the rotation speed of a stepper motor?

b. How would you reverse the rotation direction?

Warning: Motor Overheating
The motor constantly draws electrical currents and will be overheated if you leave the power on for an extended period. **Disconnect the 5V power (Vcc) to the Darlington array if you are not actively debugging/testing it.**

Use the following steps to initialize the GPIO pins **PC 5**, **PC 6**, **PC 8**, and **PC 9** on the Nucleo board. Refer to Section 9.4 in *STM32L4x6 Reference Manual* for details about registers and their bits.

1. Enable the clock for GPIO port C.
2. Configure pins 5, 6, 8, and 9 as output.
3. Set the output speed of the pins to fast.
4. Set the output type of the pins to push-pull.
5. Set the pins to no pull-up, no pull-down.

Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OSPEEDR	OSPEED15[1:0]		OSPEED14[1:0]		OSPEED13[1:0]		OSPEED12[1:0]		OSPEED11[1:0]		OSPEED10[1:0]		OSPEED9[1:0]		OSPEED8[1:0]		OSPEED7[1:0]		OSPEED6[1:0]		OSPEED5[1:0]		OSPEED4[1:0]		OSPEED3[1:0]		OSPEED2[1:0]		OSPEED1[1:0]		OSPEED0[1:0]	
Value																																

Tip: It can be hard to see the position of the motor while testing. A piece of tape stuck to the shaft can make it easier to see what direction it is pointing.

4 Part B – Interrupts

In this part of the lab, you are to implement the LED toggle. Specifically, for every **odd-numbered** click set the green LED and for every **even-numbered** click turn-off the green LED. However, you are required to use interrupts to handle button presses instead of polling.

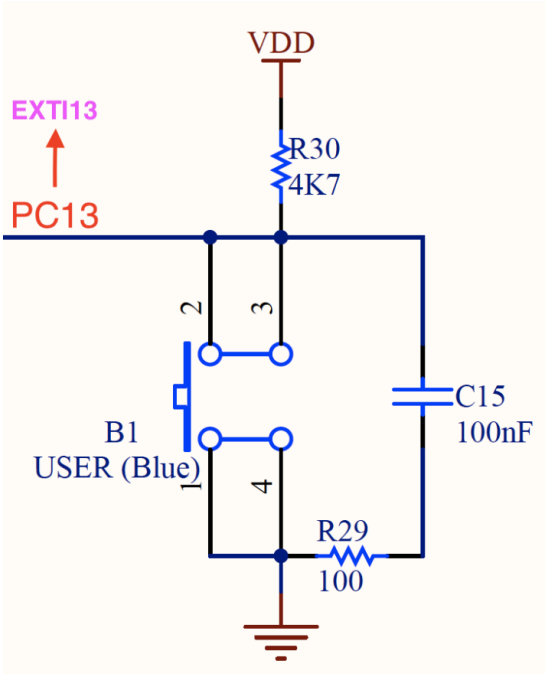


Figure 6

First, you need to set up the GPIO pins as in lab 1 (enabling the necessary clocks and configuring the desired GPIO pins as inputs). Next, you need to set up the **external interrupt** (EXTI) for GPIO pin k (the pin that corresponds to the button). See Figure 6 to see which EXTI line is connected to the GPIO pin for the user button input. The following steps outline the general procedure for doing this.

1. Configure the `SYSCFG` external interrupt configuration register (`SYSCFG_EXTICR`) to map GPIO port j pin k to the EXTI input line k. For pins 0-3 the configuration register is `SYSCFG_EXTICR1`, pins 4-7 configuration register is `SYSCFG_EXTICR2`, pins 8-11 configuration register is `SYSCFG_EXTICR3` and pins 12-15 configuration register is `SYSCFG_EXTICR4`.

```
SYSCFG->EXTICR[3] &= ~SYSCFG_EXTICR1_EXTIk;
SYSCFG->EXTICR[3] |= SYSCFG_EXTICR1_EXTIk_Pj;
```

   Note: Do not forget to enable the system configuration controller (`SYSCFG`) which is responsible for managing external interrupt line connections to the GPIOs.

```
RCC->APB2ENR |= RCC_APB2ENR_SYSCFGEN;
```

2. Select a signal change that will trigger EXTI line k. The signal can be a rising edge, a falling edge, or both. This is configured through the EXTI rising edge trigger selection register (`EXTI_RTSR1` or `EXTI_RTSR2`) and the EXTI falling edge trigger selection register (`EXTI_FTSR1` or `EXTI_FTSR2`). Setting the bit that corresponds to input line k in the register to `0` disables the trigger and setting it to `1` enables the trigger. For this lab, use a falling edge trigger.

