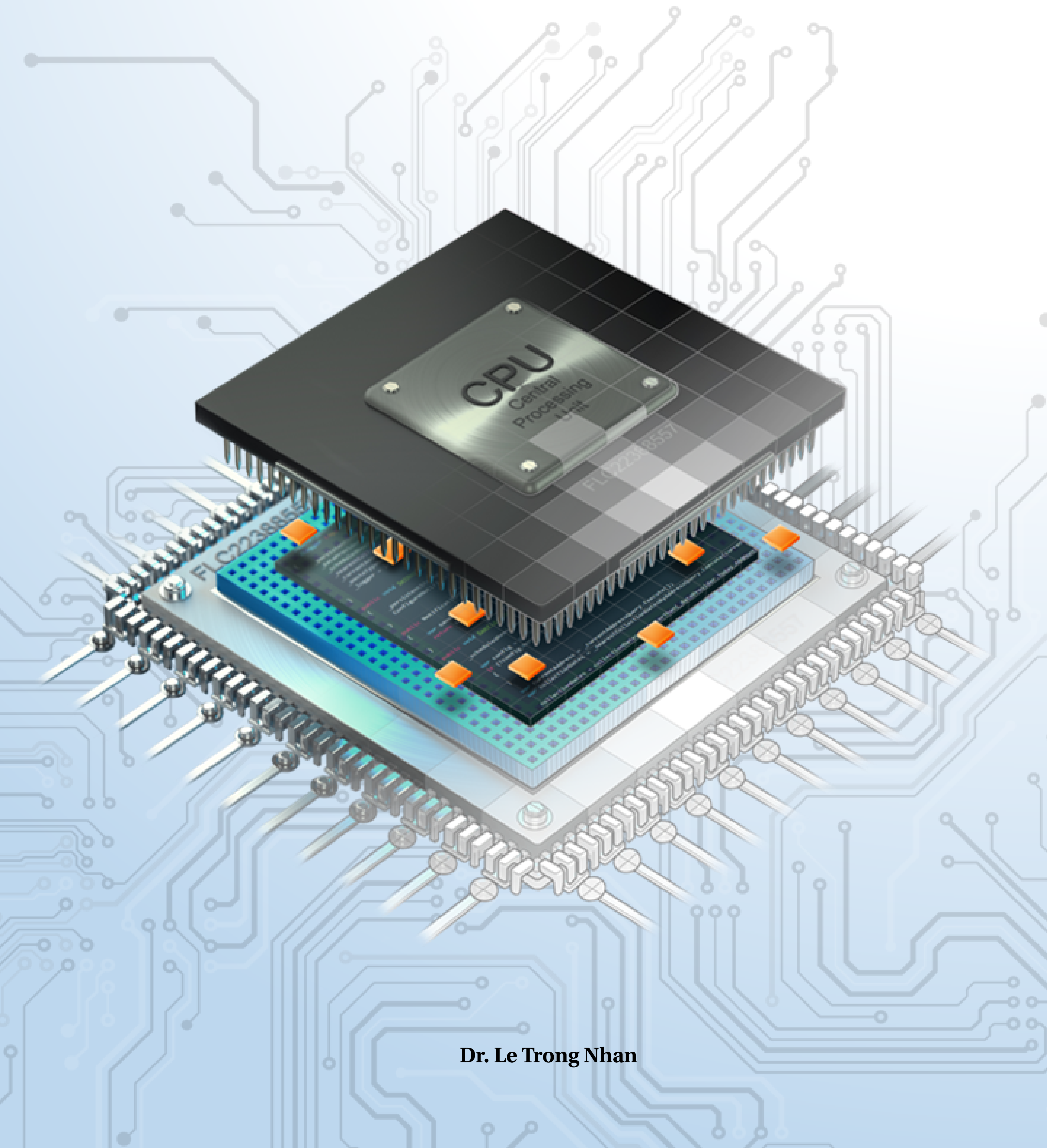
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
COMPUTER ENGINEERING

Microcontroller



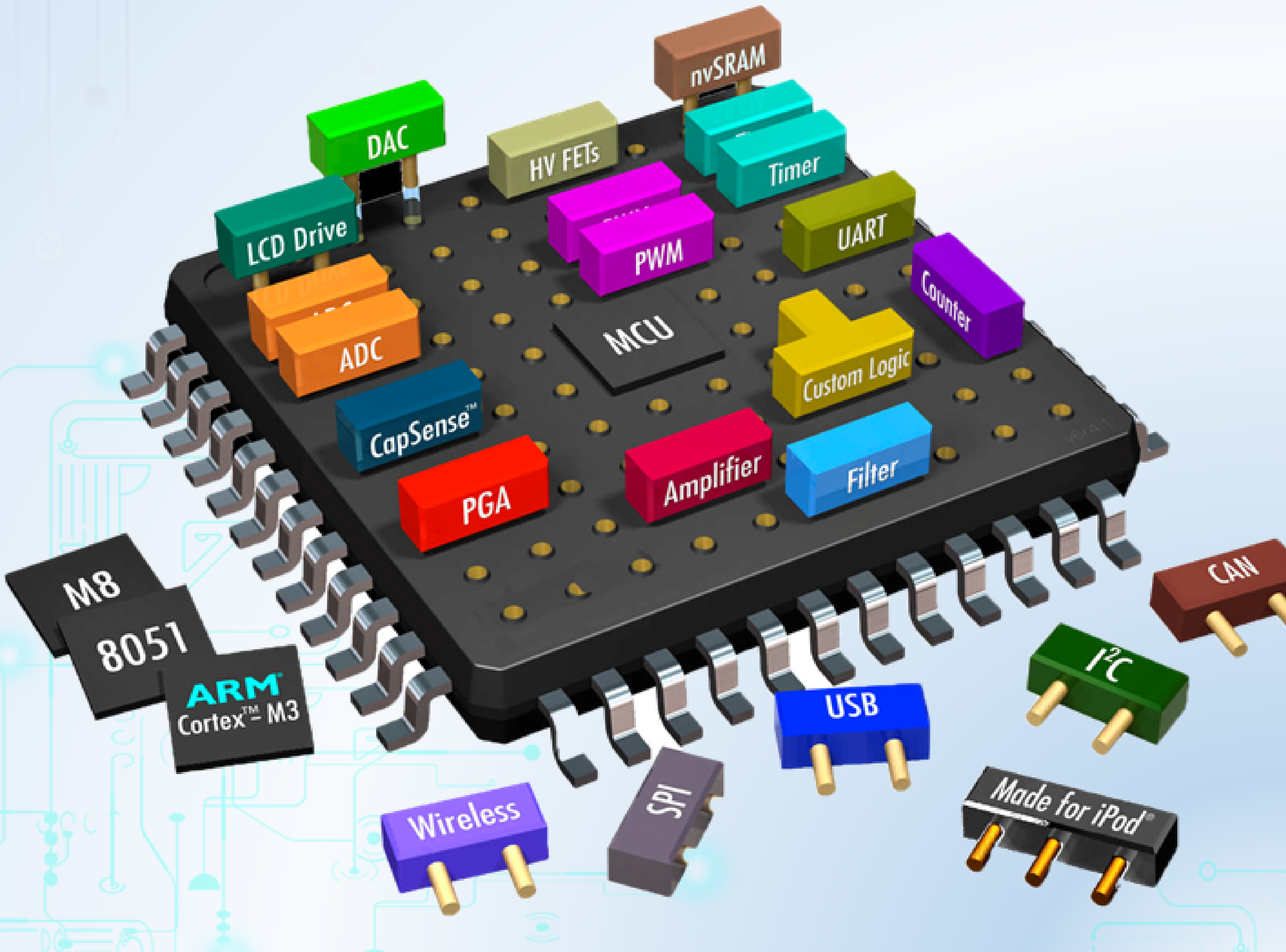
Dr. Le Trong Nhan

- **Name**: HOANG VAN PHI
- **MSSV**: 2252608
- **Class**: CO3010_CC01
- **Teacher**: NGUYEN THIEN AN
- **Lab**: 1
- **Link GitHub**: https://github.com/PhiuTheWind/VXL$_{LAB2.git}$

Mục lục

CHƯƠNG 1

LED Animations

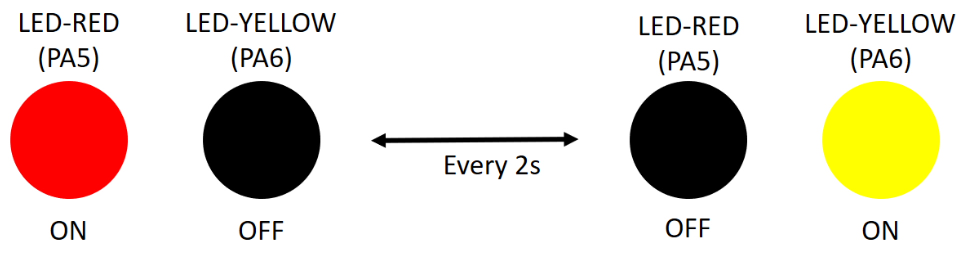


1 Exercise and Report

1.1 Exercise 1

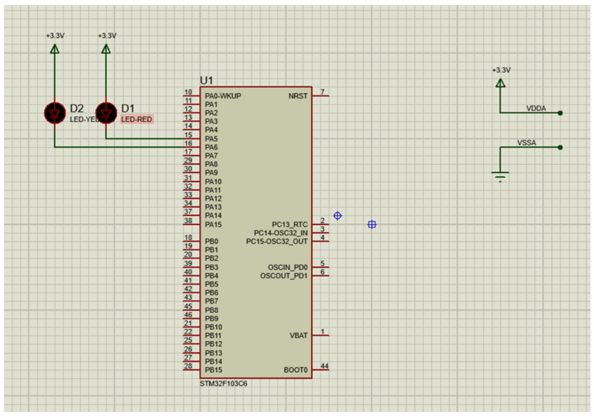
From the simulation on Proteus, one more LED is connected to pin **PA6** of the STM32 (negative pin of the LED is connected to PA6). The component suggested in this exercise is **LED-YELLOW**, which can be found from the device list.

In this exercise, the status of two LEDs are switched every 2 seconds, as demonstrated in the figure bellow.



Hình 1.1: *State transitions for 2 LEDs*

Report 1: Schematic from Proteus.



Hình 1.2: Schematic from Proteus

Report 2: Source code for the Exercise 1.

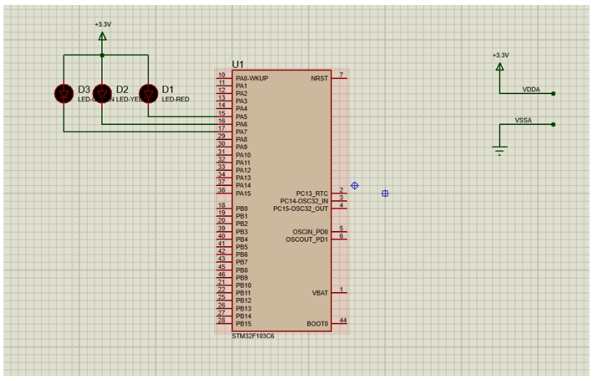
```
int counter=0;
  while (1)
  {
    /* USER CODE END WHILE */
     if(counter <2){
    HAL_GPIO_WritePin(LED_RED_GPIO_Port, LED_RED_Pin,
   GPIO_PIN_RESET);
    HAL_GPIO_WritePin(LED_YELLOW_GPIO_Port, LED_YELLOW_Pin,
   GPIO_PIN_SET);
          counter++;
  }else {

    HAL_GPIO_WritePin(LED_RED_GPIO_Port, LED_RED_Pin,
   GPIO_PIN_SET);
          HAL_GPIO_WritePin(LED_YELLOW_GPIO_Port,
   LED_YELLOW_Pin,GPIO_PIN_RESET);
              counter++;
        if(counter ==4) counter =0; }
    HAL_Delay(1000);
    /* USER CODE BEGIN 3 */
  }
```

Program 1.1: Source code

1.2 Exercise 2

Extend the first exercise to simulate the behavior of a traffic light. A third LED, named **LED-GREEN** is added to the system, which is connected to **PA7**. A cycle in this traffic light is 5 seconds for the RED, 2 seconds for the YELLOW and 3 seconds for the GREEN. The LED-GREEN is also controlled by its negative pin.

Report 1: Present the schematic.



Hình 1.3: Schematic from Proteus

Report 2: Present the source code in while.

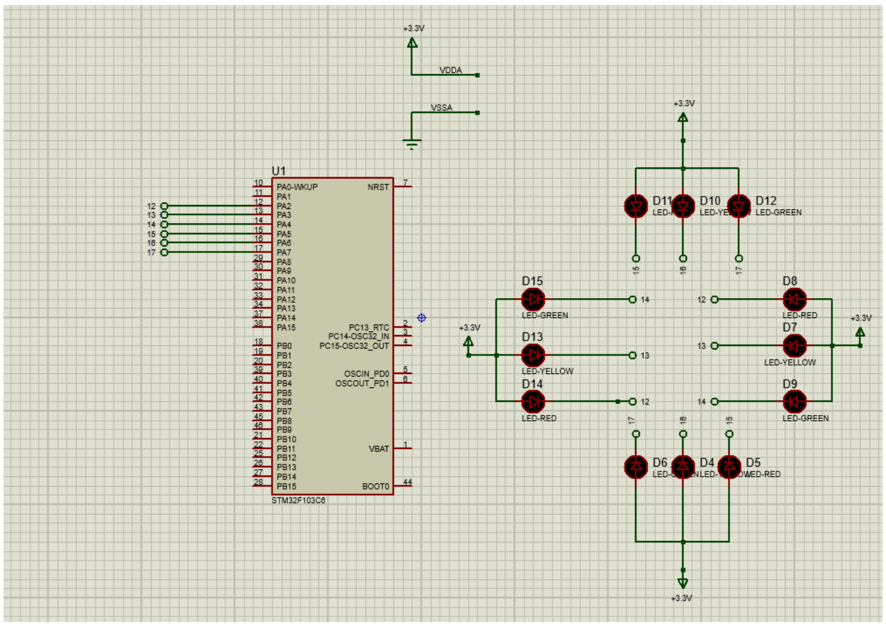
```
#define RED 0          // Define the color of the traffic
   lights
#define YELLOW 1
#define GREEN 2
int counter=0;

int led_status = 0;

while (1)
{
```

```
/* USER CODE END WHILE */
switch (led_status) {
        case RED:
    HAL_GPIO_WritePin(LED_RED_GPIO_Port, LED_RED_Pin,
  GPIO_PIN_RESET);
    HAL_GPIO_WritePin(LED_YELLOW_GPIO_Port, LED_YELLOW_Pin,
  GPIO_PIN_SET);
    HAL_GPIO_WritePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin,
  GPIO_PIN_SET);
        counter++;
        if(counter==5) {
        led_status = GREEN;
        counter =0;
        }
        break;
        case YELLOW:
    HAL_GPIO_WritePin(LED_RED_GPIO_Port, LED_RED_Pin,
  GPIO_PIN_SET);
    HAL_GPIO_WritePin(LED_YELLOW_GPIO_Port,LED_YELLOW_Pin,
  GPIO_PIN_RESET);
    HAL_GPIO_WritePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin,
  GPIO_PIN_SET);
        counter++;
        if(counter ==2){led_status = RED;
        counter =0;
        }
        break;
        case GREEN:
    HAL_GPIO_WritePin(LED_RED_GPIO_Port, LED_RED_Pin,
  GPIO_PIN_SET);
    HAL_GPIO_WritePin(LED_YELLOW_GPIO_Port, LED_YELLOW_Pin,
  GPIO_PIN_SET);
    HAL_WritePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin,
  GPIO_PIN_RESET);
        counter++;
        if(counter ==3){led_status = YELLOW;
        counter =0;
        }
        break;
        default:
        break;
        }

HAL_Delay(1000);
}
```

Program 1.2: Source code

1.3 Exercise 3

Extend to the 4-way traffic light. Arrange 12 LEDs in a nice shape to simulate the behaviors of a traffic light. A reference design can be found in the figure bellow.



Hình 1.4: Schematic from Proteus

```
//CODE FOR EX3 :
#define RED
#define GREEN
#define YELLOW


int counter=0;

  int led_status = RED;

  while (1)
  {
  /* USER CODE END WHILE */
  switch (led_status) {
  case RED:
  HAL_GPIO_WritePin(LED_RED_GPIO_Port, LED_RED_Pin,
   GPIO_PIN_RESET);
  HAL_GPIO_WritePin(LED_YELLOW_GPIO_Port, LED_YELLOW_Pin,
   GPIO_PIN_SET);
  HAL_GPIO_WritePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin,
   GPIO_PIN_SET);

  HAL_GPIO_WritePin(LED_RED_2_GPIO_Port, LED_RED_2_Pin,
   GPIO_PIN_SET);
  HAL_GPIO_WritePin(LED_YELLOW_2_GPIO_Port,
   LED_YELLOW_2_Pin,GPIO_PIN_SET);
```

```
HAL_GPIO_WritePin(LED_GREEN_2_GPIO_Port, LED_GREEN_2_Pin,
 GPIO_PIN_RESET);
counter++;
if(counter>3){
  HAL_GPIO_WritePin(LED_RED_2_GPIO_Port, LED_RED_2_Pin,
 GPIO_PIN_SET);
  HAL_GPIO_WritePin(LED_YELLOW_2_GPIO_Port,
 LED_YELLOW_2_Pin,GPIO_PIN_RESET);
  HAL_GPIO_WritePin(LED_GREEN_2_GPIO_Port,
 LED_GREEN_2_Pin,GPIO_PIN_SET);

}
if(counter==5) {
led_status = GREEN;
counter =0;
}
break;
case YELLOW:
      HAL_GPIO_WritePin(LED_RED_GPIO_Port, LED_RED_Pin,
 GPIO_PIN_SET);
      HAL_GPIO_WritePin(LED_YELLOW_GPIO_Port,
 LED_YELLOW_Pin,GPIO_PIN_RESET);
      HAL_GPIO_WritePin(LED_GREEN_GPIO_Port,
 LED_GREEN_Pin,GPIO_PIN_SET);

    HAL_GPIO_WritePin(LED_RED_2_GPIO_Port, LED_RED_2_Pin,
 GPIO_PIN_RESET);
    HAL_GPIO_WritePin(LED_YELLOW_2_GPIO_Port,
 LED_YELLOW_2_Pin,GPIO_PIN_SET);
    HAL_GPIO_WritePin(LED_GREEN_2_GPIO_Port,
 LED_GREEN_2_Pin,GPIO_PIN_SET);


counter++;
if(counter ==2){led_status = RED;

counter =0;
}
break;
case GREEN:
   HAL_GPIO_WritePin(LED_RED_GPIO_Port, LED_RED_Pin,
 GPIO_PIN_SET);
   HAL_GPIO_WritePin(LED_YELLOW_GPIO_Port, LED_YELLOW_Pin
 ,GPIO_PIN_SET);
   HAL_GPIO_WritePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin,
 GPIO_PIN_RESET);

   HAL_GPIO_WritePin(LED_RED_2_GPIO_Port, LED_RED_2_Pin,
 GPIO_PIN_RESET);
```

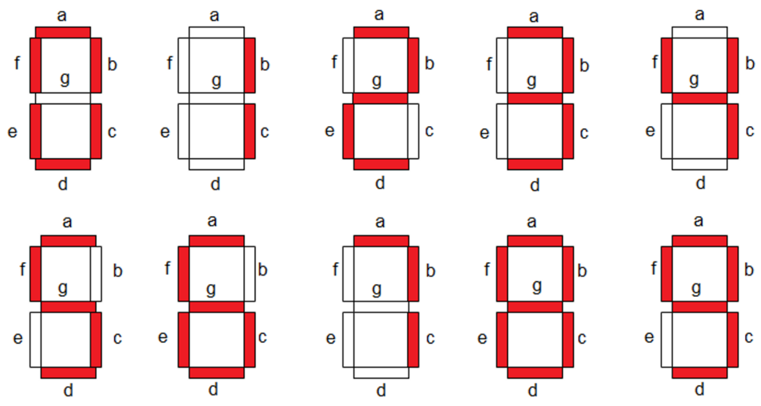
```
    HAL_GPIO_WritePin(LED_YELLOW_2_GPIO_Port,
 LED_YELLOW_2_Pin,GPIO_PIN_SET);
    HAL_GPIO_WritePin(LED_GREEN_2_GPIO_Port,
 LED_GREEN_2_Pin,GPIO_PIN_SET);
  counter++;
  if(counter ==3){led_status = YELLOW;
  counter =0;
  }
  break;
  default:
  break;
  }

  HAL_Delay(1000);
  }

  /* USER CODE END 3 */
}
```

Program 1.3: Source code

1.4 Exercise 4

Add **only one 7 led segment** to the schematic in Exercise 3. This component can be found in Proteus by the keyword **7SEG-COM-ANODE**. For this device, the common pin should be connected to the power supply and other pins are supposed to connected to PB0 to PB6. Therefore, to turn-on a segment in this 7SEG, the STM32 pin should be in logic 0 (0V).

Implement a function named **display7SEG(int num)**. The input for this function is from 0 to 9 and the outputs are listed as following:



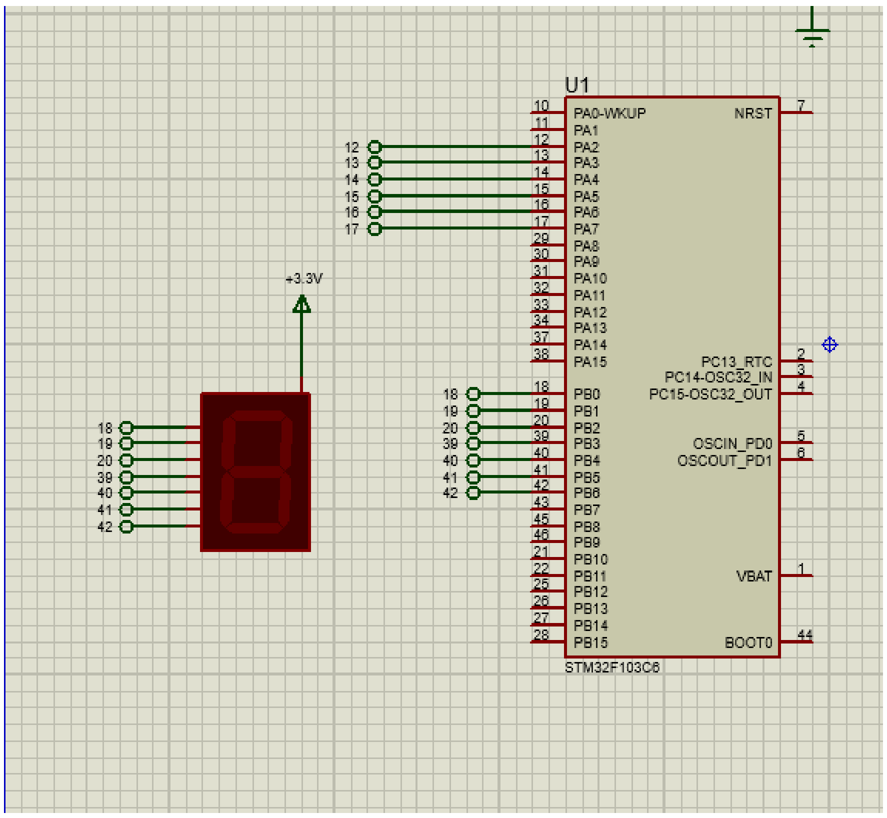
Hình 1.5: Display a number on 7 segment LED

This function is invoked in the while loop for testing as following:

```c
int counter = 0;
while (1){
    if(counter >= 10) counter = 0;
    display7SEG(counter++);
    HAL_Delay(1000);

}
```

Program 1.4: An example for your source code

Report 1: Present the schematic.
Report 2: Present the source code for display7SEG function.



