GEBZE TECHNICAL UNIVERSITY

ELECTRONIC ENGINEERING

ELM334

Microprocessors Laboratory

LAB 2 Experiment Report

1) 210102002037 – Beyzanur Cam
2) 210102002026 – Efe Bayrakçeken
3) 200102002027 – Erhan Gök

Problem 1. Write assembly code that will toggle the on-board LED at a rate of 1 second.

In order to turn on the on-board led, we have to first enable the RCC clock. Then set the GPIO mode to output. Then we can finally turn on the led.

To do all of these, first we need to find the memory addresses related to the RCC clock with relevant values. According to the memory map of the arm architecture, the memory addresses that are related to RCC are located between the AHB(advanced high-performance bus) addresses.

The exact addresses can be seen at table 6 of the reference manual. I/O port reset register has a 0x24 bit offset compared to the RCC base value. The bit 2 needs to be set to enable the GPIOC.

After the RCC configuration is done, GPIO can be set to output mode. The base value for the GPIOC again can be seen at Table 6 of the reference manual.

```
//////Enable RCC_IOPENR For C//////
      ldr r0,=RCC_IOPENR
      ldr r1, [r0]

      movs r2, #0x04 //0000 0100, For enabling RCC mask
      orrs r1, r1, r2
      str r1, [r0] //Enable RCC mask
////////////////////////////////////
```

Then the base address of the GPIOC can be set as variables. The addresses related to the GPIOC_BASE can be found at Table 6 of RM0444[1]. Then the address related to the mode of PC6 GPIO can be set as per the section 7.4.1 of the reference manual. Lastly GPIO port output data register (GPIOC_ODR) can be declared.

```
///////////GPIOx_MODER/////////
      ldr r0,=GPIOC_MODER
      ldr r1,[r0]

      ldr r2,=0x3000
      bics r1, r1, r2
      ldr r2, =0x1000
      orrs r1,r1,r2

      str r1, [r0]
//////////////////////////////
```

We assumed that the processor runs at 16mhz, so a FREQ value was declared. Then the count of cycles needed for a loop was declared. This will be explained later in the report.

```
.equ FREQ, (16000000)
.equ CYCLE_COUNT, (FREQ/3)
```

The code provided is the delay function. The function subtracts 1 to a known value. When the value of r7 becomes 0, the zero flag of the alu is raised and the branch instruction returns to the value that is stored in the link register. The subs instruction just takes 1 clock cycle but the bx instruction

[1] STM32G0x1 advanced Arm®-based 32-bit MCUs - Reference manual

requires 2-3 clock cycles to run. Therefore the clock frequency was divided by 3 to get the appropriate delay required for 1 second loops.

```
delay_func:
	subs r7, #1
	bne delay_func
	ldr r7, =CYCLE_COUNT
	bx lr
```

In order for the delay function to work r7 must be set to CYCLE_COUNT, otherwise the register starts from -1 and in the delay function it starts to subtracts from -1. So the necessary condition(Z=1) for the delay function to return to the loop is never met.

```
//////////GPIOC_ODR/////////
	ldr r0,=GPIOC_ODR
	ldr r1, [r0]

	ldr r2,=0x40

	ldr r7, =CYCLE_COUNT //Set R7 to CYCLE_COUNT
////////////////////////////
```

For the final part we have to create a loop to call function when needed. When led lights up “bl” (branch & link) saves the program counter to the link register and makes code to go to the delay_func. After “delay_func" is executed, bx recovers the program counter from the link register and brings code back to the loop. Then the bits are cleared and the led turns off. Then the same execution goes on and on. And because of the aim of delay_func our eyes can catch the led’s status.

```
loop:
	orrs r1, r1, r2
	str r1, [r0] //Turn on LED

	bl delay_func

	bics r1, r1, r2
	str r1, [r0] //Turn off LED

	bl delay_func
	b loop
```