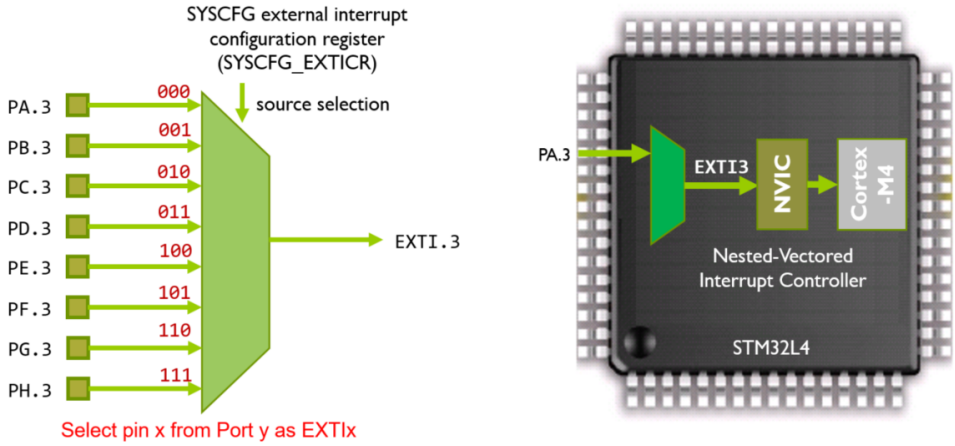


Figure 7

```
EXTI->RTSR1 |= EXTI_RTSR1_RTk;
EXTI->FTSR1 |= EXTI_FTSR1_FTk;
```

3. Enable the EXTI for input line k. This is done through the EXTI mask register (`EXTI_IMR1` or `EXTI_IMR2`). Setting the bit that corresponds to input line k in the register to `0` disables the EXTI and setting it to `1` enables the EXTI.

```
EXTI->IMR1 |= EXTI_IMR1_IMk;
```

4. Configure the enable and mask bits that control the NVIC interrupt channel corresponding to EXTI input line k, ($k < 5$). For interrupt lines with ($k >= 5$), different masks are used. 5-9 use the same IRQn, and 10-15 use the same IRQn. In addition, set the interrupt priority.

```
NVIC_EnableIRQ(EXTIk_IRQn);
NVIC_SetPriority(EXTIk_IRQn, 0);
```

5. Write the interrupt handler for EXTI input line k. The function name of the interrupt handler can be found in the startup assembly file `startup_stm32l476xx.s`. For example, the handler for EXTI0 is called `EXTI0_IRQHandler()`.

   The EXTI pending register (`EXTI_PR1` or `EXTI_PR2`) records the source of the interrupt. Note that in the interrupt handler, you must clear the corresponding pending bit (so that future interrupts can occur). To do this, you can write a 1 to the corresponding bit of the pending register.

```
EXTI->PR1 |= EXTI_PR1_PIFk;
```

5 Part C – Real Time Clock

5.1 Introduction

Internal clocks have an accuracy of 1.5%, or 1500 PPM (parts per million), where 1 PPM is 0.0001%. In one year, there are approximately

$$365 \text{ days} \times 24 \text{ hours} \times 60 \text{ minutes} \times 60 \text{ seconds} = 31536000 \text{ seconds}$$

A clock with an accuracy of 1 PPM would gain (or lose) approximately $31536000 \times 10^{-6} = 31.536$ seconds per year. Therefore, internal clocks with an accuracy of 1500 PPM are not acceptable for keeping time in the long run. External crystals have a much better accuracy than internal clocks. The typical accuracy of external clocks is within 20 PPM, which makes external clocks a good source for **RTC** (Real Time Clock).

In a modern quartz watch, the most popular clock frequency is 32.768 KHz (2^{15} Hz) because an oscillator with this frequency has an excellent tradeoff between physical size and current drain. On the STM32L4 Nucleo board, a 32.768 KHz crystal is connected to the microprocessor as the **LSE** (Low Speed External) clock via **PC14** (`OSC32_IN`) and **PC15** (`OSC32_OUT`). The RTC is often driven using the LSE clock and is often powered by a separate battery such that RTC operation does not stop even when the main power is turned off. In addition, the RTC module is very energy-efficient; it consumes only 300 nA at 1.8V, including LSE clock power consumption.

RTC registers are in the backup domain. By default the backup domain is protected from write accesses. To unlock write protection on the RTC registers (except a few), two bytes `0xCA` and `0x53` must be sequentially written into the register `RTC_WPR`. To reactivate write protection, any byte (that isn't the right key to unlocking write protection) can be written to `RTC_WPR`.