Hình 1.6: Schematic from Proteus

```c
#ifndef INC_7SEG_H_
#define INC_7SEG_H_
#include "stm32f1xx_it.h"
#include "main.h"
void display7SEG(int num);

#endif /* INC_7SEG_H_ */
```

Program 1.5: In 7SEG.h

```c
#include "7SEG.h"
void display7SEG(int num){
  // Turn off all segments first (for common anode, set
all to HIGH)
 HAL_GPIO_WritePin(A_SEG_GPIO_Port, A_SEG_Pin,
GPIO_PIN_SET);
    HAL_GPIO_WritePin(B_SEG_GPIO_Port, B_SEG_Pin,
GPIO_PIN_SET);
 HAL_GPIO_WritePin(C_SEG_GPIO_Port, C_SEG_Pin,
GPIO_PIN_SET);
 HAL_GPIO_WritePin(D_SEG_GPIO_Port, D_SEG_Pin,
GPIO_PIN_SET);
    HAL_GPIO_WritePin(E_SEG_GPIO_Port, E_SEG_Pin,
GPIO_PIN_SET);
 HAL_GPIO_WritePin(F_SEG_GPIO_Port, F_SEG_Pin,
GPIO_PIN_SET);
    HAL_GPIO_WritePin(G_SEG_GPIO_Port, G_SEG_Pin,
GPIO_PIN_SET);
switch(num) {
      case 0:
    HAL_GPIO_WritePin(A_SEG_GPIO_Port, A_SEG_Pin,
GPIO_PIN_RESET);
      HAL_GPIO_WritePin(B_SEG_GPIO_Port, B_SEG_Pin,
GPIO_PIN_RESET);
      HAL_GPIO_WritePin(C_SEG_GPIO_Port, C_SEG_Pin,
GPIO_PIN_RESET);
          HAL_GPIO_WritePin(D_SEG_GPIO_Port, D_SEG_Pin,
GPIO_PIN_RESET);
          HAL_GPIO_WritePin(E_SEG_GPIO_Port, E_SEG_Pin,
GPIO_PIN_RESET);
          HAL_GPIO_WritePin(F_SEG_GPIO_Port, F_SEG_Pin,
GPIO_PIN_RESET);
          break;
       case 1:
          HAL_GPIO_WritePin(B_SEG_GPIO_Port, B_SEG_Pin,
GPIO_PIN_RESET);
          HAL_GPIO_WritePin(C_SEG_GPIO_Port, C_SEG_Pin,
GPIO_PIN_RESET);
          break;
       case 2:
          HAL_GPIO_WritePin(A_SEG_GPIO_Port, A_SEG_Pin,
GPIO_PIN_RESET);
          HAL_GPIO_WritePin(B_SEG_GPIO_Port, B_SEG_Pin,
GPIO_PIN_RESET);
          HAL_GPIO_WritePin(G_SEG_GPIO_Port, G_SEG_Pin,
GPIO_PIN_RESET);
          HAL_GPIO_WritePin(E_SEG_GPIO_Port, E_SEG_Pin,
GPIO_PIN_RESET);
          HAL_GPIO_WritePin(D_SEG_GPIO_Port, D_SEG_Pin,
GPIO_PIN_RESET);
          break;
```

```c
        case 3:
            HAL_GPIO_WritePin(A_SEG_GPIO_Port, A_SEG_Pin, GPIO_PIN_RESET);
            HAL_GPIO_WritePin(B_SEG_GPIO_Port, B_SEG_Pin, GPIO_PIN_RESET);
            HAL_GPIO_WritePin(C_SEG_GPIO_Port, C_SEG_Pin, GPIO_PIN_RESET);
            HAL_GPIO_WritePin(D_SEG_GPIO_Port, D_SEG_Pin, GPIO_PIN_RESET);
            HAL_GPIO_WritePin(G_SEG_GPIO_Port, G_SEG_Pin, GPIO_PIN_RESET);
            break;
        case 4:
            HAL_GPIO_WritePin(F_SEG_GPIO_Port, F_SEG_Pin, GPIO_PIN_RESET);
            HAL_GPIO_WritePin(G_SEG_GPIO_Port, G_SEG_Pin, GPIO_PIN_RESET);
            HAL_GPIO_WritePin(B_SEG_GPIO_Port, B_SEG_Pin, GPIO_PIN_RESET);
            HAL_GPIO_WritePin(C_SEG_GPIO_Port, C_SEG_Pin, GPIO_PIN_RESET);
            break;
        case 5:
            HAL_GPIO_WritePin(A_SEG_GPIO_Port, A_SEG_Pin, GPIO_PIN_RESET);
            HAL_GPIO_WritePin(F_SEG_GPIO_Port, F_SEG_Pin, GPIO_PIN_RESET);
            HAL_GPIO_WritePin(G_SEG_GPIO_Port, G_SEG_Pin, GPIO_PIN_RESET);
            HAL_GPIO_WritePin(C_SEG_GPIO_Port, C_SEG_Pin, GPIO_PIN_RESET);
            HAL_GPIO_WritePin(D_SEG_GPIO_Port, D_SEG_Pin, GPIO_PIN_RESET);
            break;
        case 6:
            HAL_GPIO_WritePin(A_SEG_GPIO_Port, A_SEG_Pin, GPIO_PIN_RESET);
            HAL_GPIO_WritePin(F_SEG_GPIO_Port, F_SEG_Pin, GPIO_PIN_RESET);
            HAL_GPIO_WritePin(G_SEG_GPIO_Port, G_SEG_Pin, GPIO_PIN_RESET);
            HAL_GPIO_WritePin(C_SEG_GPIO_Port, C_SEG_Pin, GPIO_PIN_RESET);
            HAL_GPIO_WritePin(D_SEG_GPIO_Port, D_SEG_Pin, GPIO_PIN_RESET);
            HAL_GPIO_WritePin(E_SEG_GPIO_Port, E_SEG_Pin, GPIO_PIN_RESET);
            break;
        case 7:
            HAL_GPIO_WritePin(A_SEG_GPIO_Port, A_SEG_Pin, GPIO_PIN_RESET);
            HAL_GPIO_WritePin(B_SEG_GPIO_Port, B_SEG_Pin, GPIO_PIN_RESET);
            HAL_GPIO_WritePin(C_SEG_GPIO_Port, C_SEG_Pin, GPIO_PIN_RESET);
            break;
        case 8:
            HAL_GPIO_WritePin(A_SEG_GPIO_Port, A_SEG_Pin, GPIO_PIN_RESET);
            HAL_GPIO_WritePin(B_SEG_GPIO_Port, B_SEG_Pin, GPIO_PIN_RESET);
            HAL_GPIO_WritePin(C_SEG_GPIO_Port, C_SEG_Pin, GPIO_PIN_RESET);
            HAL_GPIO_WritePin(D_SEG_GPIO_Port, D_SEG_Pin, GPIO_PIN_RESET);
            HAL_GPIO_WritePin(E_SEG_GPIO_Port, E_SEG_Pin, GPIO_PIN_RESET);
            HAL_GPIO_WritePin(F_SEG_GPIO_Port, F_SEG_Pin, GPIO_PIN_RESET);
            HAL_GPIO_WritePin(G_SEG_GPIO_Port, G_SEG_Pin, GPIO_PIN_RESET);
            break;
        case 9:
            HAL_GPIO_WritePin(A_SEG_GPIO_Port, A_SEG_Pin, GPIO_PIN_RESET);
            HAL_GPIO_WritePin(B_SEG_GPIO_Port, B_SEG_Pin, GPIO_PIN_RESET);
            HAL_GPIO_WritePin(C_SEG_GPIO_Port, C_SEG_Pin, GPIO_PIN_RESET);
            HAL_GPIO_WritePin(D_SEG_GPIO_Port, D_SEG_Pin, GPIO_PIN_RESET);
            HAL_GPIO_WritePin(F_SEG_GPIO_Port, F_SEG_Pin, GPIO_PIN_RESET);
            HAL_GPIO_WritePin(G_SEG_GPIO_Port, G_SEG_Pin, GPIO_PIN_RESET);
            break;
        default:
            // Invalid number, turn off all segments
            HAL_GPIO_WritePin(A_SEG_GPIO_Port, A_SEG_Pin, GPIO_PIN_SET);
            HAL_GPIO_WritePin(B_SEG_GPIO_Port, B_SEG_Pin, GPIO_PIN_SET);
            HAL_GPIO_WritePin(C_SEG_GPIO_Port, C_SEG_Pin, GPIO_PIN_SET);
            HAL_GPIO_WritePin(D_SEG_GPIO_Port, D_SEG_Pin, GPIO_PIN_SET);
            HAL_GPIO_WritePin(E_SEG_GPIO_Port, E_SEG_Pin, GPIO_PIN_SET);
            HAL_GPIO_WritePin(F_SEG_GPIO_Port, F_SEG_Pin,
  GPIO_PIN_SET);
            HAL_GPIO_WritePin(G_SEG_GPIO_Port, G_SEG_Pin,
  GPIO_PIN_SET);
            break;
      }
  }
```

Program 1.6: In 7SEG.c

1.5 Exercise 5

Integrate the 7SEG-LED to the 4 way traffic light. In this case, the 7SEG-LED is used to display countdown value.

```
void runLed2Timer();
 int counter=0;
    int countdown = 0;
    int countdown2=0;
    int led_status = RED;

    while (1)
    {
      /* USER CODE END WHILE */
    switch (led_status) {
    case RED:
    HAL_GPIO_WritePin(LED_RED_GPIO_Port, LED_RED_Pin,
  GPIO_PIN_RESET);
    HAL_GPIO_WritePin(LED_YELLOW_GPIO_Port, LED_YELLOW_Pin,
  GPIO_PIN_SET);
    HAL_GPIO_WritePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin,
  GPIO_PIN_SET);

    HAL_GPIO_WritePin(LED_RED_2_GPIO_Port, LED_RED_2_Pin,
  GPIO_PIN_SET);
    HAL_GPIO_WritePin(LED_YELLOW_2_GPIO_Port,
  LED_YELLOW_2_Pin,GPIO_PIN_SET);
    HAL_GPIO_WritePin(LED_GREEN_2_GPIO_Port,
  LED_GREEN_2_Pin,GPIO_PIN_RESET);
    countdown = 5 - counter; // using 2 counter

    display7SEG(countdown);
    counter++;
    if(counter>3){
    HAL_GPIO_WritePin(LED_RED_2_GPIO_Port, LED_RED_2_Pin,
  GPIO_PIN_SET);
    HAL_GPIO_WritePin(LED_YELLOW_2_GPIO_Port,
  LED_YELLOW_2_Pin,GPIO_PIN_RESET);
  HAL_GPIO_WritePin(LED_GREEN_2_GPIO_Port,
LED_GREEN_2_Pin,GPIO_PIN_SET);

  }
  if(counter==5) {
  led_status = GREEN;
  counter =0;
  }
  break;
  case YELLOW:
  HAL_GPIO_WritePin(LED_RED_GPIO_Port, LED_RED_Pin,
GPIO_PIN_SET);
  HAL_GPIO_WritePin(LED_YELLOW_GPIO_Port, LED_YELLOW_Pin,
GPIO_PIN_RESET);
  HAL_GPIO_WritePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin,
GPIO_PIN_SET);

  HAL_GPIO_WritePin(LED_RED_2_GPIO_Port, LED_RED_2_Pin,
GPIO_PIN_RESET);
  HAL_GPIO_WritePin(LED_YELLOW_2_GPIO_Port,
LED_YELLOW_2_Pin,GPIO_PIN_SET);
  HAL_GPIO_WritePin(LED_GREEN_2_GPIO_Port,
LED_GREEN_2_Pin,GPIO_PIN_SET);

   countdown = 2 - counter;

   display7SEG(countdown);
  counter++;
  if(counter ==2){led_status = RED;

  counter =0;
  }
  break;
  case GREEN:
  HAL_GPIO_WritePin(LED_RED_GPIO_Port, LED_RED_Pin,
GPIO_PIN_SET);
  HAL_GPIO_WritePin(LED_YELLOW_GPIO_Port, LED_YELLOW_Pin,
GPIO_PIN_SET);
  HAL_GPIO_WritePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin,
GPIO_PIN_RESET);

  HAL_GPIO_WritePin(LED_RED_2_GPIO_Port, LED_RED_2_Pin,
GPIO_PIN_RESET);
   HAL_GPIO_WritePin(LED_YELLOW_2_GPIO_Port,
LED_YELLOW_2_Pin,GPIO_PIN_SET);
   HAL_GPIO_WritePin(LED_GREEN_2_GPIO_Port,
LED_GREEN_2_Pin,GPIO_PIN_SET);
   countdown = 3 - counter;
   countdown2= 5-counter;
```

```c
     display7SEG(countdown);
    counter++;
    if(counter ==3){led_status = YELLOW;
    counter =0;
    }
    break;
    default:
    break;
    }

    HAL_Delay(1000);
    }
```

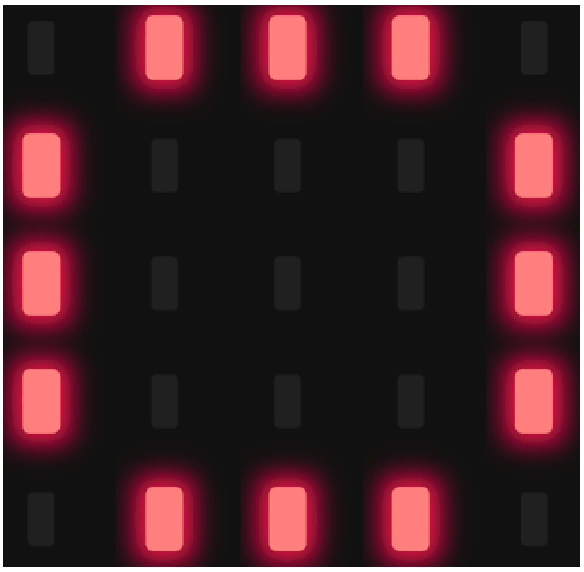
Program 1.7: In main.c

```c
//FUCTION COUNTER FOR TRAFIC LIGHT 2
void runLed2Timer()
{
    static int led2_counter = 0;
    static int led2_phase = 0;  // 0: GREEN, 1: YELLOW, 2:
   RED
    int countdown2 = 0;

    switch (led2_phase) {
    case 0:  // GREEN phase (3 seconds)
        countdown2 = 3 - led2_counter;
        break;

    case 1:  // YELLOW phase (2 seconds)
        countdown2 = 2 - led2_counter;
        break;

    case 2:  // RED phase (5 seconds)
        countdown2 = 5 - led2_counter;
        break;
    }

    display7SEG(countdown2);

    led2_counter++;

    if ((led2_phase == 0 && led2_counter == 3) ||
        (led2_phase == 1 && led2_counter == 2) ||
        (led2_phase == 2 && led2_counter == 5)) {
        led2_counter = 0;
        led2_phase = (led2_phase + 1) % 3;
    }
}
```

Program 1.8: Function counter for Traffic light 2

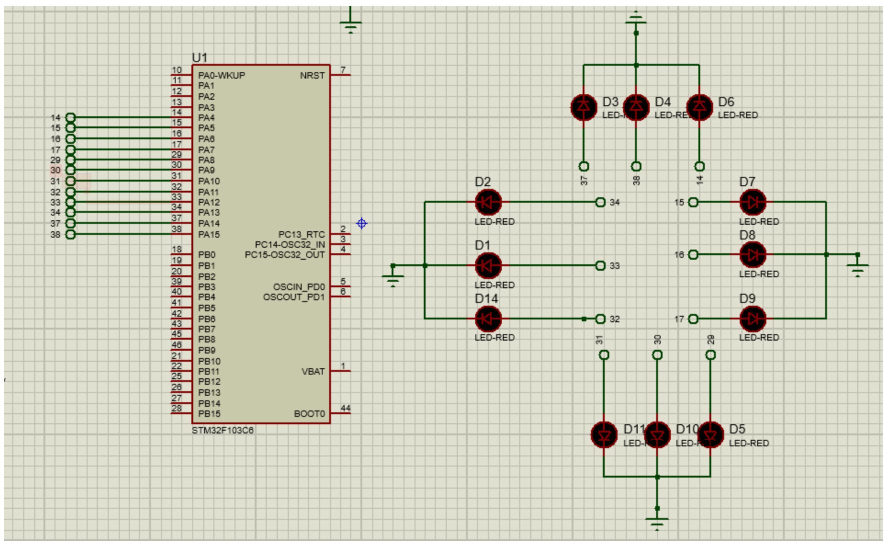
1.6 Exercise 6

In this exercise, a new Proteus schematic is designed to simulate an analog clock, with 12 different number. The connections for 12 LEDs are supposed from PA4 to PA15 of the STM32. The arrangement of 12 LEDs is depicted as follows.



Hình 1.7: 12 LEDs for an analog clock

Report 1: Present the schematic.



Hình 1.8: Schematic from Proteus

Report 2: Implement a simple program to test the connection of every single LED. This testing program should turn every LED in a sequence

```
/*
 * 7SEG.h
 *
 *  Created on: Aug 31, 2024
 *      Author: ADMIN
```

```c
 */

#ifndef INC_7SEG_H_
#define INC_7SEG_H_

#include "main.h"

typedef struct {
  GPIO_TypeDef * GPIO_Port;
  uint16_t GPIO_Pin;

}LED_CONTROL; // LED CONTROLING HANDLE

LED_CONTROL leds[]={
        {LED_1_GPIO_Port, LED_1_Pin},
        {LED_2_GPIO_Port, LED_2_Pin},
        {LED_3_GPIO_Port, LED_3_Pin},
        {LED_4_GPIO_Port, LED_4_Pin},
        {LED_5_GPIO_Port, LED_5_Pin},
        {LED_6_GPIO_Port, LED_6_Pin},
        {LED_7_GPIO_Port, LED_7_Pin},
        {LED_8_GPIO_Port, LED_8_Pin},
        {LED_9_GPIO_Port, LED_9_Pin},
        {LED_10_GPIO_Port, LED_10_Pin},
        {LED_11_GPIO_Port, LED_11_Pin},
        {LED_12_GPIO_Port, LED_12_Pin},
};


void display7SEG(int num); // Maybe use later


#endif /* INC_7SEG_H_ */
```

Program 1.9: We reuse header of 7SEG.h

```c
    int count= sizeof(leds)/sizeof(leds[0]);
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    /* USER CODE END WHILE */
        for(int i=0;i<count;i++){

          HAL_GPIO_WritePin(leds[i].GPIO_Port, leds[i].
   GPIO_Pin, GPIO_PIN_SET);
          HAL_Delay(1000);


        }

    /* USER CODE BEGIN 3 */
  }
```

Program 1.10: Testing fuction

1.7 Exercise 7

Implement a function named **clearAllClock()** to turn off all 12 LEDs. Present the source code of this function.

```
void clearAllClock(){
    for(int i=0;i<count;i++){
        HAL_GPIO_WRITEPIN(leds[i],GPIO_Port,leds[i].
   GPIO_Pin,GPIO_PIN_RESET);

    }
}
```

Program 1.11: Function Implementation

1.8 Exercise 8

Implement a function named **setNumberOnClock(int num)**. The input for this function is from **0 to 11** and an appropriate LED is turn on. Present the source code of this function.

```
void setNumberOnClock(int num){
HAL_GPIO_WritePin(leds[num-1],GPIO_Port,leds[num-1].
   GPIO_Pin,GPIO_PIN_RESET);

}
```

Program 1.12: Function Implementation

1.9 Exercise 9

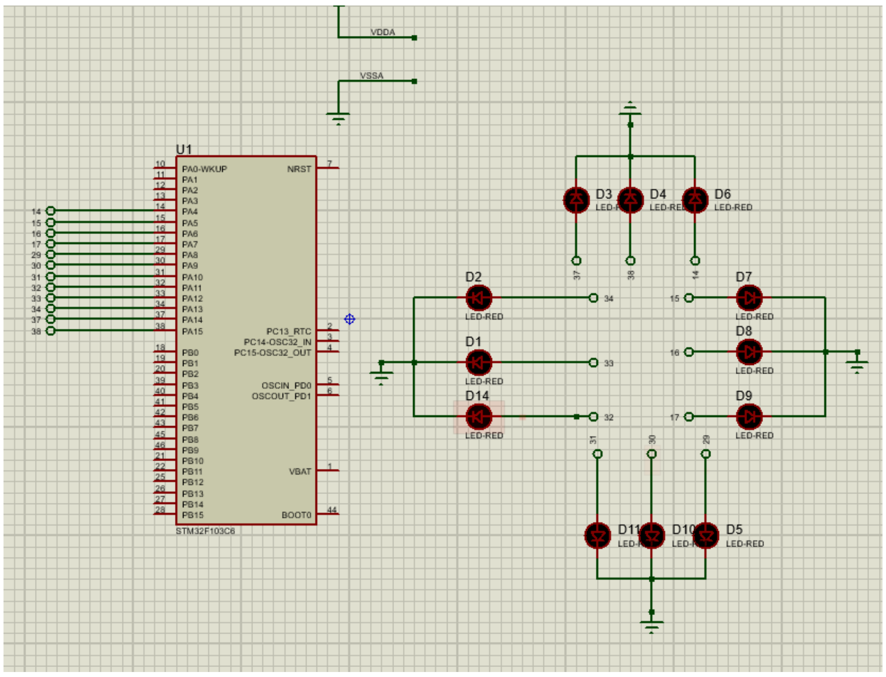
Implement a function named **clearNumberOnClock(int num)**. The input for this function is from **0 to 11** and an appropriate LED is turn off.

```
void setNumberOnClock(int num){
   HAL_GPIO_WritePin(leds[num-1],GPIO_Port,leds[num-1].
   GPIO_Pin,GPIO_PIN_SET);

}
```

Program 1.13: Function Implementation

1.10 Exercise 10

Integrate the whole system and use 12 LEDs to display a clock. At a given time, there are only 3 LEDs are turn on for hour, minute and second information. Driver



Hình 1.9: Schematic from Proteus

Function:

```
/*
 * digitalclock.h
 *
 *  Created on: Sep 1, 2024
 *      Author: ADMIN
 */
#ifndef INC_DIGITALCLOCK_H_
#define INC_DIGITALCLOCK_H_
#include "main.h"
#include "stm32f1xx_it.h"
typedef struct {
  GPIO_TypeDef *GPIO_Port;
  uint16_t GPIO_Pin;
}LED_CONTROL;
 extern LED_CONTROL leds[];
 extern count;
 void updateClock(int hour, int minute, int second);
 void ClearAllClock(void);
 void setNumberOnClock(int num);
#endif /* INC_DIGITALCLOCK_H_ */
```