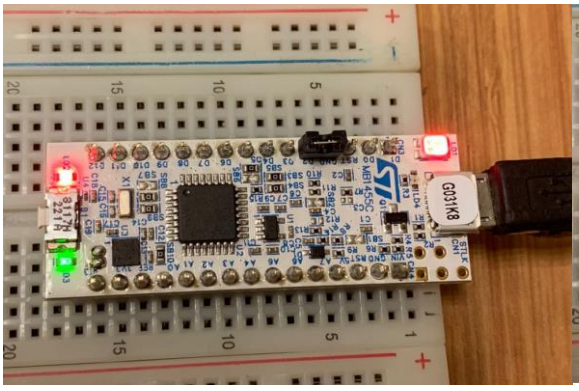


Figure 1 On-board led on

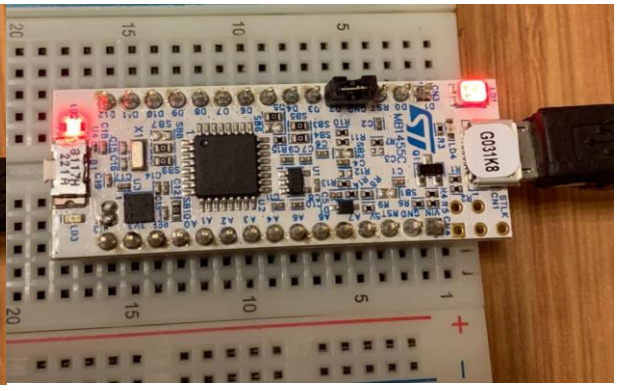


Figure 2 On-board led off

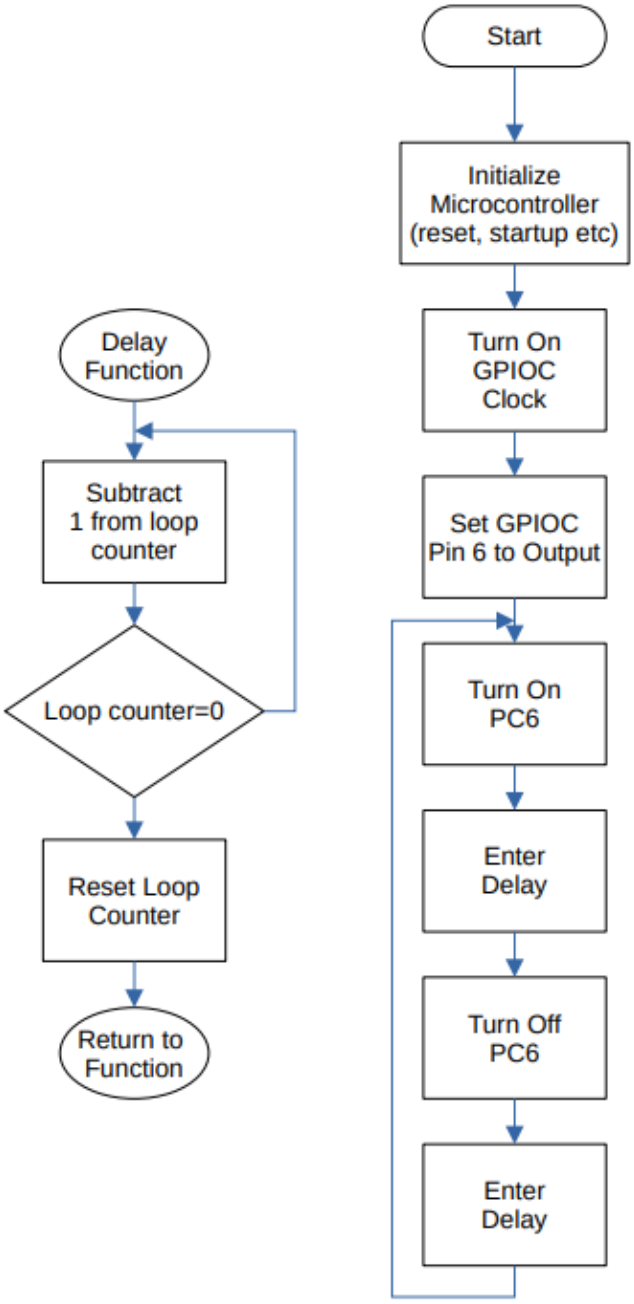


Figure 3 Problem1 Flowchart

There is no need for schematic for this problem. The board connected to power is enough.

Problem 2. Connect a button to the board, and turn on the on board LED when the button is pressed. When the button is released, the LED should turn off.

For this question we connect a button to pin PA9 and for the led on-board nothing is changed. For button we enable clock signal for A and add it's address. The difference is for this problem we will be taking a information from button. Because of that when coding the GPIOA_MODER we set the relevant bits to 00 (input mode).

```
//////////GPIOA_MODER////////
      ldr r0,=GPIOA_MODER
      ldr r1,[r0]

      ldr r2,=0xC0000
      bics r1, r1, r2
      str r1, [r0]
////////////////////////////
```

7.4.5 GPIO port input data register (GPIOx_IDR) (x = A to F)

Address offset: 0x10

Reset value: 0x0000 XXXX

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ID15	ID14	ID13	ID12	ID11	ID10	ID9	ID8	ID7	ID6	ID5	ID4	ID3	ID2	ID1	ID0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **ID[15:0]:** Port x input data I/O pin y (y = 15 to 0)
These bits are read-only. They contain the input value of the corresponding I/O port.

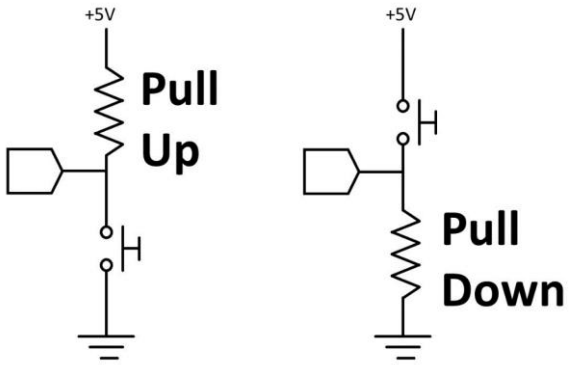
Because we are reading a data from button it becomes an input data register. As we can see from the table above we set the 9. Pin to 1 because our button was in PA9.

```
//////////GPIOC_ODR/////////
      ldr r0,=GPIOC_ODR
      ldr r1, [r0]

      ldr r5, =GPIOA_IDR

      ldr r4, =0x200 // button read mask
      ldr r2, =0x40 // set led mask
////////////////////////////
```

In this loop we read buttons state each time it changes. And two functions created for two status of led (on and off). In the first part buttons state and its read mask is compared if they are not same it goes to turn_on function otherwise it goes to turn_off function.

```
loop:
	ldr r6, [r5]
	ands r6, r4
	bne turn_on
	b turn_off
	turn_on:
		orrs r1, r1, r2
		b end
	turn_off:
		bics r1, r1, r2
	end:
		str r1, [r0]
```

In order the button to work, the button is connected to the PA9 with a pull-up resistor whose value is 10k Ω. Pull-up resistors are resistors which are used to ensure that a wire is pulled to a high logical level in the absence of an input signal. When the button is not pressed, the state of the end of the resistor high so the state of the pin is high. When the button is pressed it shorts the resistor and ground, so the state of the pin becomes low. Pull-up and pull-down resistor connections are shown in the picture below.