The seconds, minutes, hours (12- or 24- hour format) are stored in `RTC_TR`. The day of the week, date, month, and year are stored in `RTC_DR`. These values are stored in **BCD** (binary coded decimal) format. In `RTC_TR` and `RTC_DR`, there are two group of bits that are used to represent one value. One group of bits store the tens digit (the name has suffix `xT` for "tens") and another group of bits store the units digit (the name has suffix `xU` for "units"). For example, if we wanted to store the value 53 in the minutes register, we would set `RTC_TR_MNT` = `0b101` (since the tens digit is 5) and `RTC_TR_MNU` = `0b0011` (since the units digit is 3).

5.2 Lab Exercise

In this part of the lab, you will configure the RTC and set up alarms to set or toggle the green LED depending on the seconds value of the current time. To keep things simple, we will focus on one task at a time.

5.2.1 Calculating RTC

You will be writing code in `RTC.c` and in the `main()` function of `main.c`.

The functions you will be implementing are:

- `RTC_Disable_Write_Protection()` and `RTC_Enable_Write_Protection()`
  These functions will ultimately control whether your RTC initialization will be done correctly since they affect whether the RTC registers can be written to.
- `RTC_Set_Calendar_Date()` and `RTC_Set_Time()`
  These setter functions allow you to configure the date and time that the RTC will be initialized to.

- `RTC_TIME_GetHour()`, `RTC_TIME_GetMinute()`, `RTC_TIME_GetSecond()`, `RTC_DATE_GetMonth()`, `RTC_DATE_GetDay()`, `RTC_DATE_GetYear()`, and `RTC_DATE_GetWeekDay()`

  These getter functions will extract the current time and date information from the RTC registers. They are used by the function `Get_RTC_Calendar()` to generate the strings containing that represent the current date/time.

To help you determine where to write/read from (in `RTC_TR` and `RTC_DR`) to get the time and date information, refer to following table, which is a subset of the register mapping for `RTC`. More detailed information about the RTC registers and bits can be found in *STM32L4x6 Reference Manual* (Section 34.6). In addition, the `RTC_POSITION_x` macros that are defined for you in `RTC.c` may help when you write your code.

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	TR	Reserved									PM	HT[1:0]		HU[3:0]					MNT[2:0]			MNU[3:0]					ST[2:0]			SU[3:0]			
	Value																																
0x04	DR	Reserved								YT[3:0]				YU[3:0]				WDU[2:0]			MT	MU[3:0]					DT[1:0]			DU[3:0]			
	Value																																

Within the `RTC_Init()` function, set the initial date and time of the RTC to the current date/time. Within the `while` loop, write code that calls the function `Get_RTC_Calendar()` to update strTime and strDate. These values should be monitoring in the watch window for correctness.

5.2.2 Alarms and Toggling LEDs

For toggling LEDs with RTC alarms, you will be writing code in `main.c`. You should import your LED initialization code from Lab 1.

Just as you would set an alarm for 7 AM on your phone, you can set alarms in RTC that will trigger at a specified date/time (day, hour, minute, second). The alarms can be configured to trigger on sub-seconds, but this lab will only focus on the hour, minute, and second.

The RTC module provides two programmable alarms: Alarm A and Alarm B. The `ALRxE` bits in `RTC_CR` control whether Alarm x is enabled. The registers `RTC_ALRMxR` are where you can program the date/time values that the alarm should trigger on. You can select what values should be considered when the module checks whether the current time matches the specified time by setting the mask bits `MSKx`.

- `MSK4` – Alarm x mask for date
- `MSK3` – Alarm x mask for hours
- `MSK2` – Alarm x mask for minutes
- `MSK1` – Alarm x mask for seconds

For each of the mask bits, `0` includes the value in the comparison and `1` treats the value as a don't-care in the comparison.

The RTC Alarm interrupt is connected to EXTI line 18, so you also need to configure EXTI to handle RTC alarm interrupts. When the RTC date/time matches the programmed values in the `RTC_ALRMxR` registers, the corresponding `ALRxF` bit is set to `1` in `RTC_ISR`. Because both Alarm A and Alarm B interrupts are connected to EXTI line 18, these bits will help you differentiate which alarm was triggered. See Table 1 for the different interrupt control bits.

Interrupt event	Event flag	Enable control bit	Exit from Sleep mode	Exit from Stop mode	Exit from Standby mode
Alarm A	ALRAF	ALRAIE	yes	yes(1)	yes(1)
Alarm B	ALRBF	ALRBIE	yes	yes(1)	yes(1)
RTC_TS input (timestamp)	TSF	TSIE	yes	yes(1)	yes(1)
RTC_TAMP1 input detection	TAMP1F	TAMPIE	yes	yes(1)	yes(1)
RTC_TAMP2 input detection	TAMP2F	TAMPIE	yes	yes(1)	yes(1)
RTC_TAMP3 input detection	TAMP3F	TAMPIE	yes	yes(1)	yes(1)
Wakeup timer interrupt	WUTF	WUTIE	yes	yes(1)	yes(1)

1. Wakeup from STOP and Standby modes is possible only when the RTC clock source is LSE or LSI.

Table 1: Interrupt Control Bits

To set up a timer alarm, you need to 1) set up the interrupt that occurs when an alarm is triggered and 2) program and enable the alarm by setting necessary bits in the RTC registers. The following steps outline the general procedure you should follow.