Program 1.14: Function Declaration in digitalclock.h

Driver Function:

```c
/*
 * digitalclock.c
 *
 *  Created on: Sep 1, 2024
 *      Author: ADMIN
 */
#include "digitalclock.h"
LED_CONTROL leds[]={
        {LED_1_GPIO_Port, LED_1_Pin},
        {LED_2_GPIO_Port, LED_2_Pin},
        {LED_3_GPIO_Port, LED_3_Pin},
        {LED_4_GPIO_Port, LED_4_Pin},
        {LED_5_GPIO_Port, LED_5_Pin},
        {LED_6_GPIO_Port, LED_6_Pin},
        {LED_7_GPIO_Port, LED_7_Pin},
        {LED_8_GPIO_Port, LED_8_Pin},
        {LED_9_GPIO_Port, LED_9_Pin},
        {LED_10_GPIO_Port, LED_10_Pin},
        {LED_11_GPIO_Port, LED_11_Pin},
        {LED_12_GPIO_Port, LED_12_Pin},
};
int count= sizeof(leds)/sizeof(leds[0]);
void updateClock(int hour, int minute, int second){
    ClearAllClock();
      if (hour > 12) {
          hour = hour % 12;
      }
      if (hour == 0) {
          hour = 12;
      }
      int hour_pos = (hour - 1);
      int minute_pos = (minute / 5) % 12;
      int second_pos = (second / 5) % 12;
      if (second_pos == 0) {
                 second_pos = 12;
           }
          if (minute_pos == 0) {
             second_pos = 12;
            }

      setNumberOnClock(hour_pos);
      setNumberOnClock(minute_pos-1);
      setNumberOnClock(second_pos-1);
 }

 void ClearAllClock(void){

   for(int i=0;i<count;i++){
        HAL_GPIO_WritePin(leds[i].GPIO_Port,leds[i].GPIO_Pin,
      GPIO_PIN_RESET);
            }


  }
  void setNumberOnClock(int num){
  if (num >= 0 && num < 12) {
   HAL_GPIO_WritePin(leds[num].GPIO_Port, leds[num].GPIO_Pin
    , GPIO_PIN_SET);

          }
}
```

Program 1.15: Function Declaration in digitalclock.c

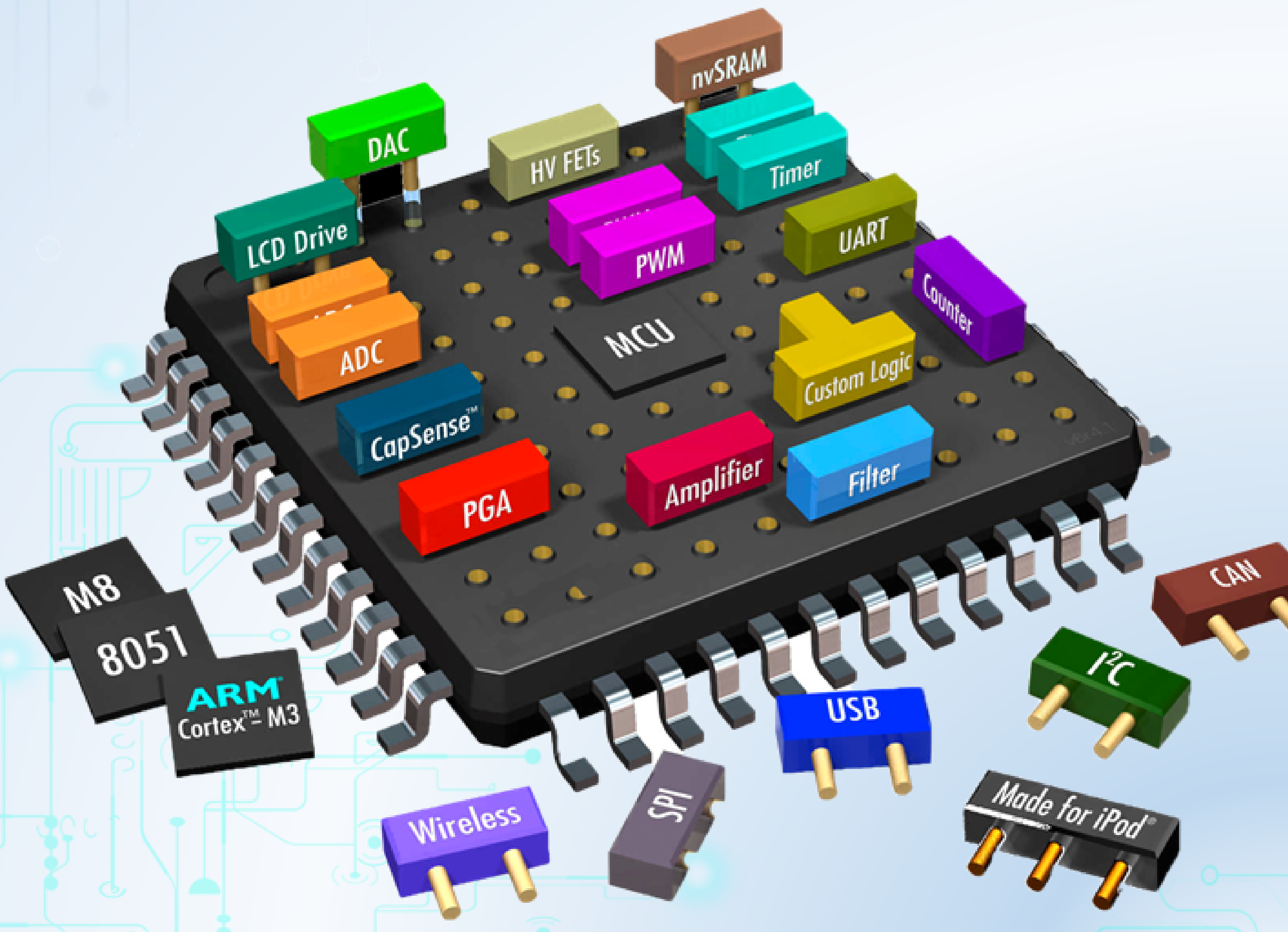
Testing Function:

```c
    int hour=12;
    int minute=5;
    int second=30;
   while (1)
   {
     /* USER CODE END WHILE */
     updateClock(hour,minute, second);
     second++;
             if (second >= 60) {
                 second = 0+5;
                 minute++;
                 if (minute >= 60) {
                     minute = 0;
                     hour++;
                     if (hour > 12) {
                         hour = 1;
                     }
                 }
             }

             HAL_Delay(10);
         }

   }

```

Program 1.16: Function Implementation in main.c

CHƯƠNG 2

Timer Interrupt and LED Scanning



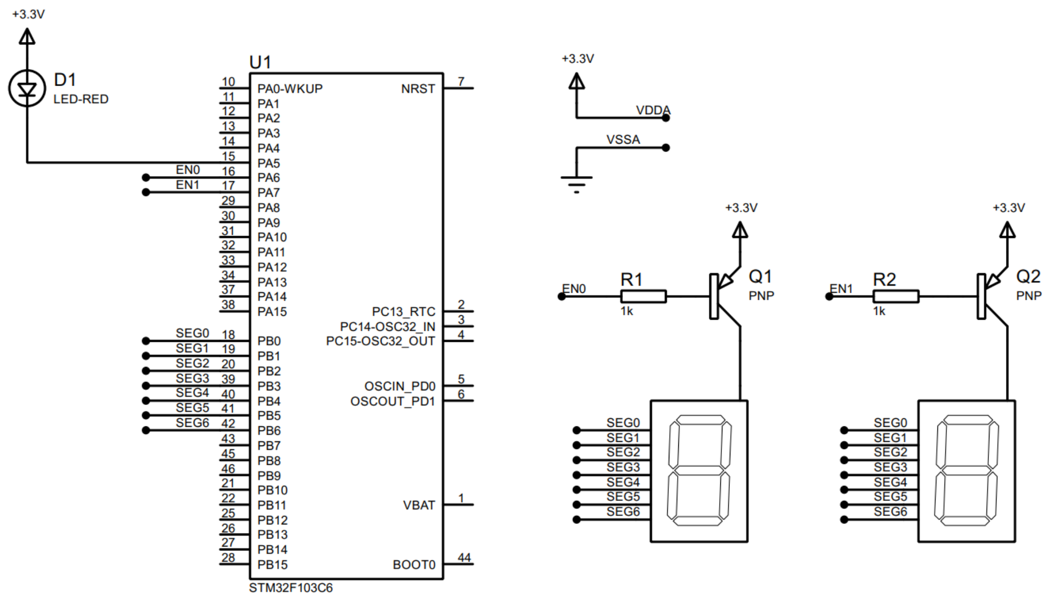
1 Exercise and Report

1.1 Exercise 1

The first exercise show how to interface for multiple seven segment LEDs to STM32F103C6 micro-controller (MCU). Seven segment displays are common anode type, meaning that the anode of all LEDs are tied together as a single terminal and cathodes are left alone as individual pins.

In order to save the resource of the MCU, individual cathode pins from all the seven segment LEDs are connected together, and connect to 7 pins of the MCU. These pins are popular known as the **signal pins**. Meanwhile, the anode pin of each seven segment LEDs are controlled under a power enabling circuit, for instance, an PNP transistor. At a given time, only one seven segment LED is turned on. However, if the delay is small enough, it seems that all LEDs are enabling.

Implement the circuit simulation in Proteus with two 7-SEGMENT LEDs as following:



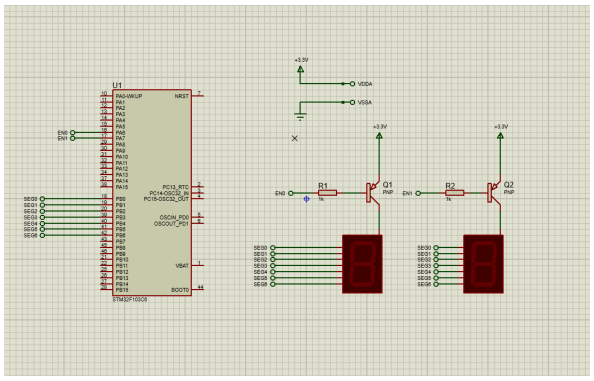
Hình 2.1: Schematic from Proteus

Components used in the schematic are listed bellow:

- 7SEG-COM-ANODE (connected from PB0 to PB6)
- LED-RED
- PNP
- RES
- STM32F103C6

Students are proposed to use the function **display7SEG(int num)** in the Lab 1 in this exercise. Implement the source code in the interrupt callback function to display number **"1"** on the first seven segment and number **"2"** for second one. The switching time between 2 LEDs is half of second.

Report 1: Capture your schematic from Proteus and show in the report.



Hình 2.2: Schematic from Proteus

Report 2: Source code in the **HAL_TIM_PeriodElapsedCallback** function.

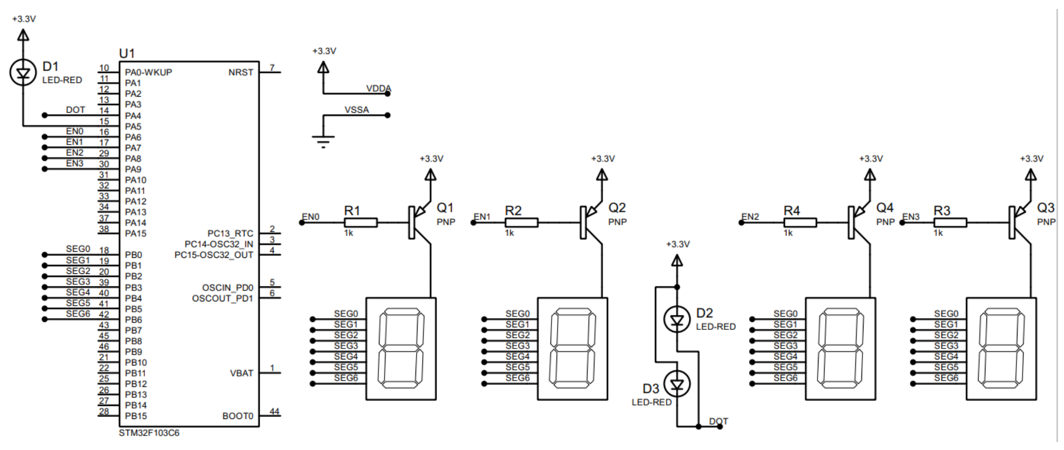
```c
/* USER CODE BEGIN 4 */
int counter=50;
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
   {
   counter--;

      if (counter >= 25) {
          state = LED1;
      } else if (counter > 0) {
          state = LED2;
      } else {
          counter = 50;
      }

      controlLed();
}
/* USER CODE END 4 */
```

Program 2.1: Add an interrupt service routine

Short question: These LEDs are scanned at freqeucy $f = 1/2T_S$ = **2 Hz**.

1.2 Exercise 2

Extend to 4 seven segment LEDs and two LEDs (connected to PA4, labeled as **DOT**) in the middle as following:



Hình 2.3: Schematic from Proteus

Blink the two LEDs every second. Meanwhile, number 3 is displayed on the third seven segment and number 0 is displayed on the last one (to present 12 hour and a half). The switching time for each seven segment LED is also a half of second (500ms). **Implement your code in the timer interrupt function.**

Report 1: Capture your schematic from Proteus and show in the report.

Report 2: Present your source code in the **HAL_TIM_PeriodElapsedCallback** function.

```c
/* USER CODE BEGIN 4 */
int counter=200;
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
     counter--;
   if(counter==200) HAL_GPIO_TogglePin(DOT_GPIO_Port, DOT_Pin);


   if (counter >= 150) {
          state = LED1;
          if(counter==150)HAL_GPIO_TogglePin(DOT_GPIO_Port, DOT_Pin);
      }
      else if (counter >= 100) {
          state = LED2;
          if(counter==100) HAL_GPIO_TogglePin(DOT_GPIO_Port, DOT_Pin);
      }
      else if (counter >= 50) {
          state = LED3;
          if(counter==50)HAL_GPIO_TogglePin(DOT_GPIO_Port, DOT_Pin);
      }
```

```
        else if (counter > 0) {
            state = LED4;
            if(counter==0)HAL_GPIO_TogglePin(DOT_GPIO_Port,
   DOT_Pin);
        }
        else {
            counter = 200;
         controlLed();  ;
}
/* USER CODE END 4 */
```

Program 2.2: Source code for EX2

Short question: What is the frequency of the scanning process?
These LEDs are scanned at frequecy $f = 1/4T_S$ = **0.5 Hz**.

1.3 Exercise 3

Implement a function named **update7SEG(int index)**. An array of 4 integer numbers are declared in this case. The code skeleton in this exercise is presented as following:

This function should be invoked in the timer interrupt, e.g update7SEG(index_led++). The variable **index_led** is updated to stay in a valid range, which is from 0 to 3.

Report 1: Present the source code of the update7SEG function.

```c
const int MAX_LED = 4;
int index_led = 0;
int led_buffer[4] = {1, 2, 3, 4};
void update7SEG(int index){
switch (index){
    case 0:
        display7SEG(led_buffer[0]);
       //EN0 ENABLE
HAL_GPIO_WritePin(EN0_GPIO_Port,EN0_Pin,GPIO_PIN_RESET);
HAL_GPIO_WritePin(EN1_GPIO_Port,EN1_Pin,GPIO_PIN_SET);
HAL_GPIO_WritePin(EN2_GPIO_Port,EN2_Pin,GPIO_PIN_SET);
HAL_GPIO_WritePin(EN3_GPIO_Port,EN3_Pin,GPIO_PIN_SET);
        break;
    case 1:
        display7SEG(led_buffer[1]);
        //EN1 ENABLE
HAL_GPIO_WritePin(EN0_GPIO_Port,EN0_Pin,GPIO_PIN_SET);
HAL_GPIO_WritePin(EN1_GPIO_Port,EN1_Pin,GPIO_PIN_RESET);
HAL_GPIO_WritePin(EN2_GPIO_Port,EN2_Pin,GPIO_PIN_SET);
HAL_GPIO_WritePin(EN3_GPIO_Port,EN3_Pin,GPIO_PIN_SET);
        break;
    case 2:
        display7SEG(led_buffer[2]);
        //EN2 ENABLE
HAL_GPIO_WritePin(EN0_GPIO_Port,EN0_Pin,GPIO_PIN_SET);
HAL_GPIO_WritePin(EN1_GPIO_Port,EN1_Pin,GPIO_PIN_SET);
HAL_GPIO_WritePin(EN2_GPIO_Port,EN2_Pin,GPIO_PIN_RESET);
HAL_GPIO_WritePin(EN3_GPIO_Port,EN3_Pin,GPIO_PIN_SET);
        break;
    case 3:
        display7SEG(led_buffer[3]);
        //EN3 ENABLE
HAL_GPIO_WritePin(EN0_GPIO_Port,EN0_Pin,GPIO_PIN_SET);
HAL_GPIO_WritePin(EN1_GPIO_Port,EN1_Pin,GPIO_PIN_SET);
HAL_GPIO_WritePin(EN2_GPIO_Port,EN2_Pin,GPIO_PIN_SET);
HAL_GPIO_WritePin(EN3_GPIO_Port,EN3_Pin,GPIO_PIN_RESET);
        break;
    default:
        break;
}

}
```

Program 2.3: update7SEG

Report 2: Present the source code in the HAL_TIM_PeriodElapsedCallback.

```c
/* USER CODE BEGIN 4 */
void initEx2(){

  update7SEG(index_led);
          index_led++;

  if (index_led >= MAX_LED) {
              index_led = 0;
      }
}
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
   {
    initEx2();
}
/* USER CODE END 4 */
```

Program 2.4: Source code for EX3

Students are proposed to change the values in the **led_buffer** array for unit test this function, which is used afterward.

1.4 Exercise 4

Change the period of invoking update7SEG function in order to set the frequency of 4 seven segment LEDs to 1Hz. The DOT is still blinking every second.
Report 1: Present the source code in the **HAL_TIM_PeriodElapsedCallback**.

```c
void initEx4(){
    led_update_counter++;
    led_blink_counter--;
    if (led_update_counter >= 25) {  // 25 * 10ms = 250ms
        update7SEG(index_led);
        index_led++;
        if (index_led >= MAX_LED) {
            index_led = 0;
        }
        led_update_counter = 0;
    }
    if (led_blink_counter <= 0) {
        HAL_GPIO_TogglePin(DOT_GPIO_Port, DOT_Pin);
        led_blink_counter = 100;
    }


}
```

Program 2.5: Source code for EX4

```c
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
  {
    initEx4();

}
```

Program 2.6: Source code for EX4

1.5 Exercise 5

Implement a digital clock with **hour** and **minute** information displayed by 2 seven segment LEDs. The code skeleton in the **main** function is presented as follows:

```c
int hour = 15, minute = 8, second = 50;

while(1){
    second++;
    if (second >= 60){
        second = 0;
        minute++;
    }
    if(minute >= 60){
        minute = 0;
        hour++;
    }
    if(hour >=24){
        hour = 0;
    }
    updateClockBuffer();
    HAL_Delay(1000);
}
```

Program 2.7: An example for your source code

The function **updateClockBuffer** will generate values for the array **led_buffer** according to the values of hour and minute. In the case these values are 1 digit number, digit 0 is added.

Report 1: Present the source code in the **updateClockBuffer** function.

```c
void  updateClockBuffer(int hour, int minute){
  if (hour >= 10) {
    led_buffer[0] = hour / 10;
    led_buffer[1] = hour % 10;
     } else {
      led_buffer[0] = 0;
    led_buffer[1] = hour;
    }
  if (minute >= 10) {
      led_buffer[2] = minute / 10;
        led_buffer[3] = minute % 10;
     } else {
    led_buffer[2] = 0;
    led_buffer[3] = minute;
        }
}
```

Program 2.8: Source code for EX5

1.6 Exercise 6

The main target from this exercise to reduce the complexity (or reduce code processing) in the timer interrupt. The time consumed in the interrupt can lead to the nested interrupt issue, which can crash the whole system. A simple solution can disable the timer whenever the interrupt occurs, the enable it again. However, the real-time processing is not guaranteed anymore.

In this exercise, a software timer is created and its counter is count down every timer interrupt is raised (every 10ms). By using this timer, the **Hal_Delay(1000)** in the main function is removed. In a MCU system, non-blocking delay is better than blocking delay. The details to create a software timer are presented bellow. The source code is added to your current program, **do not delete the source code you have on Exercise 5.**

Step 1: Declare variables and functions for a software timer, as following:

```c
/* USER CODE BEGIN 0 */
int timer0_counter = 0;
int timer0_flag = 0;
int TIMER_CYCLE = 10;
void setTimer0(int duration){
  timer0_counter = duration /TIMER_CYCLE;
  timer0_flag = 0;
}
void timer_run(){
  if(timer0_counter > 0){
    timer0_counter--;
    if(timer0_counter == 0) timer0_flag = 1;
  }
}
/* USER CODE END 0 */
```
Program 2.9: Software timer based timer interrupt

Please change the **TIMER_CYCLE** to your timer interrupt period. In the manual code above, it is **10ms**.

Step 2: The **timer_run()** is invoked in the timer interrupt as following:

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim){

  timer_run();

  //YOUR OTHER CODE
}
```
Program 2.10: Software timer based timer interrupt

Step 3: Use the timer in the main function by invoked setTimer0 function, then check for its flag (timer0_flag). An example to blink an LED connected to PA5 using software timer is shown as follows:

```
1 setTimer0(1000);
2 while (1){
3     if(timer0_flag == 1){
4         HAL_GPIO_TogglePin(LED_RED_GPIO_Port, LED_RED_Pin);
5         setTimer0(2000);
6     }
7 }
```
Program 2.11: Software timer is used in main fuction to blink the LED

Report 1: if in line 1 of the code above is miss, what happens after that and why?

```
If in line 1 of the code above is miss
- The variable timer0_counter will not be created and always equal 0
- Variable timer0_flag will not be equal to 1 because the timer0_counter never decrease from 1000 to 0
- All of the code in while(1) loop will not be executed
```
Program 2.12: Answer for Report 1

Report 2: if in line 1 of the code above is changed to setTimer0(1), what happens after that and why?

```
If in line 1 of the code above is miss :
- The LED will blink very fast so we can not see by normal
   eyes
- Also if we have multiples interupts and the Timer input
   parameters are small enough will causing Interupt
   overload
- Interrupt timing error: Software timers rely on
   interruptsto function. In the case of setTimer0(1), the
   interrupt happens very quickly (after just 10ms).
   However, if other higher-priority tasks are running when
    the interrupt occurs, the interrupt may be slightly
delayed. This can cause timing inaccuracies, leading the
   program to blink the LED at imprecise intervals.
```

Program 2.13: Answer for Report 1

Report 3: if in line 1 of the code above is changed to setTimer0(10), what is changed compared to 2 first questions and why?

```
- Compared to setTimer0(1000);, using setTimer0(10); will
   make the LED blink much faster (every 100ms instead of
   every 10 seconds).
- Compared to setTimer0(1);the LED will blink slower, but
   still much faster than the original 10 seconds, blinking
    every 100ms.
- The LED will blink at a rate that is still fast but can
   be observed with the human eye (blinking every 100ms),
   unlike the case with setTimer0(1); where it blinks too
   fast to observe.
```

1.7 Exercise 7

Upgrade the source code in Exercise 5 (update values for hour, minute and second) by using the software timer and remove the HAL_Delay function at the end. Moreover, the DOT (connected to PA4) of the digital clock is also moved to main function.

Report 1: Present your source code in the while loop on main function.

```
    while (1)
      {

          if (timer0_flag == 1) {
                second++;
            HAL_GPIO_TogglePin(DOT_GPIO_Port, DOT_Pin);
                if (second >= 60) {
                    second = 0;
                    minute++;
                    if (minute >= 60) {
                        minute = 0;
                        hour++;
                        if (hour >= 24) {
                            hour = 0;
                        }
                    }
                }
                updateClockBuffer(hour, minute);
                setTimer0(1000);
            }

      }
  }
```

Program 2.14: Source code from EX7

1.8 Exercise 8

Move also the update7SEG() function from the interrupt timer to the main. Finally, the timer interrupt only used to handle software timers. All processing (or complex computations) is move to an infinite loop on the main function, optimizing the complexity of the interrupt handler function.
Report 1: Present your source code in the the main function. In the case more extra functions are used (e.g. the second software timer), present them in the report as well.