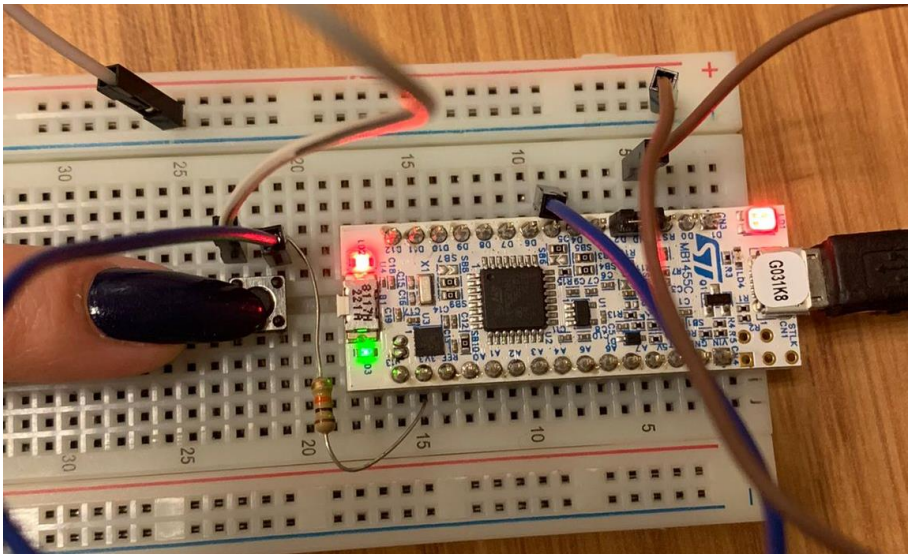


Figure 4 Problem 2 on board

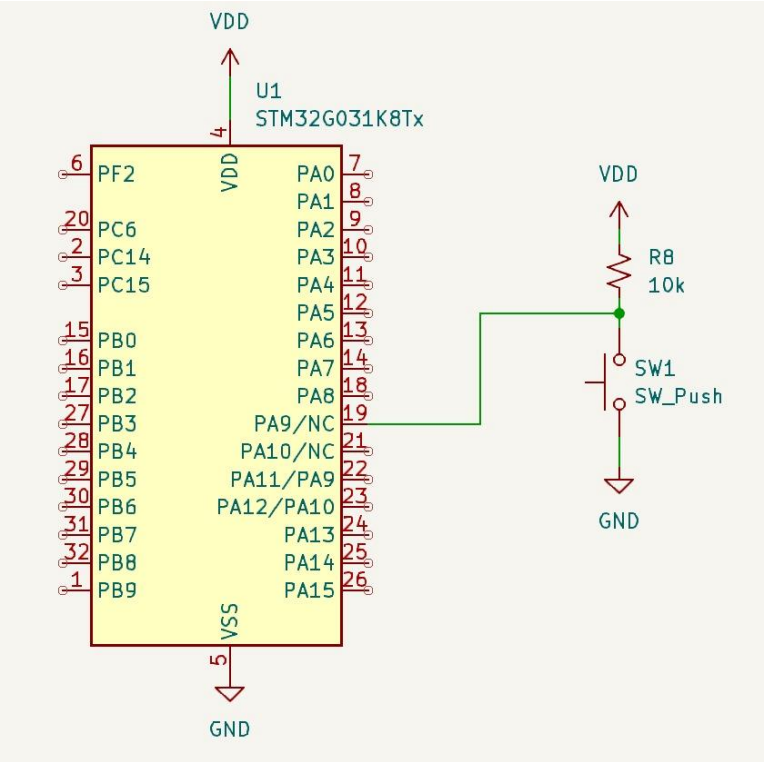


Figure 5 Schematic of problem 2

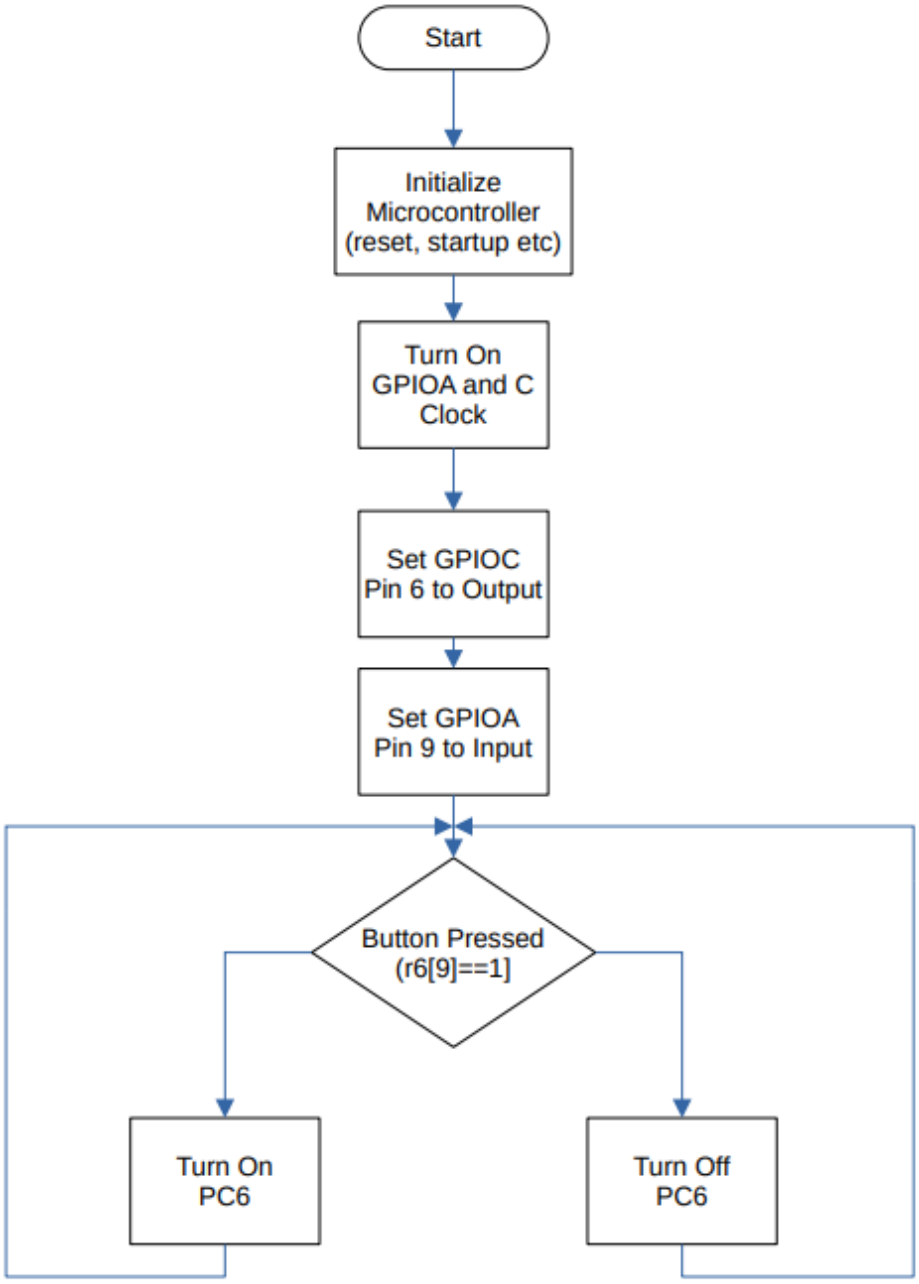


Figure 6 Flowchart of problem 2

Problem 3. Connect 8 external LEDs to the board, and toggle all the LEDs at the same time at a rate of 1 second.