1. Implement the function `RTC_Alarm_Enable()`, which sets up the RTC alarm interrupt. For details about EXTI registers and bits, refer to Section 13.5 of *STM32L4x6 Reference Manual*.
   (a) Configure the interrupt to trigger on the rising edge in `EXTI_RTSRx`.
   (b) Set the interrupt mask in `EXTI_IMRx` and the event mask in `EXTI_EMRx`.
   (c) Clear the pending interrupt in `EXTI_PRx` by writing a `1` to the bit that corresponds to the target EXTI line.
   (d) Enable the interrupt in the NVIC and set it to have the highest priority. The name of the RTC alarm in the NVIC is `RTC_Alarm_IRQn`.
2. Implement the function `RTC_Set_Alarm()`, which programs the RTC alarm. For details about RTC registers and bits, refer to Section 34.6 of *STM32L4x6 Reference Manual*. A portion of the RTC register mapping is also provided below.
   (a) Before programming the alarms, disable both alarms in `RTC_CR`.
   (b) Remove RTC write protection so that we can write to the RTC registers.
   (c) Clear the alarm enable bit and the interrupt enable bit for both alarms. In addition, wait until access to both alarm registers is allowed. The hardware sets the write flag `ALRxWF` if alarm x can be modified.
   (d) For now, program Alarm A to set off an alarm when the seconds field of the RTC is 30 seconds. During the demo, the TA will ask for a different value.

(e) Program Alarm B to set off an alarm every second.

(f) Enable both Alarms A and B and their interrupts.

(g) Re-enable write protection for the RTC registers.

3. Implement the function `RTC_Alarm_IRQHandler()`, the interrupt handling function for the alarm. Remember to clear all necessary flags (i.e. the alarm event flag and the interrupt pending bit). Implement the following functions. During checkoff, TAs will test these 2 features separately. Make sure you know how to switch between these two functions.
   - Enable Alarm A and disable Alarm B. When Alarm A is triggered, toggle the green LED.
   - Enable Alarm B and disable Alarm A. When Alarm B is triggered, toggle the green LED.

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x08	CR	Reserved							ITSE	COE	OSEL[1:0]		POL	COSEL	BKP	SUB1H	ADD1H	TSIE	WUTIE	ALRBIE	ALRAIE	TSE	WUTE	ALRBE	ALRAE		FMT	BYPSHAD	REFCKON	TSEDGE	WUCKSEL[2:0]		
	Value																																
0x0C	ISR	Reserved														ITSF	RECALPF	TAMP3F	TAMP2F	TAMP1F	TSOVF	TSF	WUTF	ALRBF	ALRAF	INIT	INITF	RSF	INITS	SHPF	WUTWF	ALRBWF	ALRAWF
	Value																																
0x10	PRER	Reserved								PREDIV_A[6:0]									PREDIV_S[14:0]														
	Value																																
0x14	WUTR	Reserved																WUT[15:0]															
	Value																																
0x1C	ALRMAR	MSK4	WDSEL	DT[1:0]		DU[3:0]				MSK3	PM	HT[1:0]		HU[3:0]				MSK2	MNT[2:0]			MNU[3:0]				MSK1	ST[2:0]			SU[3:0]			
	Value																																
0x20	ALRMBR	MSK4	WDSEL	DT[1:0]		DU[3:0]				MSK3	PM	HT[1:0]		HU[3:0]				MSK2	MNT[2:0]			MNU[3:0]				MSK1	ST[2:0]			SU[3:0]			
	Value																																

6 Checkoff Requirements

1. Part A
   a) Rotate your stepper motor exactly 360 degrees clockwise or counter-clockwise using full-stepping.
   b) Rotate your stepper motor exactly 360 degrees clockwise or counter-clockwise using half-stepping.
2. Part B - Toggling the LED via interrupt
3. Part C
   a) Calendar set to checkoff time, show date and time string in debug mode
   b) Alarm that toggles LED every minute
   c) Alarm that toggles LED every second

7 References

[1] STM32L4x6 Advanced ARM-based 32-bit MCUs Reference Manual

[2] Yifeng Zhu, "Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C", ISBN: 0982692633