```c
#ifndef INC_TIMER_H_
#define INC_TIMER_H_
    extern int timer0_counter;
    extern int timer0_flag;
    extern int TIMER_CYCLE ;
    extern int timer1_counter;
    extern int timer1_flag;
    extern int timer2_counter;
    extern int timer2_flag;
    extern int timer3_counter;
    extern int timer3_flag;
    void setTimer0(int duration);
    void setTimer1(int duration);
    void setTimer2(int duration);
    void setTimer3(int duration);
    void timer_run();
#endif /* INC_TIMER_H_ */
```

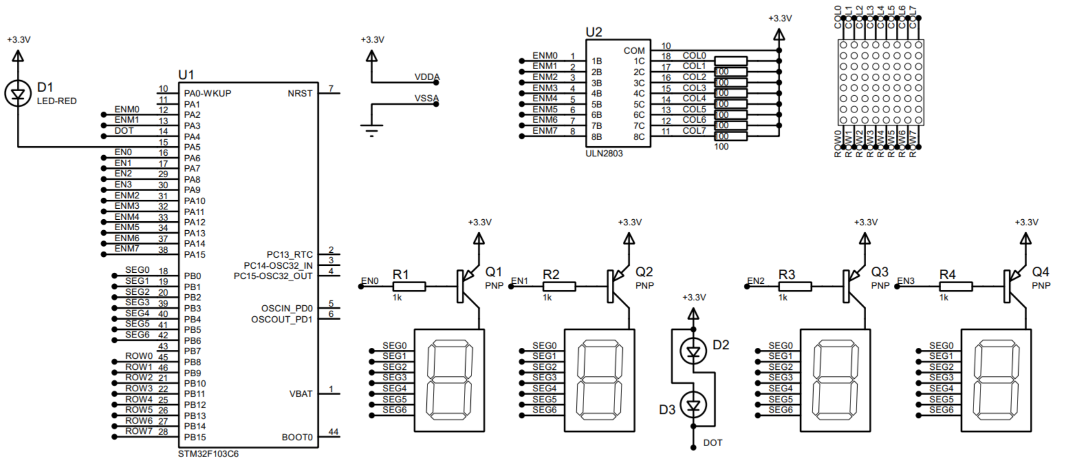
Program 2.15: Extra funtions in timer.h

```c
int hour = 15, minute = 8, second = 50;
int index=0;
    setTimer0(1000);
    setTimer1(3000);
    setTimer2(1000);
    index_led=0;
  while (1)
    {
    if(timer0_flag==1){
          HAL_GPIO_TogglePin(DOT_GPIO_Port, DOT_Pin);
          setTimer0(1000);
    }
    if(timer1_flag==1){
       switch (index){
              case 0:
            display7SEG(led_buffer[0]);
           //EN0 ENABLE
HAL_GPIO_WritePin(EN0_GPIO_Port,EN0_Pin,GPIO_PIN_RESET);
    HAL_GPIO_WritePin(EN1_GPIO_Port,EN1_Pin,GPIO_PIN_SET);
    HAL_GPIO_WritePin(EN2_GPIO_Port,EN2_Pin,GPIO_PIN_SET);
    HAL_GPIO_WritePin(EN3_GPIO_Port,EN3_Pin,GPIO_PIN_SET);
    break;
        case 1:
            display7SEG(led_buffer[1]);
            //EN1 ENABLE
    HAL_GPIO_WritePin(EN0_GPIO_Port,EN0_Pin,GPIO_PIN_SET);
HAL_GPIO_WritePin(EN1_GPIO_Port,EN1_Pin,GPIO_PIN_RESET);
    HAL_GPIO_WritePin(EN2_GPIO_Port,EN2_Pin,GPIO_PIN_SET);
    HAL_GPIO_WritePin(EN3_GPIO_Port,EN3_Pin,GPIO_PIN_SET);
    break;
```

```
        case 2:
            display7SEG(led_buffer[2]);
            //EN2 ENABLE
    HAL_GPIO_WritePin(EN0_GPIO_Port,EN0_Pin,GPIO_PIN_SET);
    HAL_GPIO_WritePin(EN1_GPIO_Port,EN1_Pin,GPIO_PIN_SET);
 HAL_GPIO_WritePin(EN2_GPIO_Port,EN2_Pin,GPIO_PIN_RESET);
    HAL_GPIO_WritePin(EN3_GPIO_Port,EN3_Pin,GPIO_PIN_SET);
    break;
        case 3:
            display7SEG(led_buffer[3]);
            //EN3 ENABLE
    HAL_GPIO_WritePin(EN0_GPIO_Port,EN0_Pin,GPIO_PIN_SET);
    HAL_GPIO_WritePin(EN1_GPIO_Port,EN1_Pin,GPIO_PIN_SET);
    HAL_GPIO_WritePin(EN2_GPIO_Port,EN2_Pin,GPIO_PIN_SET);
HAL_GPIO_WritePin(EN3_GPIO_Port,EN3_Pin,GPIO_PIN_RESET);
    break;
        default:
            break;
          }
        index++;
        if(index >=4)index=0;
          setTimer1(3000);
    }
    if(timer2_flag==1){
       second++;
       if (second >= 60) {
      second = 0;
      minute++;
      if (minute >= 60) {
        minute = 0;
        hour++;
        if (hour >= 24) {
          hour = 0;
             }
          }
         }
       updateClockBuffer(hour, minute);
       setTimer2(1000);
  }

    }
```
Program 2.16: Source code for EX8

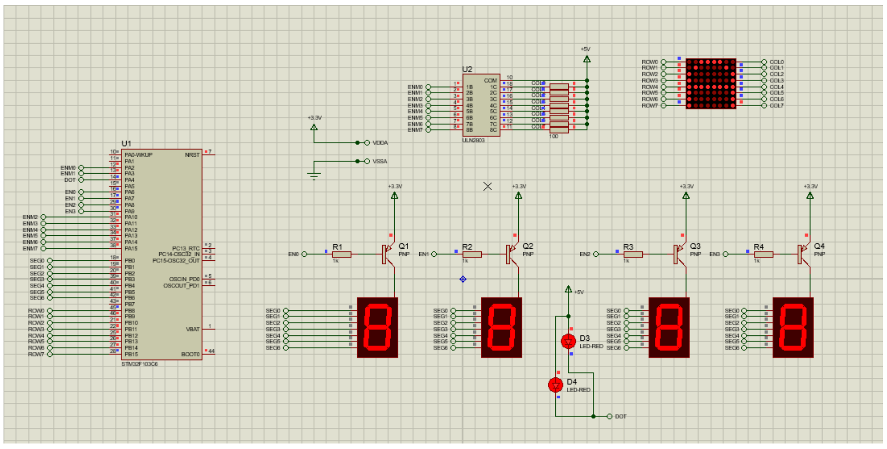
1.9 Exercise 9

This is an extra works for this lab. A LED Matrix is added to the system. A reference design is shown in figure bellow:


Hình 2.4: LED matrix is added to the simulation

In this schematic, two new components are added, including the **MATRIX-8X8-RED** and **ULN2803**, which is an NPN transistor array to enable the power supply for a column of the LED matrix. Students can change the enable signal (from ENM0 to ENM7) if needed. Finally, the data signal (from ROW0 to ROW7) is connected to PB8 to PB15.

Report 1: Present the schematic of your system by capturing the screen in Proteus.


Hình 2.5: Schematic from Proteus

Report 2: Implement the function, updateLEDMatrix(int index), which is similarly to 4 seven led segments.

```

const int MAX_LED_MATRIX = 8;
int index_led_matrix = 0;
// M ng  c h a  d   l i u  h i n  t h   c h  "A"
uint8_t matrix_buffer[8] = {
    0x3C,  // 00111100
    0x42,  // 01000010
    0x81,  // 10000001
    0x81,  // 10000001
    0xFF,  // 01000010
  0x81,  // 01000010
  0x81,  // 01000010
  0x81,  // 01000010
};
//uint8_t matrix_buffer[8] = {
//     0x3C,  // 00111100
//   0x3C,  // 01000010
//   0x3C,  // 10000001
//   0x3C,  // 10000001
//   0x3C,  // 01000010
//   0xFF,  // 01000010
//   0x3C,  // 01000010
//   0x18,  // 01000010
//};
void updateLEDMatrix(int index) {
    // Turn off all ENM pins before updating the current column
    HAL_GPIO_WritePin(ENM0_GPIO_Port, ENM0_Pin, GPIO_PIN_SET);
    HAL_GPIO_WritePin(ENM1_GPIO_Port, ENM1_Pin, GPIO_PIN_SET);
    HAL_GPIO_WritePin(ENM2_GPIO_Port, ENM2_Pin, GPIO_PIN_SET);
    HAL_GPIO_WritePin(ENM3_GPIO_Port, ENM3_Pin, GPIO_PIN_SET);
    HAL_GPIO_WritePin(ENM4_GPIO_Port, ENM4_Pin, GPIO_PIN_SET);
    HAL_GPIO_WritePin(ENM5_GPIO_Port, ENM5_Pin, GPIO_PIN_SET);
    HAL_GPIO_WritePin(ENM6_GPIO_Port, ENM6_Pin, GPIO_PIN_SET);
    HAL_GPIO_WritePin(ENM7_GPIO_Port, ENM7_Pin, GPIO_PIN_SET);

    // Activate the current column by setting the corresponding ENM pin to LOW
    switch (index) {
    case 0:
        HAL_GPIO_WritePin(ENM0_GPIO_Port, ENM0_Pin, GPIO_PIN_RESET);
        break;
    case 1:
        HAL_GPIO_WritePin(ENM1_GPIO_Port, ENM1_Pin, GPIO_PIN_RESET);
        break;
    case 2:
        HAL_GPIO_WritePin(ENM2_GPIO_Port, ENM2_Pin, GPIO_PIN_RESET);
        break;
    case 3:
        HAL_GPIO_WritePin(ENM3_GPIO_Port, ENM3_Pin, GPIO_PIN_RESET);
        break;
    case 4:
        HAL_GPIO_WritePin(ENM4_GPIO_Port, ENM4_Pin, GPIO_PIN_RESET);
        break;
    case 5:
        HAL_GPIO_WritePin(ENM5_GPIO_Port, ENM5_Pin, GPIO_PIN_RESET);
        break;
    case 6:
        HAL_GPIO_WritePin(ENM6_GPIO_Port, ENM6_Pin, GPIO_PIN_RESET);
        break;
    case 7:
        HAL_GPIO_WritePin(ENM7_GPIO_Port, ENM7_Pin, GPIO_PIN_RESET);
        break;
    default:
        break;
    }

    HAL_GPIO_WritePin(ROW0_GPIO_Port, ROW0_Pin, (matrix_buffer[index] & 0x01) ? GPIO_PIN_RESET : GPIO_PIN_SET);
    HAL_GPIO_WritePin(ROW1_GPIO_Port, ROW1_Pin, (matrix_buffer[index] & 0x02) ? GPIO_PIN_RESET : GPIO_PIN_SET);
    HAL_GPIO_WritePin(ROW2_GPIO_Port, ROW2_Pin, (matrix_buffer[index] & 0x04) ? GPIO_PIN_RESET : GPIO_PIN_SET);
    HAL_GPIO_WritePin(ROW3_GPIO_Port, ROW3_Pin, (matrix_buffer[index] & 0x08) ? GPIO_PIN_RESET : GPIO_PIN_SET);
    HAL_GPIO_WritePin(ROW4_GPIO_Port, ROW4_Pin, (matrix_buffer[index] & 0x10) ? GPIO_PIN_RESET : GPIO_PIN_SET);
    HAL_GPIO_WritePin(ROW5_GPIO_Port, ROW5_Pin, (matrix_buffer[index] & 0x20) ? GPIO_PIN_RESET : GPIO_PIN_SET);
    HAL_GPIO_WritePin(ROW6_GPIO_Port, ROW6_Pin, (matrix_buffer[index] & 0x40) ? GPIO_PIN_RESET : GPIO_PIN_SET);
    HAL_GPIO_WritePin(ROW7_GPIO_Port, ROW7_Pin, (matrix_buffer[index] & 0x80) ? GPIO_PIN_RESET : GPIO_PIN_SET);
}

//Additional Functions
//Timer 0
int timer0_counter = 0;
int timer0_flag = 0;
int TIMER_CYCLE = 10;


void setTimer0(int duration) {
    timer0_counter = duration / TIMER_CYCLE;
    timer0_flag = 0;
}

//Timer 1
int timer1_counter = 0;
int timer1_flag = 0;


void setTimer1(int duration)
{
  timer1_counter = duration / TIMER_CYCLE;
  timer1_flag = 0;
}

//Timer 2
int timer2_counter = 0;
int timer2_flag = 0;
```

```c
void setTimer2(int duration)
{
  timer2_counter = duration / TIMER_CYCLE;
  timer2_flag = 0;
}

int timer3_counter = 0;
int timer3_flag = 0;
void setTimer3(int duration)
{
  timer3_counter = duration / TIMER_CYCLE;
  timer3_flag = 0;
}


void timer_run()
{
    if (timer0_counter > 0) {
        timer0_counter--;  // Decrement the counter
        if (timer0_counter == 0) {
            timer0_flag = 1;  // Set the flag when the counter reaches 0
        }
    }

    if (timer1_counter > 0) {
           timer1_counter--;  // Decrement the counter
           if (timer1_counter == 0) {
               timer1_flag = 1;  // Set the flag when the counter reaches 0
           }
       }

    if (timer2_counter > 0) {
            timer2_counter--;  // Decrement the counter
            if (timer2_counter == 0) {
                timer2_flag = 1;  // Set the flag when the counter reaches 0
            }
        }
    if (timer3_counter > 0) {
               timer3_counter--;  // Decrement the counter
               if (timer3_counter == 0) {
                   timer3_flag = 1;  // Set the flag when the counter reaches 0
               }
           }
}

/* USER CODE END 0 */

void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  timer_run();
}
int main(void)
{
  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_TIM2_Init();
  /* USER CODE BEGIN 2 */
  HAL_TIM_Base_Start_IT (& htim2 ) ;
  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  int hour = 15, minute = 8, second = 50;
  setTimer0(1000);  // Set the timer for Dot (1 second)
  setTimer1(3000);  // Set the timer for Segment leds
  setTimer2(1000);  //Set the timer for updateClockBuffer
  setTimer3(50);  //EX10
  int index = 0;
while (1)
{
      if (timer0_flag == 1)
      {
        second++;
        HAL_GPIO_TogglePin(DOT_GPIO_Port, DOT_Pin);
          setTimer0(1000);  // Set the timer for 1000 ms (1 seconds)
      }

      if (timer1_flag == 1)
      {
        switch (index)
          {
            case 0:
              //Display led 1
              display_7SEG(led_buffer[0]);

                    HAL_GPIO_WritePin(EN0_GPIO_Port, EN0_Pin, GPIO_PIN_RESET);
                    HAL_GPIO_WritePin(EN1_GPIO_Port, EN1_Pin, GPIO_PIN_SET);
                    HAL_GPIO_WritePin(EN2_GPIO_Port, EN2_Pin, GPIO_PIN_SET);
                    HAL_GPIO_WritePin(EN3_GPIO_Port, EN3_Pin, GPIO_PIN_SET);
                    break;

            case 1:
              //Display led 2
              display_7SEG(led_buffer[1]);

                    HAL_GPIO_WritePin(EN0_GPIO_Port, EN0_Pin, GPIO_PIN_SET);
                    HAL_GPIO_WritePin(EN1_GPIO_Port, EN1_Pin, GPIO_PIN_RESET);
                    HAL_GPIO_WritePin(EN2_GPIO_Port, EN2_Pin, GPIO_PIN_SET);
                    HAL_GPIO_WritePin(EN3_GPIO_Port, EN3_Pin, GPIO_PIN_SET);
                    break;

            case 2:
               //Display led 3
               display_7SEG(led_buffer[2]);

                    HAL_GPIO_WritePin(EN0_GPIO_Port, EN0_Pin, GPIO_PIN_SET);
                    HAL_GPIO_WritePin(EN1_GPIO_Port, EN1_Pin, GPIO_PIN_SET);
                    HAL_GPIO_WritePin(EN2_GPIO_Port, EN2_Pin, GPIO_PIN_RESET);
                    HAL_GPIO_WritePin(EN3_GPIO_Port, EN3_Pin, GPIO_PIN_SET);
                    break;

            case 3:
              //Display led 4
              display_7SEG(led_buffer[3]);

                    HAL_GPIO_WritePin(EN0_GPIO_Port, EN0_Pin, GPIO_PIN_SET);
                    HAL_GPIO_WritePin(EN1_GPIO_Port, EN1_Pin, GPIO_PIN_SET);
                    HAL_GPIO_WritePin(EN2_GPIO_Port, EN2_Pin, GPIO_PIN_SET);
                    HAL_GPIO_WritePin(EN3_GPIO_Port, EN3_Pin, GPIO_PIN_RESET);
                    break;

            default:
              break;
            }
          index++;
          if(index >= 4)
          {
            index = 0;
          }
          setTimer1(3000);
      }

      if(timer2_flag == 1)
      {
        if(second >= 60)
        {
            minute++;
            second = 0;
        }
        if(minute >= 60)
        {
          hour++;
          minute = 0;
        }
          if (hour >= 24)
          {
            hour = 0;
          }
          updateClockBuffer(minute, hour);
          setTimer2(1000);
      }

    if (timer3_flag == 1)
          {
            updateLEDMatrix(index_led_matrix);
            index_led_matrix++;
          if (index_led_matrix >= 8)
          {
            shiftLeft(matrix_buffer);
            index_led_matrix = 0;
          }
              setTimer3(50);
          }
}
```

Program 2.17: Function to display data on LED Matrix

1.10 Exercise 10

Create an animation on LED matrix, for example, the character is shifted to the left.

Report 1: Briefly describe your solution and present your source code in the report.

```c
void shiftLeft(uint8_t matrix_buffer[8])
{
  for (int i = 0; i < 8; i++) {
    //  L y  bit ngo i  c ng  b n  tr i (bit  t h   7)
    uint8_t leftBit = (matrix_buffer[i] & 0x80) >> 7;

    //  D ch  tr i  to n  b   h ng  v   n i  bit ngo i c ng  b n  tr i  v o  b n  p h i
    matrix_buffer[i] = (matrix_buffer[i] << 1) | leftBit;
  }
}
```

Program 2.18: Shift left function

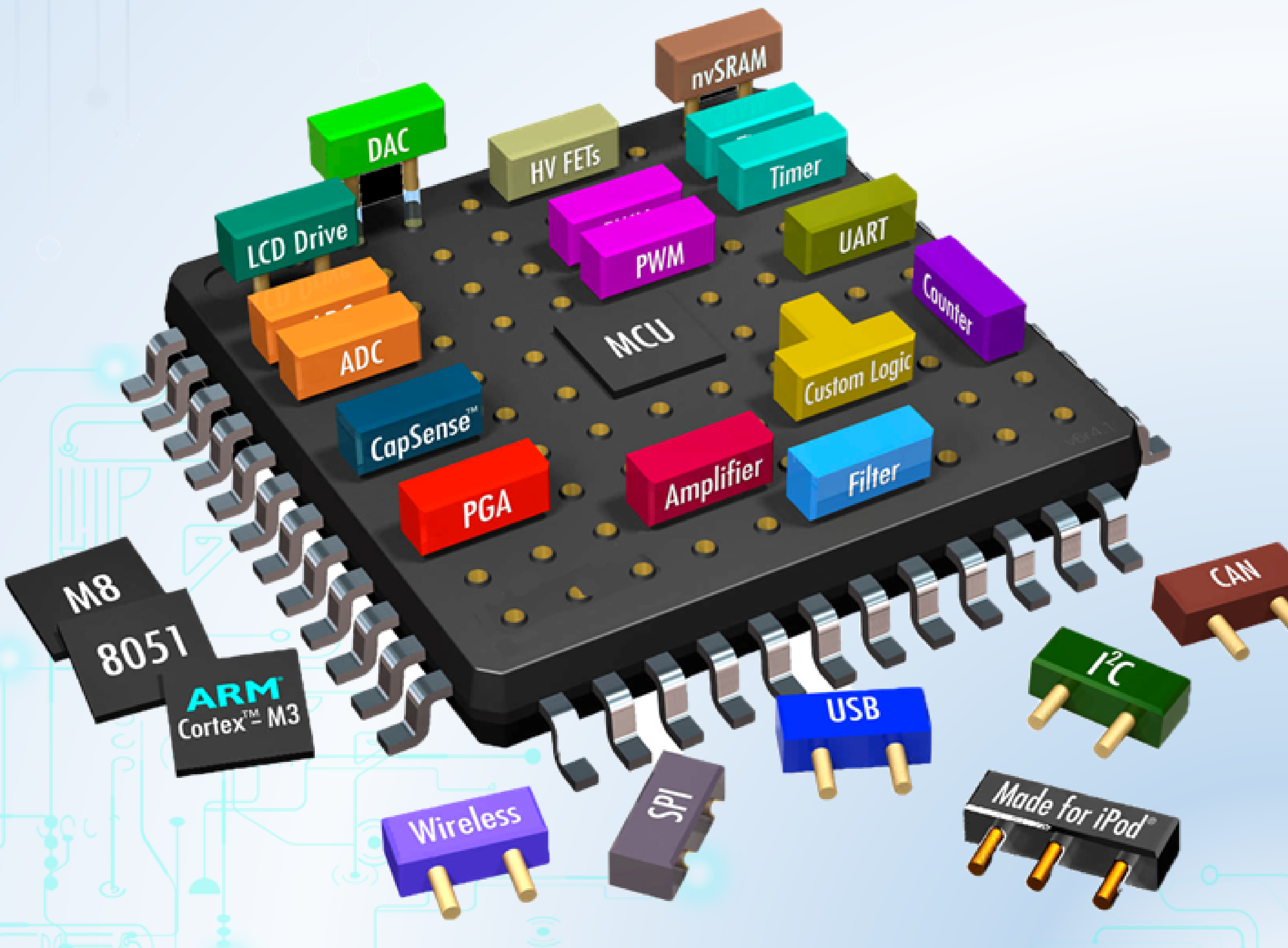
```c
int timer3_counter = 0;
int timer3_flag = 0;
void setTimer3(int duration)
{
  timer3_counter = duration / TIMER_CYCLE;
  timer3_flag = 0;
}


void timer_run()
{
    if (timer0_counter > 0) {
        timer0_counter--;  // Decrement the counter
        if (timer0_counter == 0) {
            timer0_flag = 1;  // Set the flag when the counter reaches 0
        }
    }

    if (timer1_counter > 0) {
           timer1_counter--;  // Decrement the counter
           if (timer1_counter == 0) {
               timer1_flag = 1;  // Set the flag when the counter reaches 0
           }
       }

    if (timer2_counter > 0) {
            timer2_counter--;  // Decrement the counter
            if (timer2_counter == 0) {
                timer2_flag = 1;  // Set the flag when the counter reaches 0
            }
        }
    if (timer3_counter > 0) {
               timer3_counter--;  // Decrement the counter
```