So this problem is basically the same with the problem 1 except in this problem, clock of the B line is activated and all 8 leds are being toggled at once. As we can see from the table below, 8 led are connected sequentially to the B line of the MCU for ease of coding and other lines are not used. At he end all pins are opened and closed at the same time.

```
//////Enable RCC_IOPENR For B//////
      ldr r0,=RCC_IOPENR
      ldr r1, [r0]

      movs r2, #0x02 //0000 0100, For enabling RCC mask
      orrs r1, r1, r2
      str r1, [r0] //Enable RCC mask
////////////////////////////////////

///////////GPIOx_MODER////////
      ldr r0,=GPIOB_MODER
      ldr r1,[r0]

      ldr r2,=0xFFFF
      bics r1, r1, r2
      ldr r2, =0x5555
      orrs r1,r1,r2

      str r1, [r0]
//////////////////////////////

///////////GPIOB_ODR/////////
      ldr r0,=GPIOB_ODR
      ldr r1, [r0]

      ldr r2,=0xFF

      ldr r7, =CYCLE_COUNT //Set R7 to CYCLE_COUNT
//////////////////////////////
```

PB6	1	D1	VIN	1	VIN
PB7	2	D0	GND	2	GND
NRST	3	NRST	NRST	3	NRST
GND	4	GND	+5V	4	+5V
PA15	5	D2	A7	5	PA7
PB1	6	D3	A6	6	PA6
PA10	7	D4	A5	7	PA11
PA9	8	D5	A4	8	PA12
PB0	9	D6	A3	9	PA5
PB2	10	D7	A2	10	PA4
PB8	11	D8	A1	11	PA1
PA8	12	D9	A0	12	PA0
PB9	13	D10	AREF	13	AREF
PB5	14	D11	+3V3	14	+3v3
PB4	15	D12	D13	15	PB3
	CN3			CN4	