```
                    if (timer3_counter == 0) {
                        timer3_flag = 1;  // Set the flag when
   the counter reaches 0
                    }
               }
}

void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  timer_run();
}

//Function in while
while (1)
{
     if (timer3_flag == 1)
             {
               updateLEDMatrix(index_led_matrix);
               index_led_matrix++;
             if (index_led_matrix >= 8)
             {
               shiftLeft(matrix_buffer);
               index_led_matrix = 0;
             }
                 setTimer3(50);
             }
}
```

Program 2.19: Solution to display animation in main.c

CHƯƠNG 3

Buttons/Switches



1 Objectives

In this lab, you will

- Learn how to add new C source files and C header files in an STM32 project,
- Learn how to read digital inputs and display values to LEDs using a timer interrupt of a microcontroller (MCU).
- Learn how to debounce when reading a button.
- Learn how to create an FSM and implement an FSM in an MCU.

2 Introduction

Embedded systems usually use buttons (or keys, or switches, or any form of mechanical contacts) as part of their user interface. This general rule applies from the most basic remote-control system for opening a garage door, right up to the most sophisticated aircraft autopilot system. Whatever the system you create, you need to be able to create a reliable button interface.

A button is generally hooked up to an MCU so as to generate a certain logic level when pushed or closed or “active" and the opposite logic level when unpushed or open or “inactive." The active logic level can be either ‘0’ or ‘1’, but for reasons both historical and electrical, an active level of ’0’ is more common.

We can use a button if we want to perform operations such as:

- Drive a motor while a switch is pressed.
- Switch on a light while a switch is pressed.
- Activate a pump while a switch is pressed.

These operations could be implemented using an electrical button without using an MCU; however, use of an MCU may well be appropriate if we require more complex behaviours. For example:

- Drive a motor while a switch is pressed.

  Condition: If the safety guard is not in place, don’t turn the motor. Instead sound a buzzer for 2 seconds.
- Switch on a light while a switch is pressed.

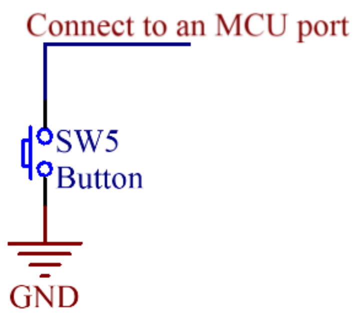
  Condition: To save power, ignore requests to turn on the light during daylight hours.
- Activate a pump while a switch is pressed

  Condition: If the main water reservoir is below 300 litres, do not start the main pump: instead, start the reserve pump and draw the water from the emergency tank.

In this lab, we consider how you read inputs from mechanical buttons in your embedded application using an MCU.

3 Basic techniques for reading from port pins

3.1 The need for pull-up resistors



Hình 3.1: *Connecting a button to an MCU*

Figure 6.2 shows a way to connect a button to an MCU. This hardware operates as follows:

- When the switch is open, it has no impact on the port pin. An internal resistor on the port “pulls up" the pin to the supply voltage of the MCU (typically 3.3V for STM32F103). If we read the pin, we will see the value ‘1’.
- When the switch is closed (pressed), the pin voltage will be 0V. If we read the pin, we will see the value ‘0’.

However, if the MCU does not have a pull-up resistor inside, when the button is pressed, the read value will be ‘0’, but even we release the button, the read value is still ‘0’ as shown in Figure 3.2.

So a reliable way to connect a button/switch to an MCU is that we explicitly use an external pull-up resistor as shown in Figure 3.3.

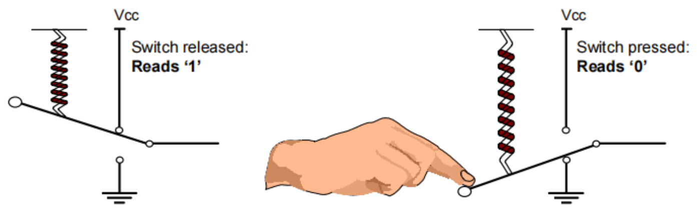
3.2 Dealing with switch bounces

In practice, all mechanical switch contacts bounce (that is, turn on and off, repeatedly, for short period of time) after the switch is closed or opened as shown in Figure 3.4.

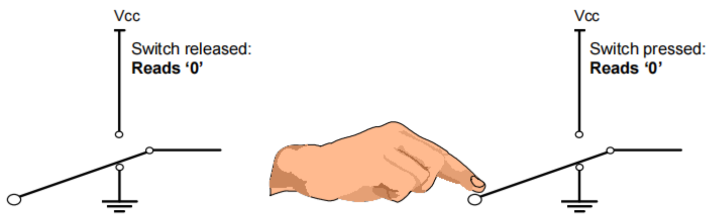
Every system that uses any kind of mechanical switch must deal with the issue of debouncing. The key task is to make sure that one mechanical switch or button action is only read as one action by the MCU, even though the MCU will typically be fast enough to detect the unwanted switch bounces and treat them as separate events. Bouncing can be eliminated by special ICs or by RC circuitry, but in most cases debouncing is done in software because software is “free".

As far as the MCU concerns, each “bounce" is equivalent to one press and release of an “ideal” switch. Without appropriate software design, this can give several problems:

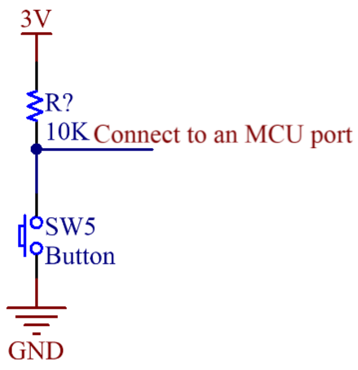
With pull-ups:



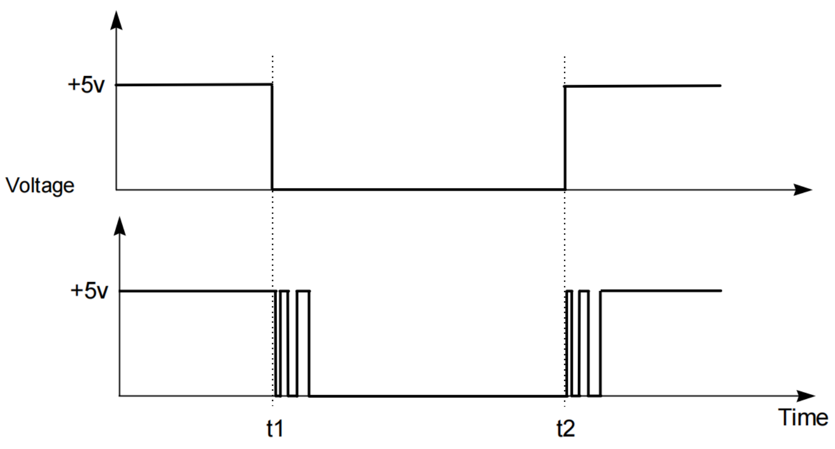
Without pull-ups:



Hình 3.2: The need of pull up resistors



Hình 3.3: A reliable way to connect a button to an MCU



Hình 3.4: Switch bounces

- Rather than reading ‘A’ from a keypad, we may read ‘AAAAA’
- Counting the number of times that a switch is pressed becomes extremely difficult
- If a switch is depressed once, and then released some time later, the ‘bounce’ may make it appear as if the switch has been pressed again (at the time of release).

The key to debouncing is to establish a minimum criterion for a valid button push, one that can be implemented in software. This criterion must involve differences in time - two button presses in 20ms must be treated as one button event, while two button presses in 2 seconds must be treated as two button events. So what are the relevant times we need to consider? They are these:

- Bounce time: most buttons seem to stop bouncing within 10ms
- Button press time: the shortest time a user can press and release a button seems to be between 50 and 100ms
- Response time: a user notices if the system response is 100ms after the button press, but not if it is 50ms after

Combining all of these times, we can set a few goals

- Ignore all bouncing within 10ms
- Provide a response within 50ms of detecting a button push (or release)
- Be able to detect a 50ms push and a 50ms release

The simplest debouncing method is to examine the keys (or buttons or switches) every N milliseconds, where N > 10ms (our specified button bounce upper limit) and N <= 50ms (our specified response time). We then have three possible outcomes every time we read a button:

- We read the button in the solid ‘0’ state
- We read the button in the solid ‘1’ state
- We read the button while it is bouncing (so we will get either a ‘0’ or a ‘1’)

Outcomes 1 and 2 pose no problems, as they are what we would always like to happen. Outcome 3 also poses no problem because during a bounce either state is acceptable. If we have just pressed an active-low button and we read a ’1’ as it bounces, the next time through we are guaranteed to read a ’0’ (remember, the next time through all bouncing will have ceased), so we will just detect the button push a bit later. Otherwise, if we read a ’0’ as the button bounces, it will still be ’0’ the next time after all bouncing has stopped, so we are just detecting the button push a bit earlier. The same applies to releasing a button. Reading a single bounce (with all bouncing over by the time of the next read) will never give us an invalid button state. It’s only reading multiple bounces (multiple reads while bouncing is occurring) that can give invalid button states such as repeated push signals from one physical push.

So if we guarantee that all bouncing is done by the time we next read the button, we're good. Well, almost good, if we're lucky...

MCUs often live among high-energy beasts, and often control the beasts. High energy devices make electrical noise, sometimes great amounts of electrical noise. This noise can, at the worst possible moment, get into your delicate button-and-high-value-pullup circuit and act like a real button push. Oops, missile launched, sorry!

If the noise is too intense we cannot filter it out using only software, but will need hardware of some sort (or even a redesign). But if the noise is only occasional, we can filter it out in software without too much bother. The trick is that instead of regarding a single button 'make' or 'break' as valid, we insist on N contiguous makes or breaks to mark a valid button event. N will be a factor of your button scanning rate and the amount of filtering you want to add. Bigger N gives more filtering. The simplest filter (but still a big improvement over no filtering) is just an N of 2, which means compare the current button state with the last button state, and only if both are the same is the output valid.

Note that now we have not two but three button states: active (or pressed), inactive (or released), and indeterminate or invalid (in the middle of filtering, not yet filtered). In most cases we can treat the invalid state the same as the inactive state, since we care in most cases only about when we go active (from whatever state) and when we cease being active (to inactive or invalid). With that simplification we can look at simple N = 2 filtering reading a button wired to STM32 MCU:

```c
void button_reading(void){
    static unsigned char last_button;
    unsigned char raw_button;
    unsigned char filtered_button;
    last_button = raw_button;
    raw_button = HAL_GPIO_ReadPin(BUTTON_1_GPIO_Port,
   BUTTON_1_Pin);
    if(last_button == raw_button){
        filtered_button = raw_button;
    }
}
```

Program 3.1: Read port pin and deboucing

The function button_reading() must be called no more often than our debounce time (10ms).

To expand to greater filtering (larger N), keep in mind that the filtering technique essentially involves reading the current button state and then either counting or reseting the counter. We count if the current button state is the same as the last button state, and if our count reaches N we then report a valid new button state. We reset the counter if the current button state is different than the last button state, and we then save the current button state as the new button state to compare against the next time. Also note that the larger our value of N the more often our filtering routine must be called, so that we get a filtered response within our specified 50ms deadline. So for example with an N of 8 we should be calling our filtering routine every 2 - 5ms, giving a response time of 16 - 40ms (>10ms and <50ms).

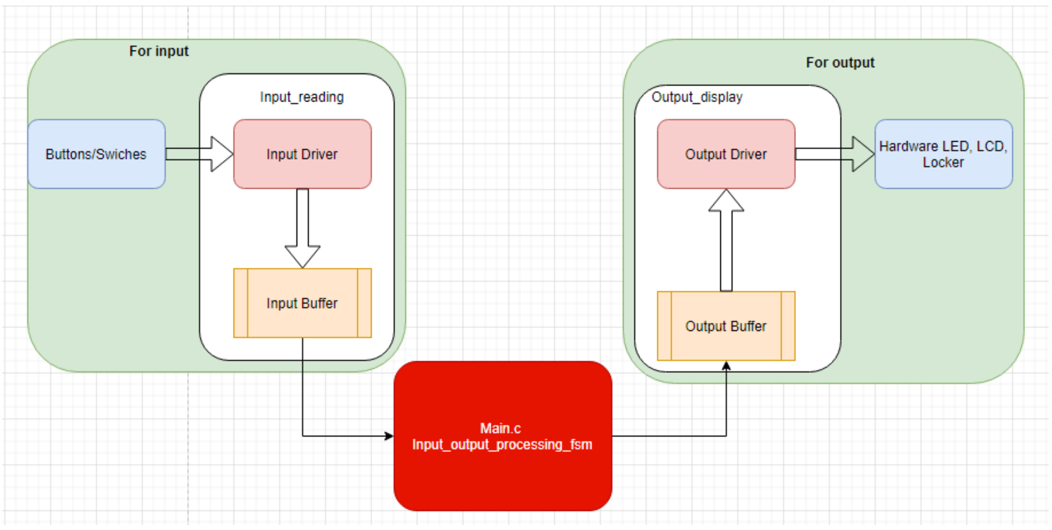
4 Reading switch input (basic code) using STM32

To demonstrate the use of buttons/switches in STM32, we use an example which requires to write a program that

- Has a timer which has an interrupt in every 10 milliseconds.
- Reads values of button PB0 every 10 milliseconds.
- Increases the value of LEDs connected to PORTA by one unit when the button PB0 is pressed.
- Increases the value of PORTA automatically in every 0.5 second, if the button PB0 is pressed in more than 1 second.

4.1 Input Output Processing Patterns

For both input and output processing, we have a similar pattern to work with. Normally, we have a module named driver which works directly to the hardware. We also have a buffer to store temporarily values. In the case of input processing, the driver will store the value of the hardware status to the buffer for further processing. In the case of output processing, the driver uses the buffer data to output to the hardware.



Hình 3.5: Input Output Processing Patterns

Figure 3.5 shows that we should have an *input_reading* module to processing the buttons, then store the processed data to the buffer. Then a module of *input_output_processin* will process the input data, and update the output buffer. The output driver gets the value from the output buffer to transfer to the hardware.

4.2 Setting up

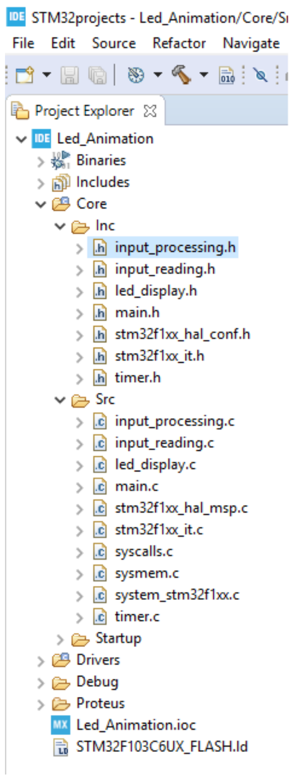
4.2.1 Create a project

Please follow the instruction in Labs 1 and 2 to create a project that includes:

- PB0 as an input port pin,
- PA0-PA7 as output port pins, and
- Timer 2 10ms interrupt

4.2.2 Create a file C source file and header file for input reading

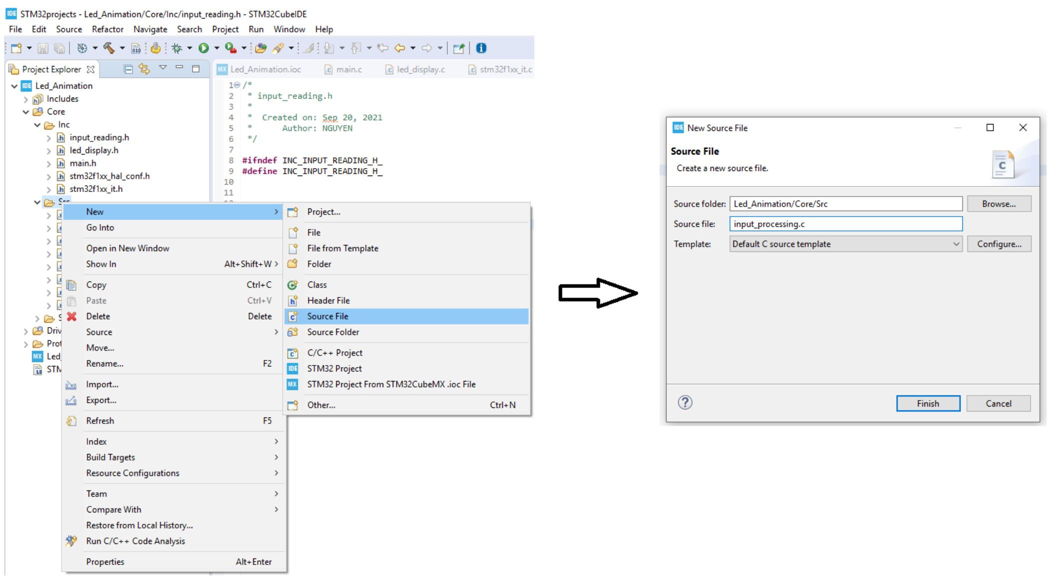
We are expected to have files for button processing and led display as shown in Figure 3.6.



Hình 3.6: File Organization

Steps 1 (Figure 3.7): Right click to the folder **Src**, select **New**, then select **Source File**. There will be a pop-up. Please type the file name, then click **Finish**.

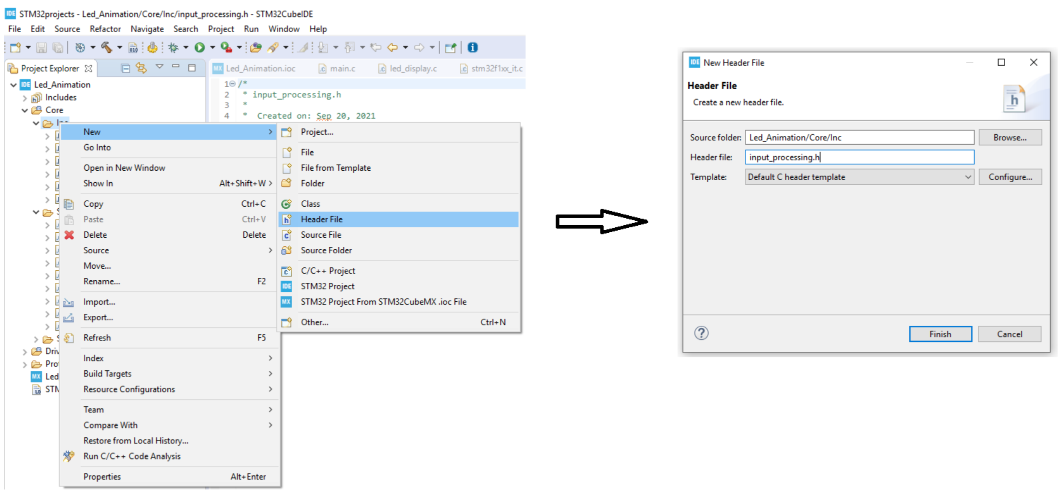
Step 2 (Figure 3.8): Do the same for the C header file in the folder **Inc**.



```
/*
 * input_reading.h
 *
 *  Created on: Sep 20, 2021
 *      Author: NGUYEN
 */

#ifndef INC_INPUT_READING_H_
#define INC_INPUT_READING_H_
```

Source File
Create a new source file.
Source folder: Led_Animation/Core/Src
Source file: input_processing.c
Template: Default C source template

Hình 3.7: Step 1: Create a C source file for input reading



```
/*
 * input_processing.h
 *
 *  Created on: Sep 20, 2021
```

Header File
Create a new header file.
Source folder: Led_Animation/Core/Inc
Header file: input_processing.h
Template: Default C header template

Hình 3.8: Step 2: Create a C header file for input processing

4.3 Code For Read Port Pin and Debouncing

4.3.1 The code in the input_reading.c file

```c
#include "main.h"
//we aim to work with more than one buttons
#define NO_OF_BUTTONS                1
//timer interrupt duration is 10ms, so to pass 1 second,
//we need to jump to the interrupt service routine 100 time
#define DURATION_FOR_AUTO_INCREASING     100
#define BUTTON_IS_PRESSED                  GPIO_PIN_RESET
#define BUTTON_IS_RELEASED                 GPIO_PIN_SET
//the buffer that the final result is stored after
//debouncing
static GPIO_PinState buttonBuffer[NO_OF_BUTTONS];
//we define two buffers for debouncing
static GPIO_PinState debounceButtonBuffer1[NO_OF_BUTTONS];
static GPIO_PinState debounceButtonBuffer2[NO_OF_BUTTONS];
//we define a flag for a button pressed more than 1 second.
static uint8_t flagForButtonPress1s[NO_OF_BUTTONS];
//we define counter for automatically increasing the value
//after the button is pressed more than 1 second.
static uint16_t counterForButtonPress1s[NO_OF_BUTTONS];
void button_reading(void){
  for(char i = 0; i < NO_OF_BUTTONS; i ++){
    debounceButtonBuffer2[i] =debounceButtonBuffer1[i];
    debounceButtonBuffer1[i] = HAL_GPIO_ReadPin(
   BUTTON_1_GPIO_Port, BUTTON_1_Pin);
    if(debounceButtonBuffer1[i] == debounceButtonBuffer2[i
   ])
      buttonBuffer[i] = debounceButtonBuffer1[i];
      if(buttonBuffer[i] == BUTTON_IS_PRESSED){
      //if a button is pressed, we start counting
        if(counterForButtonPress1s[i] <
   DURATION_FOR_AUTO_INCREASING){
          counterForButtonPress1s[i]++;
        } else {
        //the flag is turned on when 1 second has passed
        //since the button is pressed.
          flagForButtonPress1s[i] = 1;
          //todo
        }
      } else {
        counterForButtonPress1s[i] = 0;
        flagForButtonPress1s[i] = 0;
      }
  }
}
```

Program 3.2: Define constants buffers and button_reading function

```c
unsigned char is_button_pressed(uint8_t index){
  if(index >= NO_OF_BUTTONS) return 0;
  return (buttonBuffer[index] == BUTTON_IS_PRESSED);
}
```
Program 3.3: Checking a button is pressed or not

```c
unsigned char is_button_pressed_1s(unsigned char index){
  if(index >= NO_OF_BUTTONS) return 0xff;
  return (flagForButtonPress1s[index] == 1);
}
```
Program 3.4: Checking a button is pressed more than a second or not

4.3.2 The code in the input_reading.h file

```c
#ifndef INC_INPUT_READING_H_
#define INC_INPUT_READING_H_
void button_reading(void);
unsigned char is_button_pressed(unsigned char index);
unsigned char is_button_pressed_1s(unsigned char index);
#endif /* INC_INPUT_READING_H_ */
```
Program 3.5: Prototype in input_reading.h file

4.3.3 The code in the timer.c file

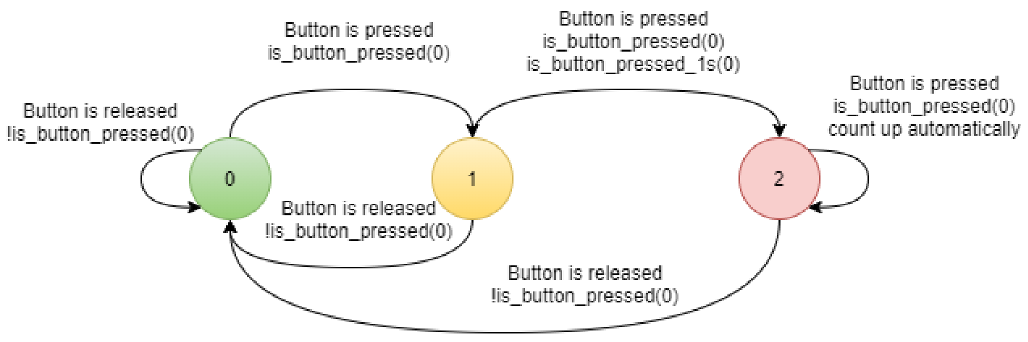
```c
#include "main.h"
#include "input_reading.h"

void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  if(htim->Instance == TIM2){
    button_reading();
  }
}
```
Program 3.6: Timer interrupt callback function

4.4 Button State Processing

4.4.1 Finite State Machine

To solve the example problem, we define 3 states as follows:

- State 0: The button is released or the button is in the initial state.
- State 1: When the button is pressed, the FSM will change to State 1 that is increasing the values of PORTA by one value. If the button is released, the FSM goes back to State 0.
- State 2: while the FSM is in State 1, the button is kept pressing more than 1 second, the state of FSM will change from 1 to 2. In this state, if the button is kept pressing, the value of PORTA will be increased automatically in every 500ms. If the button is released, the FSM goes back to State 0.



Hình 3.9: An FSM for processing a button

4.4.2 The code for the FSM in the input_processing.c file

Please note that *fsm_for_input_processing* function should be called inside the super loop of the main functin.