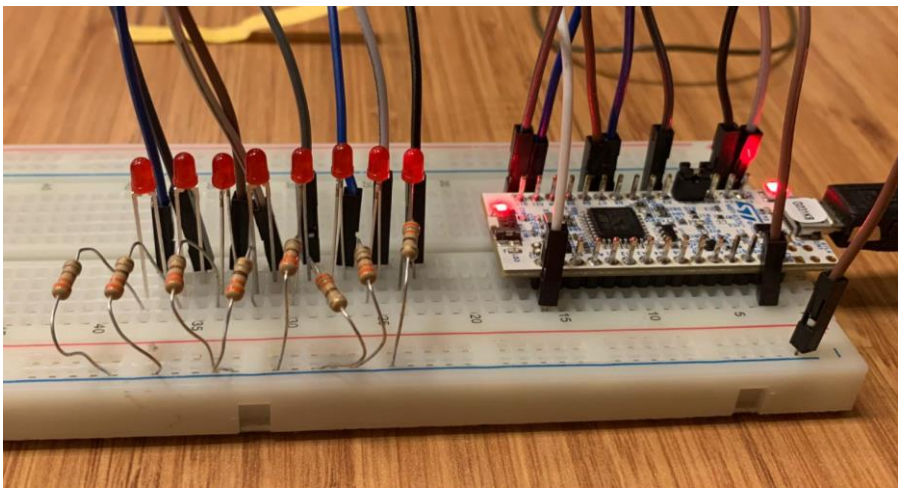


Figure 7 Problem3 leds off

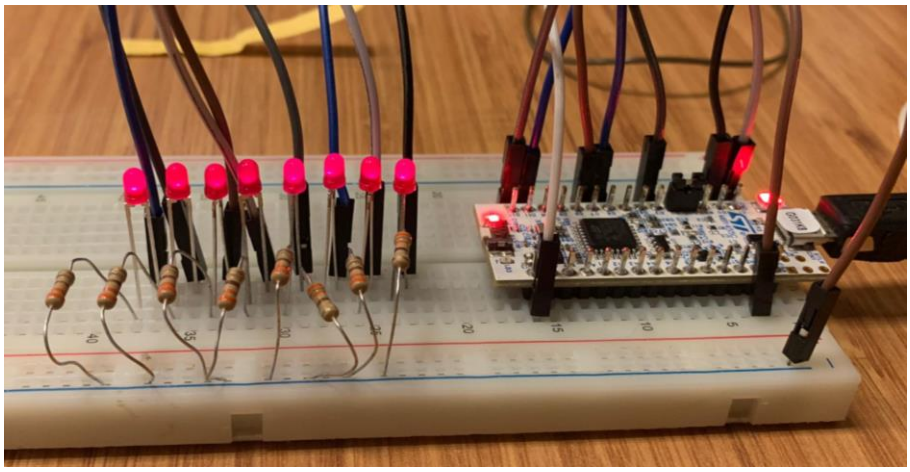


Figure 8 Problem 3 leds on

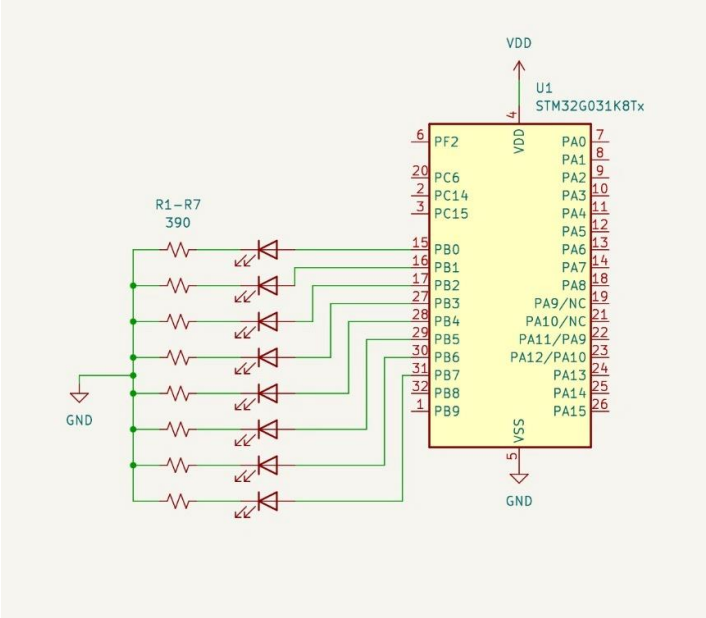


Figure 9 Problem 3 schematic

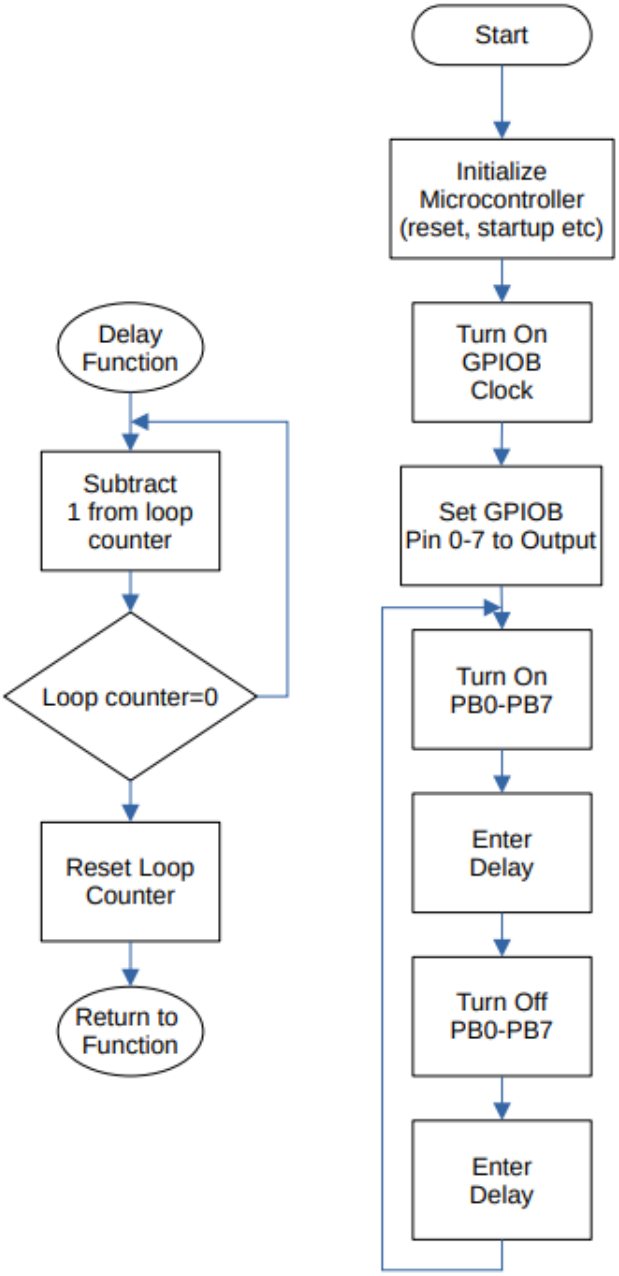
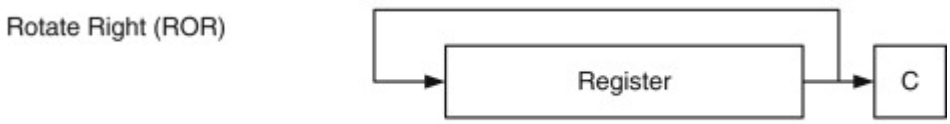


Figure 10 Problem 3 Flowchart

Problem 4. Connect 8 LEDs and 1 button to the board, and implement a shift pattern.

To implement this pattern, the usual prosidure of turning on the GPIO clocks were performed. For this problem, GPIOA was needed for the button operation and GPIOB was needed for all the indidvual leds. This is almost the same as what was done In problem 2, but the leds now are connected to GPIOB in order instead of a single pin connected to PC6. Since a toggle algorithm was requested, that was also implemented with a branching instruction while the button is pressed.

The actual pattern was done with the ROR (rotate right) psudo instruction. This instruction takes the shifted values of the register (LSB's) and writes to the MSB of the register. With a clever mask, this instruction makes it possible to create a cascading pattern.



The mask mentioned will be "11100000" repeated 4 times. When this mask is applied to our GPIOA_ODR, the leds will cycle with the intended pattern.

t0	...11100000
t1	...01110000
t2	...00111000
t3	...00011100
t3	...00001110
t4	...00000111
t5	...10000011
t6	...11000001
t7	...11100000

In the table above, the pattern was shown as binary values. As can clearly be seen the mask constantly rotates to set the appopriate values.

The toggling the direction of the cascading effect was requested with the question as well. This can be accomplished with rotating the same mask left.

A problem that exists with the ARM's thumb instruction set is that there is no rotate left instuction. This can easily be mitigated by the rotate right instruction. This is because Rotatating right by 31 digits are equal to rotation left by 1 digits. With this clever trick, the reversing of the pattern can easily be accomplished.

A register was used to hold the current button state. When the button is pressed, the button state register will be not'ed with itself to invert all the bits from 0x00000000 to 0xFFFFFFFF, and vice versa when the button is pressed again. When the toggle is at 0, the zero flag is set. This sets the rotate ammount to either 1 or 31 depending on the state.

While the button is pressed, the microprocessor is stopped from doing further operations. This is needed when not using interrupts since the microcontroller will read the button value each loop and constantly toggle the button state which makes the led jitter and ultimately makes the toggling unreliable.

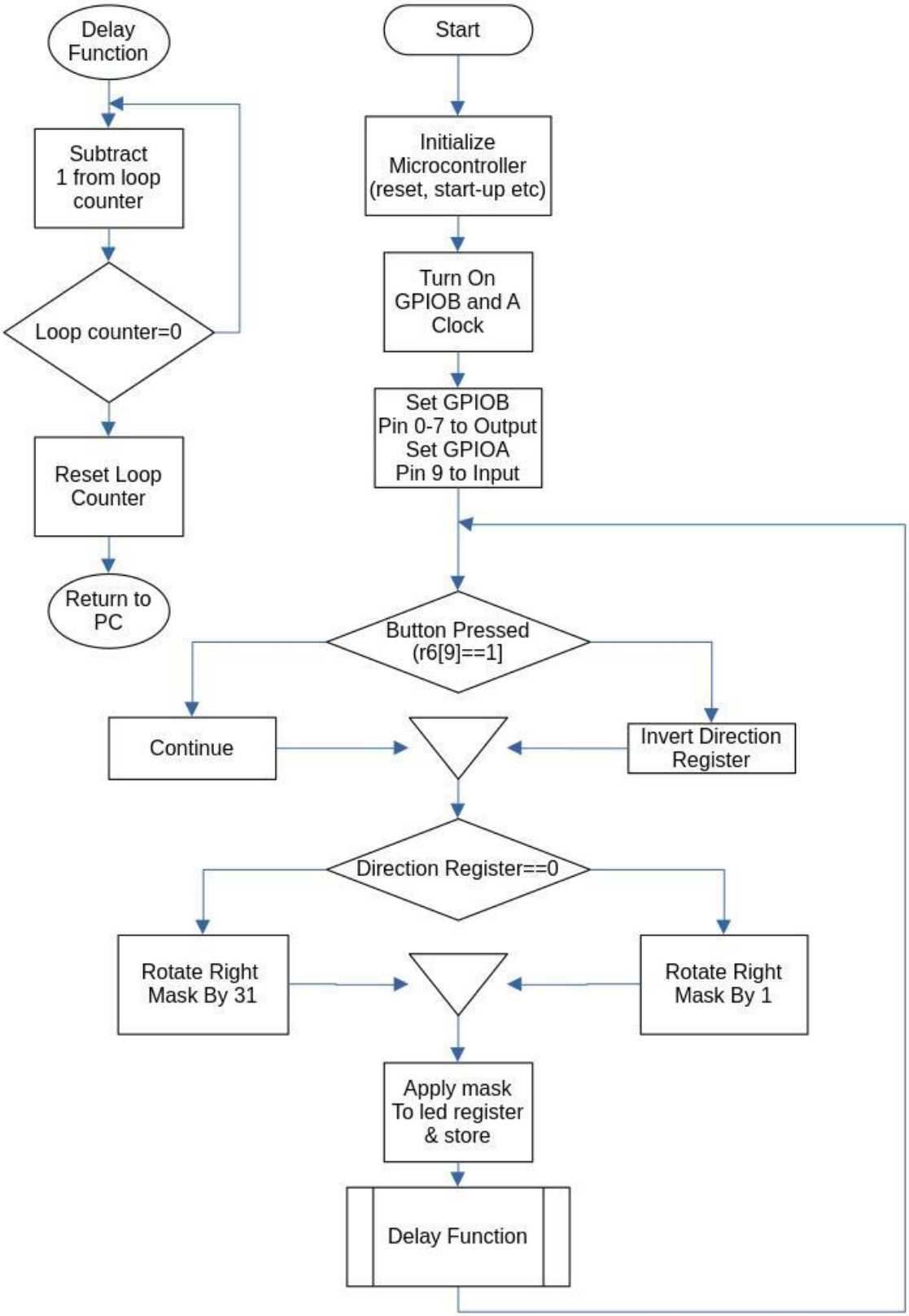


Figure 11 Problem 4 Flowchart

Up untill line 140 (the line with the *//////////GPIOX_ODR/////////* lines), the code is the same as the code in question 2. The delay function from problem 1 was added to the code as well. This time, clock frequency was divided by 30 (instead of 3) to get 100ms delays.

```
//////////GPIOA_ODR/////////
ldr r0,=GPIOB_ODR
ldr r1, [r0]

ldr r5, =GPIOA_IDR
ldr r4, =0x200 // button read mask

ldr r2, =0xE0E0E0E0 // LED Rolling mask

ldr r7, =CYCLE_COUNT //Set R7 to CYCLE_COUNT

movs r6, #31 // Rotate Direction
ldr r3, =0xFFFFFFFF
```

Here, the masks said above were assigned to r2, the rotate direction to r6. The button state is at r3.

```
loop:
//read button
ldr r6, [r5]
ands r6, r4
beq pressed
b not_pressed
pressed:

wait_button_release:
ldr r6, [r5] //read button register
ands r6, r4 //Apply button read mask
beq wait_button_release
mvns r3, r3 //invert the rotate direction
not_pressed:
```

The button logic is provided above. The button is again read and checked for its current state. If it was pressed, it jumps to “pressed:” label of the code. This part of the code is basically a while loop that loops if the button is still pressed. After the button is released, the rotate direction is inverted and the program continues as usual.

```
cmp r3, #0 //If r3 is zero, rotate right, else rotate left.
bne rotate_left
b rotate_right
rotate_left:
movs r6, #31
b end
rotate_right:
movs r6, #1
end:
//mask operations
rors r2, r2, r6
orrs r1, r1, r2
ands r1, r1, r2
```

```
str r1, [r0] // Store the updated ODR values

bl delay_func

b loop
```

In this part, the rotate toggle is checked. If the toggle is 1, it jumps to rotate left, else it jumps to rotate right. At the end the mask is finally rotated and it is applied, then written to the memory.

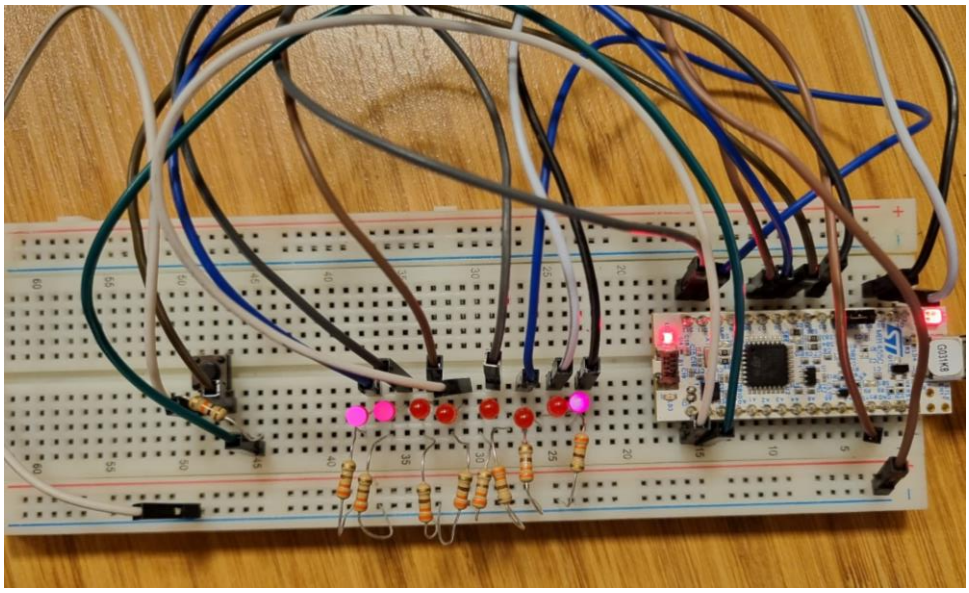


Figure 12 Problem 4 on board

The leds cycle as expected.The button also works properly.