```c
#include "main.h"
#include "input_reading.h"

enum ButtonState{BUTTON_RELEASED, BUTTON_PRESSED,
   BUTTON_PRESSED_MORE_THAN_1_SECOND} ;
enum ButtonState buttonState = BUTTON_RELEASED;
void fsm_for_input_processing(void){
  switch(buttonState){
  case BUTTON_RELEASED:
    if(is_button_pressed(0)){
      buttonState = BUTTON_PRESSED;
      //INCREASE VALUE OF PORT A BY ONE UNIT
    }
    break;
  case BUTTON_PRESSED:
    if(!is_button_pressed(0)){
      buttonState = BUTTON_RELEASED;
    } else {
      if(is_button_pressed_1s(0)){
        buttonState = BUTTON_PRESSED_MORE_THAN_1_SECOND;
      }
    }
    break;
  case BUTTON_PRESSED_MORE_THAN_1_SECOND:
    if(!is_button_pressed(0)){
      buttonState = BUTTON_RELEASED;
    }
    //todo
    break;
  }
}
```

Program 3.7: The code in the input_processing.c file

4.4.3 The code in the input_processing.h

```c
#ifndef INC_INPUT_PROCESSING_H_
#define INC_INPUT_PROCESSING_H_

void fsm_for_input_processing(void);

#endif /* INC_INPUT_PROCESSING_H_ */
```

Program 3.8: Code in the input_processing.h file

4.4.4 The code in the main.c file

```c
#include "main.h"
#include "input_processing.h"
//don't modify this part
int main(void){
    HAL_Init();
    /* Configure the system clock */
    SystemClock_Config();
    /* Initialize all configured peripherals */
    MX_GPIO_Init();
    MX_TIM2_Init();
    while (1)
    {
        //you only need to add the fsm function here
        fsm_for_input_processing();
    }
}
```

Program 3.9: The code in the main.c file

5 Exercises and Report

5.1 Specifications

You are required to build an application of a traffic light in a cross road which includes some features as described below:

- The application has 12 LEDs including 4 red LEDs, 4 amber LEDs, 4 green LEDs.
- The application has 4 seven segment LEDs to display time with 2 for each road. The 2 seven segment LEDs will show time for each color LED corresponding to each road.
- The application has three buttons which are used

  - to select modes,

  - to modify the time for each color led on the fly, and

  - to set the chosen value.
- The application has at least 4 modes which is controlled by the first button. Mode 1 is a normal mode, while modes 2 3 4 are modification modes. You can press the first button to change the mode. Modes will change from 1 to 4 and back to 1 again.

  Mode 1 - Normal mode:

  - The traffic light application is running normally.

  Mode 2 - Modify time duration for the red LEDs: This mode allows you to change the time duration of the red LED in the main road. The expected behaviours of this mode include:

  - All single red LEDs are blinking in 2 Hz.

  - Use two seven-segment LEDs to display the value.

  - Use the other two seven-segment LEDs to display the mode.

  - The second button is used to increase the time duration value for the red LEDs.

  - The value of time duration is in a range of 1 - 99.

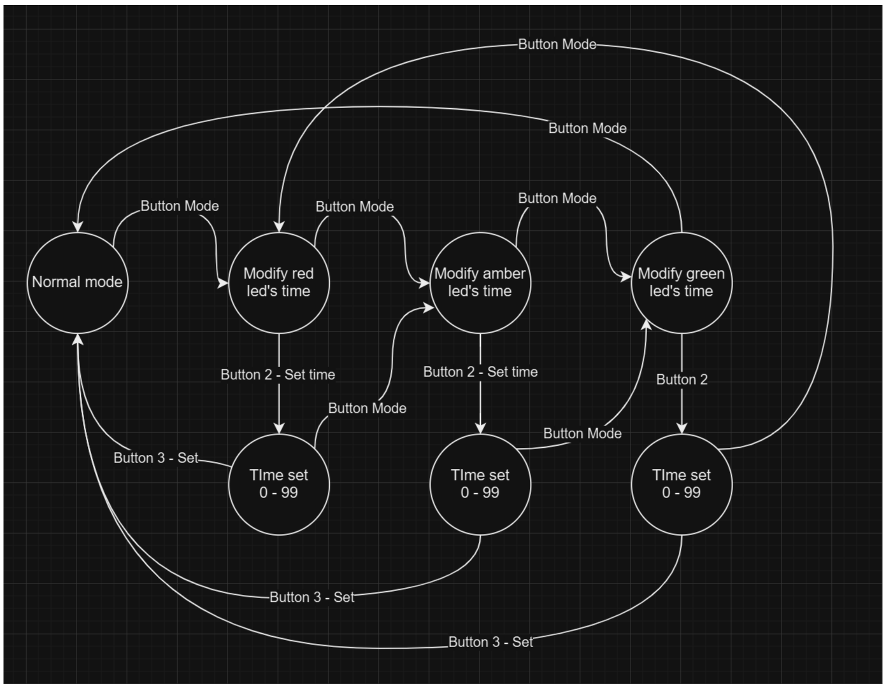
  - The third button is used to set the value.

  Mode 3 - Modify time duration for the amber LEDs: Similar for the red LEDs described above with the amber LEDs.

  Mode 4 - Modify time duration for the green LEDs: Similar for the red LEDs described above with the green LEDs.

5.2 Exercise 1: Sketch an FSM

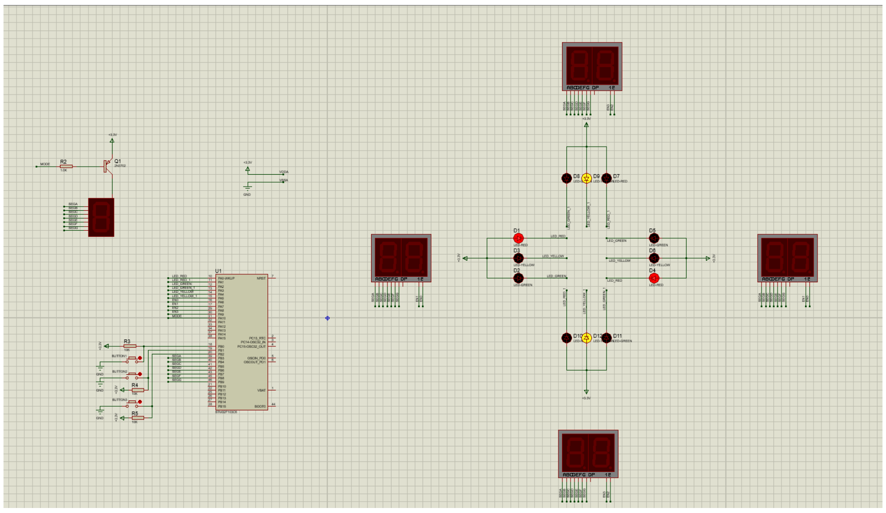
Your task in this exercise is to sketch an FSM that describes your idea of how to solve the problem. Please add your report here.



Hình 3.10: FSM

5.3 Exercise 2: Proteus Schematic

Your task in this exercise is to draw a Proteus schematic for the problem above. Please add your report here.

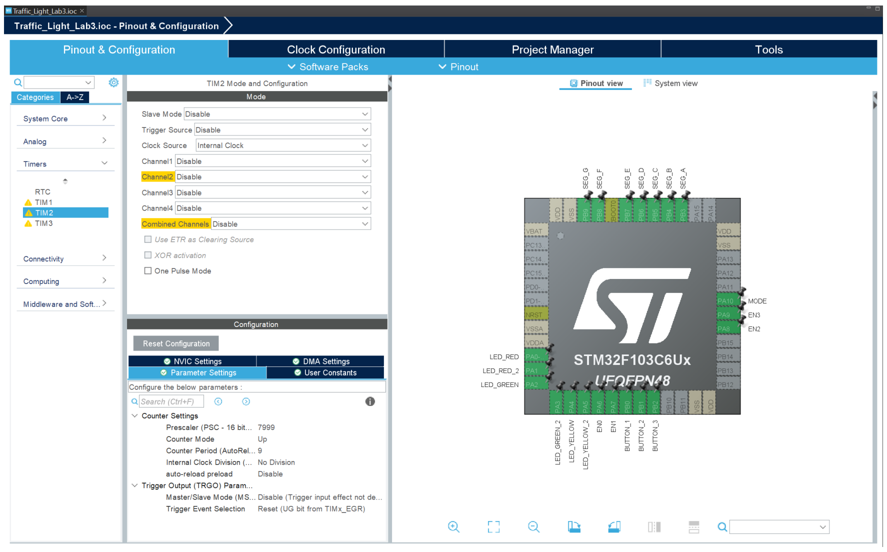


Hình 3.11: Schematic from Proteus

5.4 Exercise 3: Create STM32 Project

Your task in this exercise is to create a project that has pin corresponding to the Proteus schematic that you draw in previous section. You need to set up your timer interrupt is about 10ms.

Please add your report here.



TIM2 Mode and Configuration

Mode

Slave Mode: Disable
Trigger Source: Disable
Clock Source: Internal Clock
Channel1: Disable
Channel2: Disable
Channel3: Disable
Channel4: Disable
Combined Channels: Disable
Use ETR as Clearing Source
XOR activation
One Pulse Mode

Configuration

Counter Settings
Prescaler (PSC - 16 bit... 7999
Counter Mode: Up
Counter Period (AutoRel... 9
Internal Clock Division (... No Division
auto-reload preload: Disable
Trigger Output (TRGO) Param...
Master/Slave Mode (MS... Disable (Trigger input effect not de...
Trigger Event Selection: Reset (UG bit from TIMx_EGR)

Hình 3.12: IOC

5.5 Exercise 4: Modify Timer Parameters

Your task in this exercise is to modify the timer settings so that when we want to change the time duration of the timer interrupt, we change it the least and it will not affect the overall system. For example, the current system we have implemented is that it can blink an LED in 2 Hz, with the timer interrupt duration is 10ms. However, when we want to change the timer interrupt duration to 1ms or 100ms, it will not affect the 2Hz blinking LED.

Please add your report here.

```c
/*
 * timer.c
 *
 *  Created on: Oct 28, 2024
 *      Author: phihv
 */

#include "main.h"
#include "input_reading.h"
#include "timer.h"

// Initialize counter variables and flag indicators
int timer0_counter = 0;
int timer0_flag = 0;

int timer1_counter = 0;
int timer1_flag = 0;

int timer2_counter = 0;
int timer2_flag = 0;

int timer3_counter = 0;
int timer3_flag = 0;

// Define initialization functions for timers
void setTimer0(int duration) {
    timer0_counter = duration / TIMER_CYCLE;
    timer0_flag = 0;
}

void setTimer1(int duration) {
    timer1_counter = duration / TIMER_CYCLE;
    timer1_flag = 0;
}

void setTimer2(int duration) {
    timer2_counter = duration / TIMER_CYCLE;
    timer2_flag = 0;
}

void setTimer3(int duration) {
    timer3_counter = duration / TIMER_CYCLE;
    timer3_flag = 0;
}

// Function to run all counters, called in each interrupt cycle
void timer_run() {
    if (timer0_counter > 0) {
        timer0_counter--;
        if (timer0_counter == 0) timer0_flag = 1;
    }

    if (timer1_counter > 0) {
        timer1_counter--;
        if (timer1_counter == 0) timer1_flag = 1;
    }

    if (timer2_counter > 0) {
        timer2_counter--;
        if (timer2_counter == 0) timer2_flag = 1;
    }

    if (timer3_counter > 0) {
        timer3_counter--;
        if (timer3_counter == 0) timer3_flag = 1;
    }
}

// Define functions to get the value of flag indicators
int getTimer0Flag(void) {
    return timer0_flag;
}

int getTimer1Flag(void) {
    return timer1_flag;
}

int getTimer2Flag(void) {
    return timer2_flag;
}

int getTimer3Flag(void) {
    return timer3_flag;
}

// Callback function called when timer period elapses
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
    timer_run();
    if(htim->Instance == TIM2) {
        button_reading();
    }
}
```

Program 3.10: timer.c

5.6 Exercise 5: Adding code for button debouncing

Following the example of button reading and debouncing in the previous section, your tasks in this exercise are:

- To add new files for input reading and output display,
- To add code for button debouncing,
- To add code for increasing mode when the first button is pressed.

Please add your report here.

```c
/*
 * input_reading.h
 *
 *  Created on: Oct 28, 2024
 *      Author: phihv
 */

#ifndef INC_INPUT_READING_H_
#define INC_INPUT_READING_H_

#include "main.h"

// Structure to manage button information
typedef struct {
    GPIO_TypeDef *pGPIOx;  // GPIO port of the button
    uint16_t pin;          // GPIO pin corresponding to the button
} BUTTON_CONTROL;

// Array containing the buttons
extern BUTTON_CONTROL Button[];

// Function declarations related to button control
void init_button();                           // Initialize button
void button_reading(void);                    // Read button signal
unsigned char is_button_pressed(unsigned char index);  // Check if button is pressed
unsigned char is_button_pressed_1s(unsigned char index); // Check if button is pressed for more than 1 second
unsigned char is_button_held(unsigned char index);      // Check if button is held
void reset_flagForButtonHold(unsigned char index);      // Reset button hold flag

#endif /* INC_INPUT_READING_H_ */
```

Program 3.11: Header files for input reading

```c
/*
 * input_reading.c
 *
 *  Created on: Oct 28, 2024
 *      Author: phihv
 */

#include "main.h"
#include "input_reading.h"

// Number of buttons
#define NO_OF_BUTTONS 3

BUTTON_CONTROL Button[] = {
    {BUTTON_1_GPIO_Port, BUTTON_1_Pin},
    {BUTTON_2_GPIO_Port, BUTTON_2_Pin},
    {BUTTON_3_GPIO_Port, BUTTON_3_Pin},
};

// Define 1 second duration (10ms * 100 = 1 second)
#define DURATION_FOR_AUTO_INCREASING 100
#define DURATION_FOR_HOLD 50

// Define button states
#define BUTTON_IS_PRESSED GPIO_PIN_RESET
#define BUTTON_IS_RELEASED GPIO_PIN_SET

// Buffer to store the final result after debouncing
static GPIO_PinState buttonBuffer[NO_OF_BUTTONS];

// Three buffers for debouncing
static GPIO_PinState debounceButtonBuffer1[NO_OF_BUTTONS];
static GPIO_PinState debounceButtonBuffer2[NO_OF_BUTTONS];
static GPIO_PinState debounceButtonBuffer3[NO_OF_BUTTONS];

// Flag to indicate if the button is pressed and held for more than 1 second
static uint8_t flagForButtonPress1s[NO_OF_BUTTONS];
static uint8_t flagForButtonHold[NO_OF_BUTTONS];

// Counter for the time the button is held for more than 1 second
static uint16_t counterForButtonPress1s[NO_OF_BUTTONS];
static uint16_t counterForButtonHold[NO_OF_BUTTONS];

void button_reading(void) {
    for (char i = 0; i < NO_OF_BUTTONS; i++) {
        // Transfer data between buffers
        debounceButtonBuffer3[i] = debounceButtonBuffer2[i];
        debounceButtonBuffer2[i] = debounceButtonBuffer1[i];
        debounceButtonBuffer1[i] = HAL_GPIO_ReadPin(Button[i].pGPIOx, Button[i].pin);

        // If the readings are identical, store the value in buttonBuffer
        if (debounceButtonBuffer1[i] == debounceButtonBuffer2[i] && debounceButtonBuffer2[i] == debounceButtonBuffer3[i])
      {
            buttonBuffer[i] = debounceButtonBuffer1[i];
        }

        // If the button is pressed
        if (buttonBuffer[i] == BUTTON_IS_PRESSED) {
            // Increment the counter if the button is held for less than 1 second
            if (counterForButtonPress1s[i] < DURATION_FOR_AUTO_INCREASING) {
                counterForButtonPress1s[i]++;
            } else {
                // Set the flag when the button is held for more than 1 second
                flagForButtonPress1s[i] = 1;

                if (counterForButtonHold[i] < DURATION_FOR_HOLD) {
                    counterForButtonHold[i]++;
                }
                if (counterForButtonHold[i] >= DURATION_FOR_HOLD) {
                    counterForButtonHold[i] = 0;
                    flagForButtonHold[i] = 1;
                }
            }
        } else {
            // Reset the counter and flag if the button is released
            counterForButtonPress1s[i] = 0;
            flagForButtonPress1s[i] = 0;
            counterForButtonHold[i] = 0;
            flagForButtonHold[i] = 0;
        }
    }
}

// Check if a button is pressed
unsigned char is_button_pressed(uint8_t index) {
    if (index >= NO_OF_BUTTONS) return 0;  // Validate the index
    return (buttonBuffer[index] == BUTTON_IS_PRESSED);
}

// Check if a button is pressed for more than 1 second
unsigned char is_button_pressed_1s(unsigned char index) {
    if (index >= NO_OF_BUTTONS) return 0xFF;  // Validate the index
    return (flagForButtonPress1s[index] == 1);
}

// Check if a button is held continuously
unsigned char is_button_held(unsigned char index) {
    if (index >= NO_OF_BUTTONS) return 0;
    return (flagForButtonHold[index] == 1);
}

```

```c
void reset_flagForButtonHold(unsigned char index) {
    flagForButtonHold[index] = 0;
}

void init_button() {
    // Initialize all buttons to the released state
    for (int i = 0; i < NO_OF_BUTTONS; i++) {
        buttonBuffer[i] = BUTTON_IS_RELEASED;
        debounceButtonBuffer1[i] = BUTTON_IS_RELEASED;
        debounceButtonBuffer2[i] = BUTTON_IS_RELEASED;
        debounceButtonBuffer3[i] = BUTTON_IS_RELEASED;

        // Reset flags and counters for each button
        flagForButtonPress1s[i] = 0;
        flagForButtonHold[i] = 0;
        counterForButtonPress1s[i] = 0;
        counterForButtonHold[i] = 0;
    }
}
```

Program 3.12: Source code for reading input

```c
/*
 * input_processing.c
 *
 *  Created on: Oct 28, 2024
 *      Author: phihv
 */

#include "main.h"
#include "input_processing.h"
#include "global.h"
ButtonState buttonState[3] = { BUTTON_RELEASED, BUTTON_RELEASED, BUTTON_RELEASED };

// Define global variables
int mode = 1;
int red_value = 5, yellow_value = 2, green_value = 3;
int red_temp = 0, yellow_temp = 0, green_temp = 0;

void Mode_Buffer()
{
    switch(buttonState[0])
    {
        case BUTTON_RELEASED:
            if(is_button_pressed(0))
            {
                buttonState[0] = BUTTON_PRESSED;
                mode++;
                if (mode > 4)
                {
                    mode = 1;
                    resetCountValue();
                }
            }
            break;

        case BUTTON_PRESSED:
            if(!is_button_pressed(0))
            {
                buttonState[0] = BUTTON_RELEASED;
            }
            break;

        default:
            break;
    }
}

static void increaseTimeValue()
{
    switch(mode)
    {
        case 2:
            red_temp = red_temp + 1;
            if (red_temp > 99)
            {
                red_temp = 0;
            }
            break;
        case 3:
            yellow_temp = yellow_temp + 1;
            if (yellow_temp > 99)
            {
                yellow_temp = 0;
            }
            break;
        case 4:
            green_temp = green_temp + 1;
            if (green_temp > 99)
            {
                green_temp = 0;
            }
            break;
        default:
            break;
    }
}

void Duration_Update()
{
    int diff = 0;

    switch (mode)
    {
        case 2:
            diff = red_temp - red_value;
            red_value = red_temp;
            green_value += diff;
            green_temp += diff;
            break;
        case 3:
            diff = yellow_temp - yellow_value;
            yellow_value = yellow_temp;
            red_value += diff;
            red_temp += diff;
            break;
        case 4:
            diff = green_temp - green_value;
            green_value = green_temp;
            red_value += diff;
            red_temp += diff;
            break;
        default:
            break;
    }
}

void Update_value()
{
    switch(buttonState[1])
    {
        case BUTTON_RELEASED:
            if (is_button_pressed(1))
            {
                buttonState[1] = BUTTON_PRESSED;
                increaseTimeValue();
            }
            break;

        case BUTTON_PRESSED:
            if (!is_button_pressed(1))
            {
                buttonState[1] = BUTTON_RELEASED;
            }
            if(is_button_pressed_1s(1))
            {
                buttonState[1] = BUTTON_PRESSED_MORE_THAN_1_SECOND;
                increaseTimeValue();
            }
            break;

        case BUTTON_PRESSED_MORE_THAN_1_SECOND:
            if (!is_button_pressed(1))
            {
                buttonState[1] = BUTTON_RELEASED;
            }
            if (is_button_held(1))
            {
                increaseTimeValue();
                reset_flagForButtonHold(1);
            }
            break;

        default:
            break;
    }

    switch(buttonState[2])
    {
        case BUTTON_RELEASED:
            if (is_button_pressed(2))
            {
                buttonState[2] = BUTTON_PRESSED;
                Duration_Update();
                mode = 1;
            }
            break;

        case BUTTON_PRESSED:
            if (!is_button_pressed(2))
            {
                buttonState[2] = BUTTON_RELEASED;
            }
            break;

        default:
            break;
    }
}

// FSM for button input processing
void fsm_for_input_processing()
{
    Mode_Buffer();  // Manage mode

    if (getTimer2Flag())
    {  // Check Timer 2 Flag
        update7SEG();  // Update 7-segment display
        setTimer2(8);  // Reset Timer 2
    }
    displayMode();   // Control LEDs based on mode
    Update_value();  // Update values
}
```

Program 3.13: Code for input processing

5.7 Exercise 6: Adding code for displaying modes

Your tasks in this exercise are:

- To add code for display mode on seven-segment LEDs, and
- To add code for blinking LEDs depending on the mode that is selected.