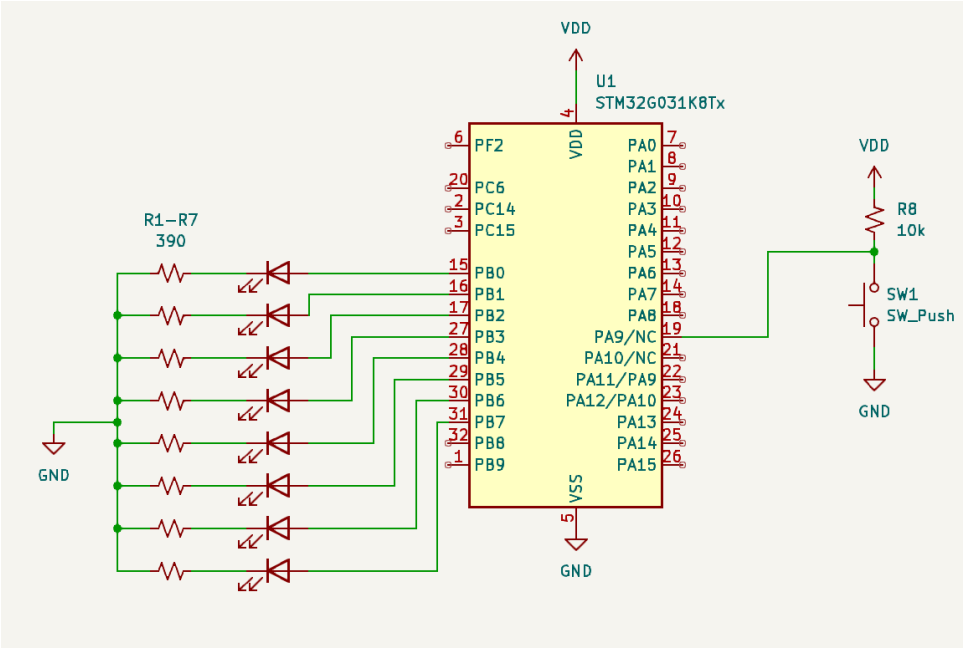


Figure 13 Problem 4 schematic

The schematic is simple, just a pull-up resistor was connected to relevant pins with a button, and the leds were connected in order to PB0-PB7.

Since the microcontroller enters a delay function right after the button press, a rc filter was not needed to debounce the button

Problem 5. Connect 8 LEDs to the board and implement a “Knight Rider (Kara Şimşek)”.

This problem is almost the same as problem 4. The only difference is that the toggling is done with a with a counter instead of a button. If the counter has reached the desired value, the rotate direction reverses. It again uses the rotate instruction.

Instead of the button logic, the controller subtracts 1 from the bounce_counter resumes operation untill the bounce counter is 0, then resets the bounce counter to an appopirate value (in this case 5). Lastly the switch is toggled to its reverse value. Thus, the loop is complete! After function cycles, the loop is reset but in the reverse direction.

Code is almost identical to problem 4 as said before. The only changes are that the loop counter is added, and input registers removed.

```
//////////GPIOX_ODR//////////
ldr r0,=GPIOB_ODR
ldr r1, [r0]

ldr r2, =0x000000E0 //Led Mask

ldr r7, =CYCLE_COUNT //Set R7 to CYCLE_COUNT
movs r6, #31 //Rotate Direction
ldr r3, =0x06 //Ammount of loops before toggle+1
movs r5, #0x0 //State storage
```

The loop counter starts from 6 because the loop subtracts 1 before the actual loop which shifts the registers too early.

To save on a compare operation, the loop counter is decremented and checked in the branch operation to be less than 0.

```
loop:
bounce_func:
subs r3, r3, #1
bgt bounce_end
reset_bounce:
movs r3, #5
mvns r5, r5
bounce_end:
```

Then the loop continues as usual.

```
movs r5, r5
bne rotate_left
b rotate_right
rotate_left:
movs r6, #31
b end
rotate_right:
movs r6, #1
end:
//mask operations
rors r2, r2, r6
orrs r1, r1, r2
ands r1, r1, r2
str r1, [r0]

bl delay_func

b loop
```

This part is identical to problem 4 it applies the mask to the GPIO_ODR and saves.

In the image below the button is redundant. The exact same circuit was used as problem 4 and the button was left there as it made it easier to switch between problems.