```c
/*
 * led_display.c
 *
 *  Created on: Oct 28, 2024
 *      Author: phihv
 */

#include "led_display.h"
#include "global.h"
#include "timer.h"

void display7SEG(int num)
{
    // Turn off all segments first (for common anode, set
  all to HIGH)
    HAL_GPIO_WritePin(SEG_A_GPIO_Port, SEG_A_Pin,
  GPIO_PIN_SET);
    HAL_GPIO_WritePin(SEG_B_GPIO_Port, SEG_B_Pin,
  GPIO_PIN_SET);
    HAL_GPIO_WritePin(SEG_C_GPIO_Port, SEG_C_Pin,
  GPIO_PIN_SET);
    HAL_GPIO_WritePin(SEG_D_GPIO_Port, SEG_D_Pin,
  GPIO_PIN_SET);
    HAL_GPIO_WritePin(SEG_E_GPIO_Port, SEG_E_Pin,
  GPIO_PIN_SET);
    HAL_GPIO_WritePin(SEG_F_GPIO_Port, SEG_F_Pin,
  GPIO_PIN_SET);
    HAL_GPIO_WritePin(SEG_G_GPIO_Port, SEG_G_Pin,
  GPIO_PIN_SET);

    switch(num)
    {
        case 0:
            HAL_GPIO_WritePin(SEG_A_GPIO_Port, SEG_A_Pin,
  GPIO_PIN_RESET);
            HAL_GPIO_WritePin(SEG_B_GPIO_Port, SEG_B_Pin,
  GPIO_PIN_RESET);
            HAL_GPIO_WritePin(SEG_C_GPIO_Port, SEG_C_Pin,
  GPIO_PIN_RESET);
            HAL_GPIO_WritePin(SEG_D_GPIO_Port, SEG_D_Pin,
  GPIO_PIN_RESET);
            HAL_GPIO_WritePin(SEG_E_GPIO_Port, SEG_E_Pin,
  GPIO_PIN_RESET);
            HAL_GPIO_WritePin(SEG_F_GPIO_Port, SEG_F_Pin,
  GPIO_PIN_RESET);
            break;
        case 1:
            HAL_GPIO_WritePin(SEG_B_GPIO_Port, SEG_B_Pin,
  GPIO_PIN_RESET);
```

```
            HAL_GPIO_WritePin(SEG_C_GPIO_Port, SEG_C_Pin,
   GPIO_PIN_RESET);
            break;
        case 2:
            HAL_GPIO_WritePin(SEG_A_GPIO_Port, SEG_A_Pin,
   GPIO_PIN_RESET);
            HAL_GPIO_WritePin(SEG_B_GPIO_Port, SEG_B_Pin,
   GPIO_PIN_RESET);
            HAL_GPIO_WritePin(SEG_G_GPIO_Port, SEG_G_Pin,
   GPIO_PIN_RESET);
            HAL_GPIO_WritePin(SEG_E_GPIO_Port, SEG_E_Pin,
   GPIO_PIN_RESET);
            HAL_GPIO_WritePin(SEG_D_GPIO_Port, SEG_D_Pin,
   GPIO_PIN_RESET);
            break;
        case 3:
            HAL_GPIO_WritePin(SEG_A_GPIO_Port, SEG_A_Pin,
   GPIO_PIN_RESET);
            HAL_GPIO_WritePin(SEG_B_GPIO_Port, SEG_B_Pin,
   GPIO_PIN_RESET);
            HAL_GPIO_WritePin(SEG_C_GPIO_Port, SEG_C_Pin,
   GPIO_PIN_RESET);
            HAL_GPIO_WritePin(SEG_D_GPIO_Port, SEG_D_Pin,
   GPIO_PIN_RESET);
            HAL_GPIO_WritePin(SEG_G_GPIO_Port, SEG_G_Pin,
   GPIO_PIN_RESET);
            break;
        case 4:
            HAL_GPIO_WritePin(SEG_F_GPIO_Port, SEG_F_Pin,
   GPIO_PIN_RESET);
            HAL_GPIO_WritePin(SEG_G_GPIO_Port, SEG_G_Pin,
   GPIO_PIN_RESET);
            HAL_GPIO_WritePin(SEG_B_GPIO_Port, SEG_B_Pin,
   GPIO_PIN_RESET);
            HAL_GPIO_WritePin(SEG_C_GPIO_Port, SEG_C_Pin,
   GPIO_PIN_RESET);
            break;
        case 5:
            HAL_GPIO_WritePin(SEG_A_GPIO_Port, SEG_A_Pin,
   GPIO_PIN_RESET);
            HAL_GPIO_WritePin(SEG_F_GPIO_Port, SEG_F_Pin,
   GPIO_PIN_RESET);
            HAL_GPIO_WritePin(SEG_G_GPIO_Port, SEG_G_Pin,
   GPIO_PIN_RESET);
            HAL_GPIO_WritePin(SEG_C_GPIO_Port, SEG_C_Pin,
   GPIO_PIN_RESET);
            HAL_GPIO_WritePin(SEG_D_GPIO_Port, SEG_D_Pin,
   GPIO_PIN_RESET);
            break;
```

```
     case 6:
         HAL_GPIO_WritePin(SEG_A_GPIO_Port, SEG_A_Pin,
GPIO_PIN_RESET);
         HAL_GPIO_WritePin(SEG_F_GPIO_Port, SEG_F_Pin,
GPIO_PIN_RESET);
         HAL_GPIO_WritePin(SEG_G_GPIO_Port, SEG_G_Pin,
GPIO_PIN_RESET);
         HAL_GPIO_WritePin(SEG_C_GPIO_Port, SEG_C_Pin,
GPIO_PIN_RESET);
         HAL_GPIO_WritePin(SEG_D_GPIO_Port, SEG_D_Pin,
GPIO_PIN_RESET);
         HAL_GPIO_WritePin(SEG_E_GPIO_Port, SEG_E_Pin,
GPIO_PIN_RESET);
         break;
     case 7:
         HAL_GPIO_WritePin(SEG_A_GPIO_Port, SEG_A_Pin,
GPIO_PIN_RESET);
         HAL_GPIO_WritePin(SEG_B_GPIO_Port, SEG_B_Pin,
GPIO_PIN_RESET);
         HAL_GPIO_WritePin(SEG_C_GPIO_Port, SEG_C_Pin,
GPIO_PIN_RESET);
         break;
     case 8:
         HAL_GPIO_WritePin(SEG_A_GPIO_Port, SEG_A_Pin,
GPIO_PIN_RESET);
         HAL_GPIO_WritePin(SEG_B_GPIO_Port, SEG_B_Pin,
GPIO_PIN_RESET);
         HAL_GPIO_WritePin(SEG_C_GPIO_Port, SEG_C_Pin,
GPIO_PIN_RESET);
         HAL_GPIO_WritePin(SEG_D_GPIO_Port, SEG_D_Pin,
GPIO_PIN_RESET);
         HAL_GPIO_WritePin(SEG_E_GPIO_Port, SEG_E_Pin,
GPIO_PIN_RESET);
         HAL_GPIO_WritePin(SEG_F_GPIO_Port, SEG_F_Pin,
GPIO_PIN_RESET);
         HAL_GPIO_WritePin(SEG_G_GPIO_Port, SEG_G_Pin,
GPIO_PIN_RESET);
         break;
     case 9:
         HAL_GPIO_WritePin(SEG_A_GPIO_Port, SEG_A_Pin,
GPIO_PIN_RESET);
         HAL_GPIO_WritePin(SEG_B_GPIO_Port, SEG_B_Pin,
GPIO_PIN_RESET);
         HAL_GPIO_WritePin(SEG_C_GPIO_Port, SEG_C_Pin,
GPIO_PIN_RESET);
         HAL_GPIO_WritePin(SEG_D_GPIO_Port, SEG_D_Pin,
GPIO_PIN_RESET);
         HAL_GPIO_WritePin(SEG_F_GPIO_Port, SEG_F_Pin,
GPIO_PIN_RESET);
```

```
            HAL_GPIO_WritePin(SEG_G_GPIO_Port, SEG_G_Pin,
   GPIO_PIN_RESET);
            break;
        default:
            // Invalid number, turn off all segments
            HAL_GPIO_WritePin(SEG_A_GPIO_Port, SEG_A_Pin,
   GPIO_PIN_SET);
            HAL_GPIO_WritePin(SEG_B_GPIO_Port, SEG_B_Pin,
   GPIO_PIN_SET);
            HAL_GPIO_WritePin(SEG_C_GPIO_Port, SEG_C_Pin,
   GPIO_PIN_SET);
            HAL_GPIO_WritePin(SEG_D_GPIO_Port, SEG_D_Pin,
   GPIO_PIN_SET);
            HAL_GPIO_WritePin(SEG_E_GPIO_Port, SEG_E_Pin,
   GPIO_PIN_SET);
            HAL_GPIO_WritePin(SEG_F_GPIO_Port, SEG_F_Pin,
   GPIO_PIN_SET);
            HAL_GPIO_WritePin(SEG_G_GPIO_Port, SEG_G_Pin,
   GPIO_PIN_SET);
            break;
    }
}

// Variable declarations
int led1 = RED_TIME;
int led2 = GREEN_TIME;

int state1 = AUTO_RED;
int state2 = AUTO_GREEN;

void resetCountValue()
{
    led1 = red_value;
    led2 = green_value;
    state1 = AUTO_RED;
    state2 = AUTO_GREEN;
}

void mode1Counter()
{
    led1--;
    led2--;
    switch (state1) {
        case AUTO_RED:
            HAL_GPIO_WritePin(LED_RED_GPIO_Port,
   LED_RED_Pin, GPIO_PIN_RESET);
            HAL_GPIO_WritePin(LED_YELLOW_GPIO_Port,
   LED_YELLOW_Pin, GPIO_PIN_SET);
            HAL_GPIO_WritePin(LED_GREEN_GPIO_Port,
```

```c
LED_GREEN_Pin, GPIO_PIN_SET);

            if (led1 < 0) {
                led1 = green_value;
                state1 = AUTO_GREEN;
                HAL_GPIO_WritePin(LED_RED_GPIO_Port,
LED_RED_Pin, GPIO_PIN_SET);
                HAL_GPIO_WritePin(LED_YELLOW_GPIO_Port,
LED_YELLOW_Pin, GPIO_PIN_SET);
                HAL_GPIO_WritePin(LED_GREEN_GPIO_Port,
LED_GREEN_Pin, GPIO_PIN_RESET);
            }
            break;
        case AUTO_YELLOW:
            HAL_GPIO_WritePin(LED_RED_GPIO_Port,
LED_RED_Pin, GPIO_PIN_SET);
            HAL_GPIO_WritePin(LED_YELLOW_GPIO_Port,
LED_YELLOW_Pin, GPIO_PIN_RESET);
            HAL_GPIO_WritePin(LED_GREEN_GPIO_Port,
LED_GREEN_Pin, GPIO_PIN_SET);

            if (led1 < 0) {
                led1 = red_value;
                state1 = AUTO_RED;
                HAL_GPIO_WritePin(LED_RED_GPIO_Port,
LED_RED_Pin, GPIO_PIN_RESET);
                HAL_GPIO_WritePin(LED_YELLOW_GPIO_Port,
LED_YELLOW_Pin, GPIO_PIN_SET);
                HAL_GPIO_WritePin(LED_GREEN_GPIO_Port,
LED_GREEN_Pin, GPIO_PIN_SET);
            }
            break;
        case AUTO_GREEN:
            HAL_GPIO_WritePin(LED_RED_GPIO_Port,
LED_RED_Pin, GPIO_PIN_SET);
            HAL_GPIO_WritePin(LED_YELLOW_GPIO_Port,
LED_YELLOW_Pin, GPIO_PIN_SET);
            HAL_GPIO_WritePin(LED_GREEN_GPIO_Port,
LED_GREEN_Pin, GPIO_PIN_RESET);

            if (led1 < 0) {
                led1 = yellow_value;
                state1 = AUTO_YELLOW;
                HAL_GPIO_WritePin(LED_RED_GPIO_Port,
LED_RED_Pin, GPIO_PIN_SET);
                HAL_GPIO_WritePin(LED_YELLOW_GPIO_Port,
LED_YELLOW_Pin, GPIO_PIN_RESET);
                HAL_GPIO_WritePin(LED_GREEN_GPIO_Port,
LED_GREEN_Pin, GPIO_PIN_SET);
```

```
            }
            break;
        default:
            break;
    }

    switch (state2)
    {
        case AUTO_RED:
            HAL_GPIO_WritePin(LED_RED_2_GPIO_Port,
  LED_RED_2_Pin, GPIO_PIN_RESET);
            HAL_GPIO_WritePin(LED_YELLOW_2_GPIO_Port,
  LED_YELLOW_2_Pin, GPIO_PIN_SET);
            HAL_GPIO_WritePin(LED_GREEN_2_GPIO_Port,
  LED_GREEN_2_Pin, GPIO_PIN_SET);

            if (led2 < 0) {
                led2 = green_value;
                state2 = AUTO_GREEN;
                HAL_GPIO_WritePin(LED_RED_2
```

5.8 Exercise 7: Adding code for increasing time duration value for the red LEDs

5.9 Exercise 8: Adding code for increasing time duration value for the amber LEDs

5.10 Exercise 9: Adding code for increasing time duration value for the green LEDs

```
int mode = 1;
int red_value = 5, yellow_value = 2, green_value = 3;
int red_temp = 0, yellow_temp = 0, green_temp = 0;

void Mode_Buffer() {
    switch(buttonState[0]) {
        case BUTTON_RELEASED:
            if(is_button_pressed(0)) {
                buttonState[0] = BUTTON_PRESSED;
                mode++;
                if (mode > 4) {
                    mode = 1;
                    resetCountValue();
                }
            }
            break;

```

```c
        case BUTTON_PRESSED:
            if(!is_button_pressed(0)) {
                buttonState[0] = BUTTON_RELEASED;
            }
            break;

        default:
            break;
    }
}

static void increaseTimeValue() {
    switch(mode) {
        case 2:
            red_temp = red_temp + 1;
            if (red_temp > 99) {
                red_temp = 0;
            }
            break;
        case 3:
            yellow_temp = yellow_temp + 1;
            if (yellow_temp > 99) {
                yellow_temp = 0;
            }
            break;
        case 4:
            green_temp = green_temp + 1;
            if (green_temp > 99) {
                green_temp = 0;
            }
            break;
        default:
            break;
    }
}

void Duration_Update() {
    int diff = 0;

    switch (mode) {
        case 2:
            diff = red_temp - red_value;
            red_value = red_temp;
            green_value += diff;
            green_temp += diff;
            break;
        case 3:
            diff = yellow_temp - yellow_value;
            yellow_value = yellow_temp;
            red_value += diff;
            red_temp += diff;
            break;
        case 4:
            diff = green_temp - green_value;
            green_value = green_temp;
            red_value += diff;
            red_temp += diff;
            break;
        default:
            break;
    }
}

void Update_value() {
    switch(buttonState[1]) {
        case BUTTON_RELEASED:
            if (is_button_pressed(1)) {
                buttonState[1] = BUTTON_PRESSED;
                increaseTimeValue();
            }
            break;

        case BUTTON_PRESSED:
            if (!is_button_pressed(1)) {
                buttonState[1] = BUTTON_RELEASED;
            }
            if(is_button_pressed_1s(1)) {
                buttonState[1] =
   BUTTON_PRESSED_MORE_THAN_1_SECOND;
                increaseTimeValue();
            }
            break;

        case BUTTON_PRESSED_MORE_THAN_1_SECOND:
            if (!is_button_pressed(1)) {
                buttonState[1] = BUTTON_RELEASED;
            }
            if (is_button_held(1)) {
                increaseTimeValue();
                reset_flagForButtonHold(1);
            }
            break;

        default:
            break;
    }

    switch(buttonState[2]) {
        case BUTTON_RELEASED:
            if (is_button_pressed(2)) {
                buttonState[1] = BUTTON_PRESSED;
                Duration_Update();
                mode = 1;
            }
            break;

        case BUTTON_PRESSED:
            if (!is_button_pressed(2)) {
                buttonState[2] = BUTTON_RELEASED;
            }
            break;

        default:
            break;
    }
}
```
Program 3.14: CODE FOR EX 7 8 9

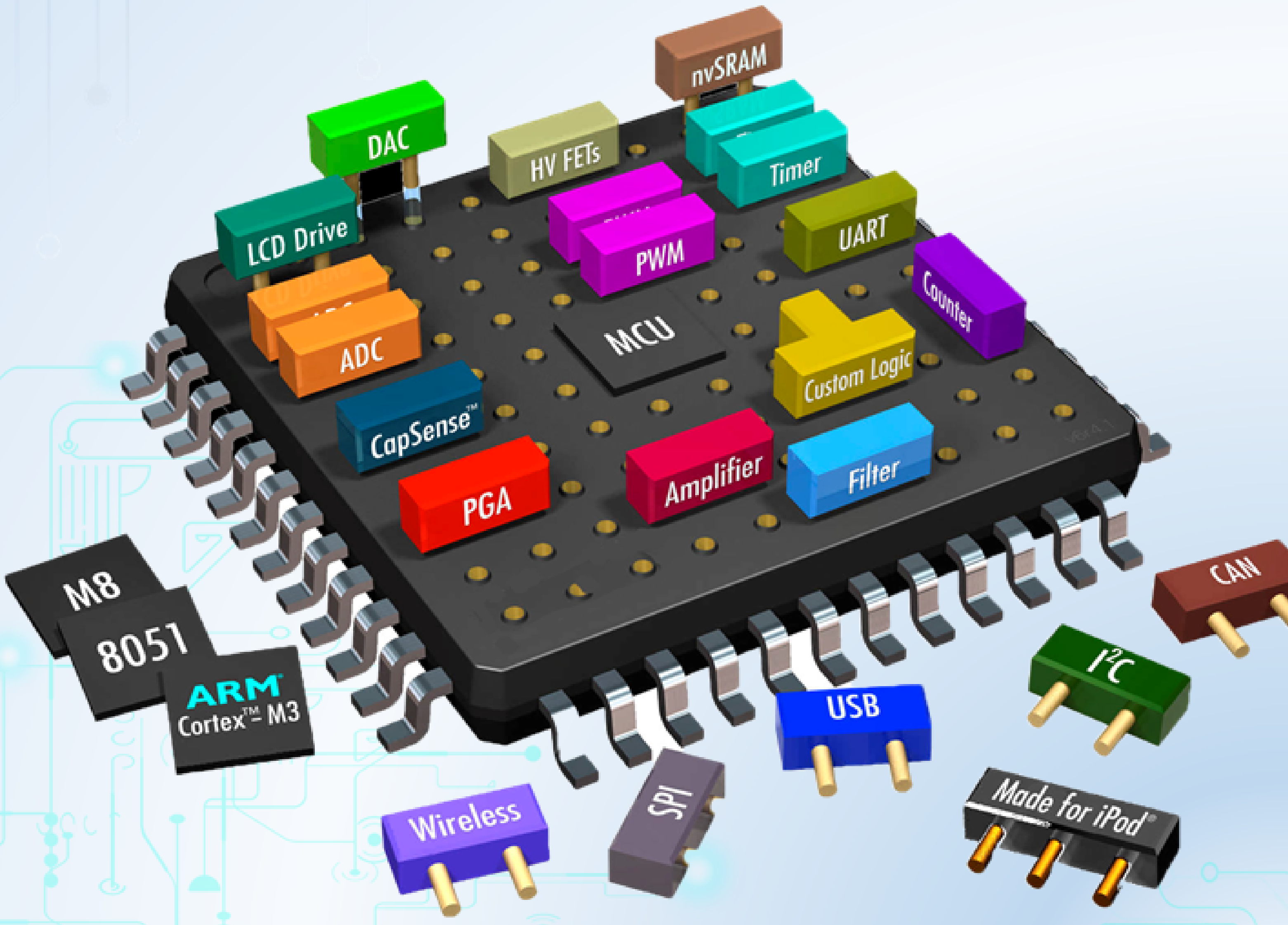
5.11 Exercise 10: To finish the project

Your tasks in this exercise are:

- To integrate all the previous tasks to one final project
- To create a video to show all features in the specification
- To add a report to describe your solution for each exercise.
- To submit your report and code on the BKeL

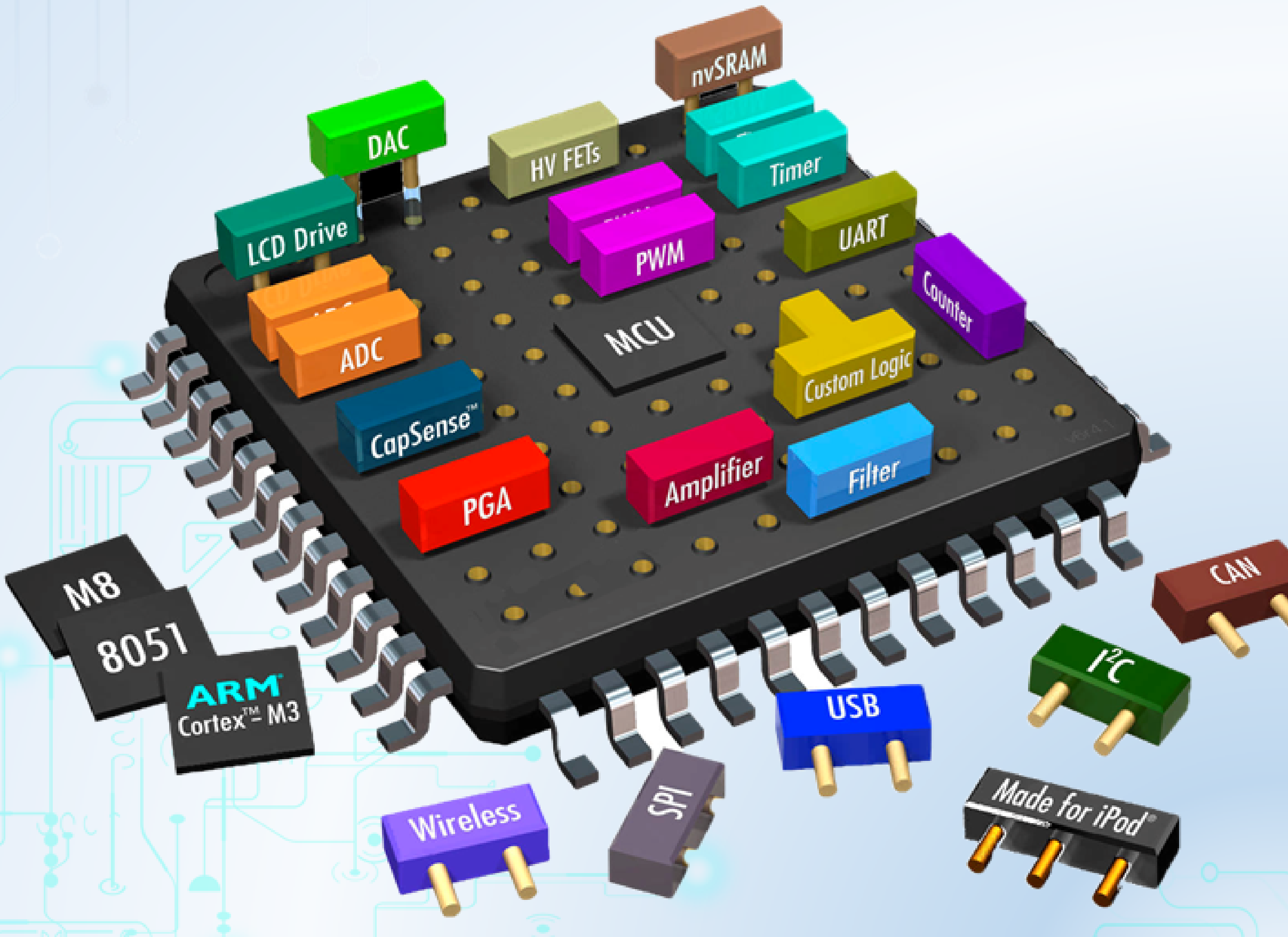
CHƯƠNG 4

Digital Clock Project



CHƯƠNG 5

A cooperative scheduler



1 Introduction

1.1 Super Loop Architecture

```c
}
void main(void){
    // Prepare for Task X
    X_Init();
    while(1) {// 'for ever' (Super Loop)
        X(); // Perform the task
    }
}
```

Program 5.1: Super loop program

The main advantages of the Super Loop architecture illustrated above are:

- (1) that it is simple, and therefore easy to understand, and
- (2) that it consumes virtually no system memory or CPU resources.

However, we get 'nothing for nothing': Super Loops consume little memory or processor resources because they provide few facilities to the developer. A particular limitation with this architecture is that it is very difficult to execute Task X at precise intervals of time: as we will see, this is a very significant drawback.

For example, consider a collection of requirements assembled from a range of different embedded projects (in no particular order):

- The current speed of the vehicle must be measured at 0.5 second intervals.
- The display must be refreshed 40 times every second.
- The calculated new throttle setting must be applied every 0.5 seconds.
- A time-frequency transform must be performed 20 times every second.
- If the alarm sounds, it must be switched off (for legal reasons) after 20 minutes.
- If the front door is opened, the alarm must sound in 30 seconds if the correct password is not entered in this time.
- The engine vibration data must be sampled 1,000 times per second.
- The frequency-domain data must be classified 20 times every second.
- The keypad must be scanned every 200 ms.
- The master (control) node must communicate with all other nodes (sensor nodes and sounder nodes) once per second.
- The new throttle setting must be calculated every 0.5 seconds.

- The sensors must be sampled once per second.

We can summarize this list by saying that many embedded systems must carry out tasks at particular instants of time. More specifically, we have two kinds of activity to perform:

- Periodic tasks, to be performed (say) once every 100 ms
- One-shot tasks, to be performed once after a delay of (say) 50 ms

This is very difficult to achieve with the primitive architecture shown in Program above. Suppose, for example, that we need to start Task X every 200 ms, and that the task takes 10 ms to complete. Program below illustrates one way in which we might adapt the code in order to try to achieve this.