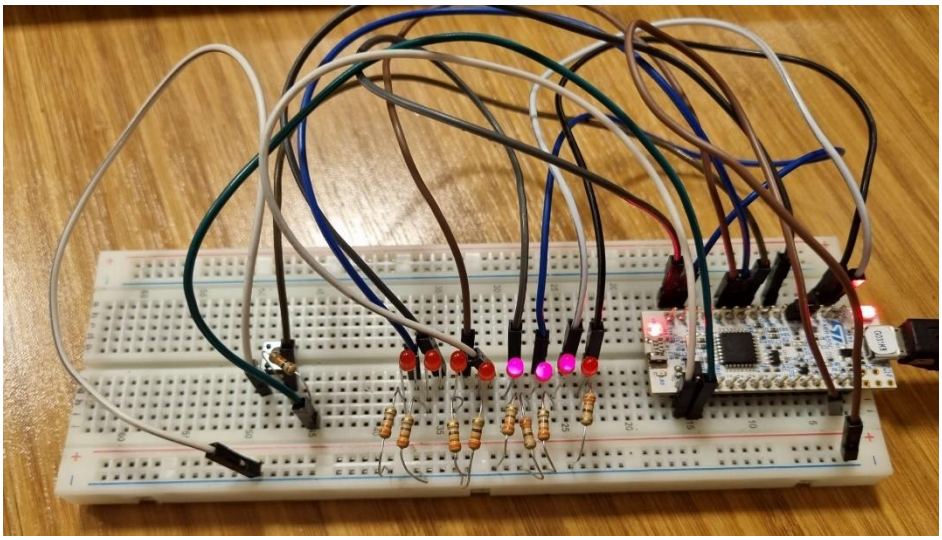


Figure 14 Problem 5 on board

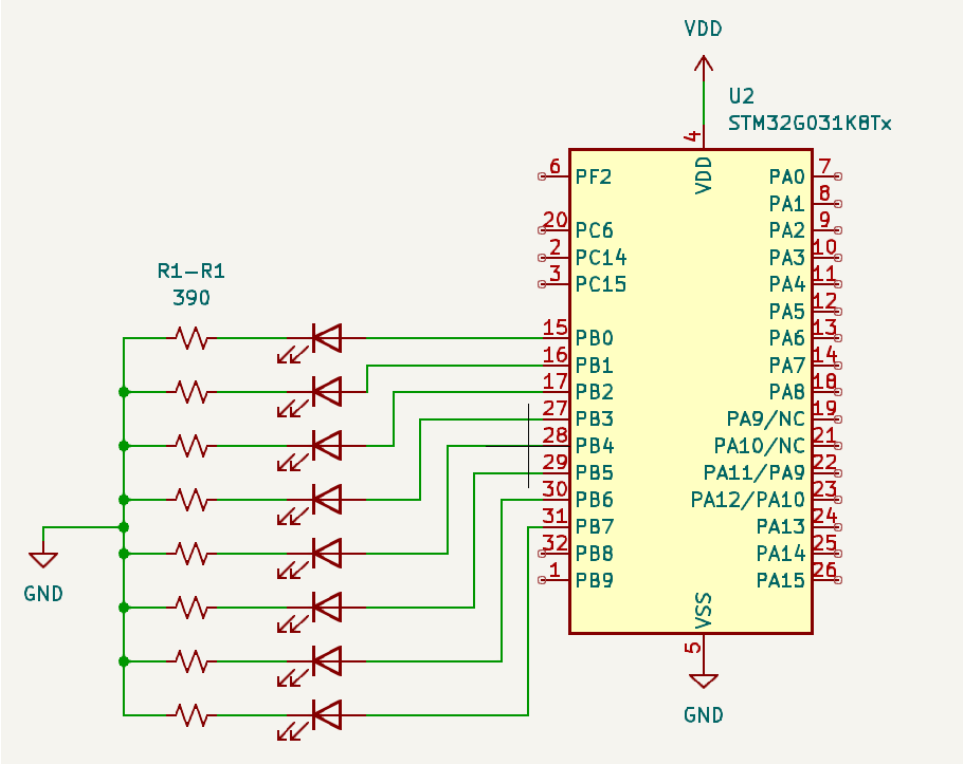


Figure 15 Problem 5 schematic

Schematic is the same as Problem 4 but with the button removed

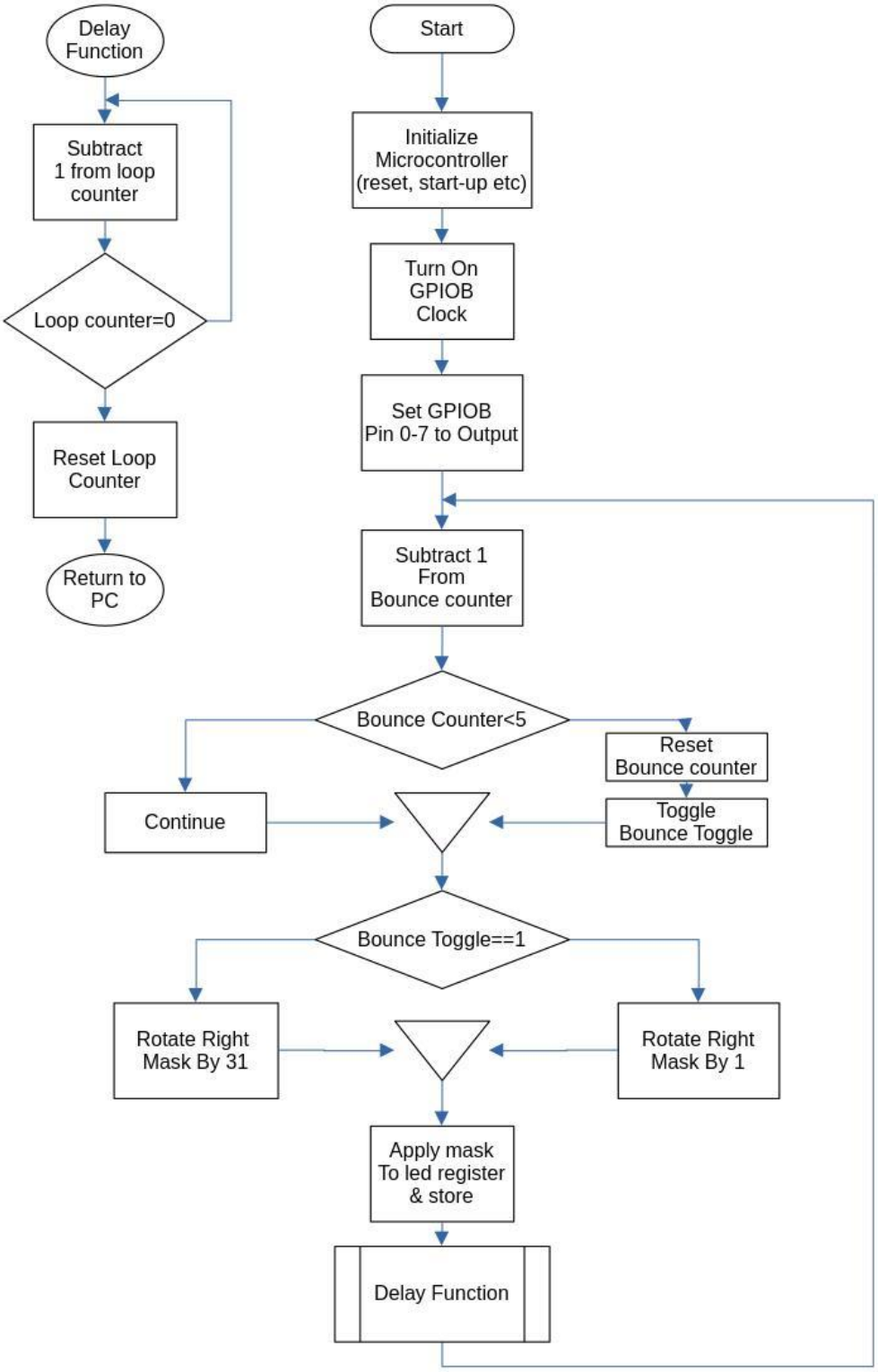


Figure 16 Problem 5 Flowchart

References:

https://developer.arm.com/documentation/ddi0337/e/BABBCJII

https://github.com/fcayci/stm32g0

STM32G0x1 advanced Arm®-based 32-bit MCUs - Reference manual

STM32G0 Nucleo-32 board - User manual