```
}
void main(void){
    // Prepare for Task X
    X_Init();
    while(1) {              // 'for ever' (Super Loop)
        X();                // Perform the task (10 ms duration)
        Delay_190ms();      // Delay for 190 ms
    }
}
```

Program 5.2: Trying to use the Super Loop architecture to execute tasks at regular intervals

The approach is not generally adequate, because it will only work if the following conditions are satisfied:

- We know the precise duration of Task X
- This duration never varies

In practical applications, determining the precise task duration is rarely straightforward. Suppose we have a very simple task that does not interact with the outside world but, instead, performs some internal calculations. Even under these rather restricted circumstances, changes to compiler optimization settings – even changes to an apparently unrelated part of the program – can alter the speed at which the task executes. This can make fine-tuning the timing very tedious and error prone.

The second condition is even more problematic. Often in an embedded system the task will be required to interact with the outside world in a complex way. In these circumstances the task duration will vary according to outside activities in a manner over which the programmer has very little control.

1.2 Timer-based interrupts and interrupt service routines

A better solution to the problems outlined is to use timer-based interrupts as a means of invoking functions at particular times.

An interrupt is a hardware mechanism used to notify a processor that an ‘event’ has taken place: such events may be internal events or external events.

When an interrupt is generated, the processor ‘jumps’ to an address at the bottom of the CODE memory area. These locations must contain suitable code with which the microcontroller can respond to the interrupt or, more commonly, the locations will include another ‘jump’ instruction, giving the address of suitable ‘interrupt service routine’ located elsewhere in (CODE) memory.

Please see lab 3 for the more information of this approach.

2 What is a scheduler?

There are two ways of viewing a scheduler:

- At one level, a scheduler can be viewed as a simple operating system that allows tasks to be called periodically or (less commonly) on a one-shot basis.
- At a lower level, a scheduler can be viewed as a single timer interrupt service routine that is shared between many different tasks. As a result, only one timer needs to be initialized, and any changes to the timing generally requires only one function to be altered. Furthermore, we can generally use the same scheduler whether we need to execute one, ten or 100 different tasks.

```
void main(void) {
    // Set up the scheduler
    SCH_Init();
    // Add the tasks (1ms tick interval)
    // Function_A will run every 2 ms
    SCH_Add_Task(Function_A, 0, 2);
    // Function_B will run every 10 ms
    SCH_Add_Task(Function_B, 1, 10);
    // Function_C will run every 15 ms
    SCH_Add_Task(Function_C, 3, 15);
    while(1) {
        SCH_Dispatch_Tasks();
    }
}
```

Program 5.3: Example of how a scheduler uses

2.1 The co-operative scheduler

A co-operative scheduler provides a single-tasking system architecture

Operation:

- Tasks are scheduled to run at specific times (either on a periodic or one-shot basis)

- When a task is scheduled to run it is added to the waiting list
- When the CPU is free, the next waiting task (if any) is executed
- The task runs to completion, then returns control to the scheduler

Implementation:

- The scheduler is simple and can be implemented in a small amount of code
- The scheduler must allocate memory for only a single task at a time
- The scheduler will generally be written entirely in a high-level language (such as 'C')
- The scheduler is not a separate application; it becomes part of the developer's code

Performance:

- Obtaining rapid responses to external events requires care at the design stage Reliability and safety:

Co-operate scheduling is simple, predictable, reliable and safe

A co-operative scheduler provides a simple, highly predictable environment. The scheduler is written entirely in 'C' and becomes part of the application: this tends to make the operation of the whole system more transparent and eases development, maintenance and porting to different environments. Memory overheads are 17 bytes per task and CPU requirements (which vary with tick interval) are low.

2.2 Function pointers

One area of the language with which many 'C' programmers are unfamiliar is the function pointer. While comparatively rarely used in desktop programs, this language feature is crucial in the creation of schedulers: we therefore provide a brief introductory example here.

The key point to note is that – just as we can, for example, determine the starting address of an array of data in memory – we can also find the address in memory at which the executable code for a particular function begins. This address can be used as a 'pointer' to the function; most importantly, it can be used to call the function. Used with care, function pointers can make it easier to design and implement complex programs. For example, suppose we are developing a large, safety-critical, application, controlling an industrial plant. If we detect a critical situation, we may wish to shut down the system as rapidly as possible. However, the appropriate way to shut down the system will vary, depending on the system state. What we can do is create a number of different recovery functions and a function pointer. Every time the system state changes, we can alter the function pointer so that it is always pointing to the most appropriate recovery function. In this way, we know that – if there is ever an emergency situation – we can rapidly call the most appropriate function, by means of the function pointer.

```c
// ------ Private function prototypes ------------------------------
void Square_Number(int, int*);

int main(void)
{
    int a = 2, b = 3;
    /* Declares pFn to be a pointer to fn with
    int and int pointer parameters (returning void) */
    void (* pFn)(int, int*);

    int Result_a, Result_b;
    pFn = Square_Number; // pFn holds address of Square_Number
    printf("Function code starts at address: %u\n", (tWord) pFn);
    printf("Data item a starts at address: %u\n\n", (tWord) &a);
    // Call 'Square_Number' in the conventional way
    Square_Number(a, &Result_a);
    // Call 'Square_Number' using function pointer
    (*pFn)(b,&Result_b);
    printf("%d squared is %d (using normal fn call)\n", a, Result_a
   );
    printf("%d squared is %d (using fn pointer)\n", b, Result_b);
    while(1);
    return 0;
}

void Square_Number(int a, int* b)
{// Demo - calculate square of a
    *b = a * a;
}
```

Program 5.4: Example of how to use function pointers

2.3 Solution

A scheduler has the following key components:

- The scheduler data structure.
- An initialization function.
- A single interrupt service routine (ISR), used to update the scheduler at regular time intervals.
- A function for adding tasks to the scheduler.
- A dispatcher function that causes tasks to be executed when they are due to run.
- A function for removing tasks from the scheduler (not required in all applications).

We consider each of the required components in this section

2.3.1 Overview

Before discussing the scheduler components, we consider how the scheduler will typically appear to the user. To do this we will use a simple example: a scheduler used to flash a single LED on and off repeatedly: on for one second off for one second etc.

```
int main(void){
    //Init all the requirments for the system to run
  System_Initialization();
  //Init a schedule
  SCH_Init();
  //Add a task to repeatly call in every 1 second.
  SCH_Add_Task(Led_Display, 0, 1000);
  while (1){
    SCH_Dispatch_Tasks();
  }
  return 0;
}
```

Program 5.5: Example of how to use a scheduler

- We assume that the LED will be switched on and off by means of a 'task' Led_Display(). Thus, if the LED is initially off and we call Led_Display() twice, we assume that the LED will be switched on and then switched off again.

  To obtain the required flash rate, we therefore require that the scheduler calls Led_Display() every second ad infinitum.

- We prepare the scheduler using the function SCH_Init().

- After preparing the scheduler, we add the function Led_Display() to the scheduler task list using the SCH_Add_Task() function. At the same time we specify that the LED will be turned on and off at the required rate as follows:

  SCH_Add_Task(Led_Display, 0, 1000);

  We will shortly consider all the parameters of SCH_Add_Task(), and examine its internal structure.

- The timing of the Led_Display() function will be controlled by the function SCH_Update(), an interrupt service routine triggered by the overflow of Timer 2:

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim){
  SCH\_Update();
}
```

Program 5.6: Example of how to call SCH_Update function

- The 'Update' function does not execute the task: it calculates when a task is due to run and sets a flag. The job of executing LED_Display() falls to the dispatcher function (SCH_Dispatch_Tasks()), which runs in the main ('super') loop:

```c
while(1){
    SCH_Dispatch_Tasks();
}
```

Before considering these components in detail, we should acknowledge that this is, undoubtedly, a complicated way of flashing an LED: if our intention were to develop an LED flasher application that requires minimal memory and minimal code size, this would not be a good solution. However, the key point is that we will be able to use the same scheduler architecture in all our subsequent examples, including a number of substantial and complex applications and the effort required to understand the operation of this environment will be rapidly repaid.

It should also be emphasized that the scheduler is a 'low-cost' option: it consumes a small percentage of the CPU resources (we will consider precise percentages shortly). In addition, the scheduler itself requires no more than 17 bytes of memory for each task. Since a typical application will require no more than four to six tasks, the task – memory budget (around 60 bytes) is not excessive, even on an 8-bit microcontroller.

2.3.2 The scheduler data structure and task array

At the heart of the scheduler is the scheduler data structure: this is a user-defined data type which collects together the information required about each task.

```c

typedef struct {
    // Pointer to the task (must be a 'void (void)' function)
  void ( * pTask)(void);
  // Delay (ticks) until the function will (next) be run
  uint32_t Delay;
  // Interval (ticks) between subsequent runs.
  uint32_t Period;
  // Incremented (by scheduler) when task is due to execute
  uint8_t RunMe;
  //This is a hint to solve the question below.
  uint32_t TaskID;
} sTask;

// MUST BE ADJUSTED FOR EACH NEW PROJECT
#define SCH_MAX_TASKS          40
#define NO_TASK_ID             0
sTask SCH_tasks_G[SCH_MAX_TASKS];
```
Program 5.7: A struct of a task

The size of the task array

You must ensure that the task array is sufficiently large to store the tasks required in your application, by adjusting the value of SCH_MAX_TASKS. For example, if you schedule three tasks as follows:

- SCH_Add_Task(Function_A, 0, 2);

- SCH_Add_Task(Function_B, 1, 10);
- SCH_Add_Task(Function_C, 3, 15);

then SCH_MAX_TASKS must have a value of three (or more) for correct operation of the scheduler.

Note also that, if this condition is not satisfied, the scheduler should generate an error code.

2.3.3 The initialization function

Like most of the tasks we wish to schedule, the scheduler itself requires an initialization function. While this performs various important operations – such as preparing the scheduler array (discussed earlier) and the error code variable (discussed later) – the main purpose of this function is to set up a timer that will be used to generate the regular 'ticks' that will drive the scheduler.

```c
void SCH_Init(void) {
    unsigned char i;
    for (i = 0; i < SCH_MAX_TASKS; i++) {
        SCH_Delete_Task(i);
    }
    // Reset the global error variable
    // - SCH_Delete_Task() will generate an error code,
    // (because the task array is empty)
    Error_code_G = 0;
    Timer_init();
    Watchdog_init();
}
```

Program 5.8: Example of how

2.3.4 The 'Update' function

The 'Update' function is involved in the ISR. It is invoked when the timer is overflow.

When it determines that a task is due to run, the update function increments the RunMe field for this task: the task will then be executed by the dispatcher, as we discuss later.

```c
void SCH_Update(void){
    unsigned char Index;
    // NOTE: calculations are in *TICKS* (not milliseconds)
    for (Index = 0; Index < SCH_MAX_TASKS; Index++) {
        // Check if there is a task at this location
        if (SCH_tasks_G[Index].pTask){
            if (SCH_tasks_G[Index].Delay == 0) {
                // The task is due to run
                // Inc. the 'RunMe' flag
                SCH_tasks_G[Index].RunMe += 1;
                if (SCH_tasks_G[Index].Period) {
                    // Schedule periodic tasks to run again
                    SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
                }
            } else {
                // Not yet ready to run: just decrement the delay
                SCH_tasks_G[Index].Delay -= 1;
            }
        }
    }
}

void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
  SCH_Update();
}
```
Program 5.9: Example of how to write an SCH_Update function

2.3.5 The 'Add Task' function

As the name suggests, the 'Add Task' function is used to add tasks to the task array, to ensure that they are called at the required time(s). Here is the example of add task function: **unsigned char SCH_Add_Task (Task_Name , Initial_Delay, Period)**

The parameters for the 'Add Task' function are described as follows:

- **Task_Name**: the name of the function (task) that you wish to schedule
- **Initial_Delay**: the delay (in ticks) before task is first executed. If set to 0, the task is executed immediately.
- **Period**: the interval (in ticks) between repeated executions of the task. If set to 0, the task is executed only once

Here are some examples.

This set of parameters causes the function Do_X() to be executed once after 1,000 scheduler ticks:

SCH_Add_Task(Do_X,1000,0);

This does the same, but saves the task ID (the position in the task array) so that the task may be subsequently deleted, if necessary (see SCH_Delete_Task() for further information about the removal of tasks from the task array):

Task_ID = SCH_Add_Task(Do_X,1000,0);

This causes the function Do_X() to be executed regularly every 1,000 scheduler ticks; the task will first be executed as soon as the scheduling is started:

SCH_Add_Task(Do_X,0,1000);

This causes the function Do_X() to be executed regularly every 1,000 scheduler ticks; task will be first executed at T = 300 ticks, then 1,300, 2,300 etc:

SCH_Add_Task(Do_X,300,1000);

```
/*
   ------------------------------------------------------------------*-
SCH_Add_Task() Causes a task (function) to be executed at regular
   intervals
or after a user-defined delay
-*------------------------------------------------------------------
   */
unsigned char SCH_Add_Task(void (* pFunction)(), unsigned int DELAY
   , unsigned int PERIOD)
{
    unsigned char Index = 0;
    // First find a gap in the array (if there is one)
    while ((SCH_tasks_G[Index].pTask != 0) && (Index <
   SCH_MAX_TASKS))
    {
       Index++;
    }
    // Have we reached the end of the list?
    if (Index == SCH_MAX_TASKS)
    {
        // Task list is full
        // Set the global error variable
        Error_code_G = ERROR_SCH_TOO_MANY_TASKS;
        // Also return an error code
        return SCH_MAX_TASKS;
    }
    // If we're here, there is a space in the task array
    SCH_tasks_G[Index].pTask = pFunction;
    SCH_tasks_G[Index].Delay = DELAY;
    SCH_tasks_G[Index].Period = PERIOD;
    SCH_tasks_G[Index].RunMe = 0;
    // return position of task (to allow later deletion)
    return Index;
}
```

Program 5.10: An implementation of the scheduler ‘add task’ function

2.3.6 The ‘Dispatcher’

As we have seen, the ‘Update’ function does not execute any tasks: the tasks that are due to run are invoked through the ‘Dispatcher’ function.

```
void SCH_Dispatch_Tasks(void)
{
    unsigned char Index;
    // Dispatches (runs) the next task (if one is ready)
    for (Index = 0; Index < SCH_MAX_TASKS; Index++){
        if (SCH_tasks_G[Index].RunMe > 0) {
            (*SCH_tasks_G[Index].pTask)(); // Run the task
            SCH_tasks_G[Index].RunMe -= 1; // Reset / reduce RunMe flag
            // Periodic tasks will automatically run again
            // - if this is a 'one shot' task, remove it from the array
            if (SCH_tasks_G[Index].Period == 0)
            {
                SCH_Delete_Task(Index);
            }
        }
    }
    // Report system status
    SCH_Report_Status();
    // The scheduler enters idle mode at this point
    SCH_Go_To_Sleep();
}
```

Program 5.11: An implementation of the scheduler 'dispatch task' function

The dispatcher is the only component in the Super Loop:

```
void main(void)
{
    ...
    while(1)
    {
        SCH_Dispatch_Tasks();
    }
```

Program 5.12: The dispatcher in the super loop

Do we need a Dispatch function?

At first inspection, the use of both the 'Update' and 'Dispatch' functions may seem a rather complicated way of running the tasks. Specifically, it may appear that the Dispatch function in unnecessary and that the Update function could invoke the tasks directly. However, the split between the Update and Dispatch operations is necessary, to maximize the reliability of the scheduler in the presence of long tasks.

Suppose we have a scheduler with a tick interval of 1 ms and, for whatever reason, a scheduled task sometimes has a duration of 3 ms.

If the Update function runs the functions directly then – all the time the long task is being executed – the tick interrupts are effectively disabled. Specifically, two 'ticks' will be missed. This will mean that all system timing is seriously affected and may mean that two (or more) tasks are not scheduled to execute at all.

If the Update and Dispatch function are separated, system ticks can still be processed while the long task is executing. This means that we will suffer task 'jitter' (the 'missing' tasks will not be run at the correct time), but these tasks will, eventually, run.

2.3.7 The 'Delete Task' function

When tasks are added to the task array, SCH_Add_Task() returns the position in the task array at which the task has been added: Task_ID = SCH_Add_Task(Do_X,1000,0);

Sometimes it can be necessary to delete tasks from the array. To do so, SCH_Delete_Task() can be used as follows: SCH_Delete_Task(Task_ID)

```
/*
   -------------------------------------------------------------------
   */
unsigned char SCH_Delete_Task(const tByte TASK_INDEX){
    unsigned char Return_code;
    if (SCH_tasks_G[TASK_INDEX].pTask == 0) {
        // No task at this location...
        //
        // Set the global error variable
        Error_code_G = ERROR_SCH_CANNOT_DELETE_TASK

        // ...also return an error code
        Return_code = RETURN_ERROR;
    } else {
        Return_code = RETURN_NORMAL;
    }
    SCH_tasks_G[TASK_INDEX].pTask = 0x0000;
    SCH_tasks_G[TASK_INDEX].Delay = 0;
    SCH_tasks_G[TASK_INDEX].Period = 0;
    SCH_tasks_G[TASK_INDEX].RunMe = 0;
    return Return_code; // return status
}
```

Program 5.13: An implementation of the scheduler 'delete task' function

2.3.8 Reducing power consumption

An important feature of scheduled applications is that they can lend themselves to low-power operation. This is possible because all modern MCU provide an 'idle' mode, where the CPU activity is halted, but the state of the processor is maintained. In this mode, the power required to run the processor is typically reduced by around 50

This idle mode is particularly effective in scheduled applications because it may be entered under software control, and the MCU returns to the normal operating mode when any interrupt is received. Because the scheduler generates regular timer interrupts as a matter of course, we can put the system ' to sleep' at the end of every dispatcher call: it will then wake up when the next timer tick occurs.

This is an optional feature. Students can do by yourself by looking at the reference manual of the MCU that is used.

```
void SCH_Go_To_Sleep(){
    //todo: Optional
}
```
Program 5.14: An implementation of the scheduler 'go to sleep' function

2.3.9 Reporting errors

Hardware fails; software is never perfect; errors are a fact of life. To report errors at any part of the scheduled application, we can use an (8-bit) error code variable Error_code_G

unsigned char Error_code_G = 0;

To record an error we include lines such as:

- Error_code_G = ERROR_SCH_TOO_MANY_TASKS;
- Error_code_G = ERROR_SCH_WAITING_FOR_SLAVE_TO_ACK;
- Error_code_G = ERROR_SCH_WAITING_FOR_START_COMMAND_FROM_MASTER;
- Error_code_G = ERROR_SCH_ONE_OR_MORE_SLAVES_DID_NOT_START;
- Error_code_G = ERROR_SCH_LOST_SLAVE;
- Error_code_G = ERROR_SCH_CAN_BUS_ERROR;
- Error_code_G = ERROR_I2C_WRITE_BYTE_AT24C64;

To report these error codes, the scheduler has a function **SCH_Report_Status()**, which is called from the Update function.

```
void SCH_Report_Status(void) {
#ifdef SCH_REPORT_ERRORS
    // ONLY APPLIES IF WE ARE REPORTING ERRORS
    // Check for a new error code
    if (Error_code_G != Last_error_code_G) {
        // Negative logic on LEDs assumed
        Error_port = 255 - Error_code_G;
        Last_error_code_G = Error_code_G;
        if (Error_code_G != 0){
            Error_tick_count_G = 60000;
        } else {
            Error_tick_count_G = 0;
        }
    } else {
        if (Error_tick_count_G != 0){
            if (--Error_tick_count_G == 0)    {
                Error_code_G = 0; // Reset error code
            }
        }
    }
#endif
}
```
Program 5.15: An implementation of the 'report status' function

Note that error reporting may be disabled via the main.h header file:

```
// Comment this line out if error reporting is NOT required
//#define SCH_REPORT_ERRORS
//Where error reporting is required, the port on which error codes
   will be displayed
//is also determined via main.h:
#ifdef SCH_REPORT_ERRORS
// The port on which error codes will be displayed
// ONLY USED IF ERRORS ARE REPORTED
#define Error_port PORTA
#endif
```

Program 5.16: Define a constant to allow errors are reported

Note that, in this implementation, error codes are reported for 60,000 ticks (1 minute at a 1 ms tick rate). The simplest way of displaying these codes is to attach eight LEDs (with suitable buffers) to the error port, as discussed in IC DRIVER [page 134]: Figure 14.3 illustrates one possible approach.

What does that error code mean? The forms of error reporting discussed here are low-level in nature and are primarily intended to assist the developer of the application or a qualified service engineer performing system maintenance. An additional user interface may also be required in your application to notify the user of errors, in a more user-friendly manner.

2.3.10 Adding a watchdog

The basic scheduler presented here does not provide support for a watchdog timer. Such support can be useful and is easily added, as follows:

- Start the watchdog in the scheduler Start function.
- Refresh the watchdog in the scheduler Update function.

```
IWDG_HandleTypeDef hiwdg;
static uint32_t counter_for_watchdog = 0;

void MX_IWDG_Init(void){
  hiwdg.Instance = IWDG;
  hiwdg.Init.Prescaler = IWDG_PRESCALER_32;
  hiwdg.Init.Reload = 4095;
  if (HAL_IWDG_Init(&hiwdg) != HAL_OK) {
    Error_Handler();
  }
}
void Watchdog_Refresh(void){
  HAL_IWDG_Refresh(&hiwdg);
}
unsigned char Is_Watchdog_Reset(void){
  if(counter_for_watchdog > 3){
    return 1;
  }
  return 0;
}
void Watchdog_Counting(void){
  counter_for_watchdog++;
}

void Reset_Watchdog_Counting(void){
  counter_for_watchdog = 0;
}
```

Program 5.17: An implementation of the ‘watchdog’ functions

2.3.11 Reliability and safety implications

- Make sure the task array is large enough
- Take care with function pointers
- Dealing with task overlap

  Suppose we have two tasks in our application (Task A, Task B). We further assume that Task A is to run every second and Task B every three seconds. We assume also that each task has a duration of around 0.5 ms.

  Suppose we schedule the tasks as follows (assuming a 1ms tick interval):

  SCH_Add_Task(TaskA, 0, 1000);

  SCH_Add_Task(TaskB, 0, 3000);

  In this case, the two tasks will sometimes be due to execute at the same time. On these occasions, both tasks will run, but Task B will always execute after Task A. This will mean that if Task A varies in duration, then Task B will suffer from ‘jitter’: it will not be called at the correct time when the tasks overlap.

  Alternatively, suppose we schedule the tasks as follows:

  SCH_Add_Task(TaskA, 0, 1000);

  SCH_Add_Task(TaskB, 5, 3000);

  Now, both tasks still run every 1,000 ms and 3,000 ms (respectively), as required. However, Task A is explicitly scheduled always to run 5 ms before Task B. As a result,Task B will always run on time.

  In many cases, we can avoid all (or most) task overlaps simply by the judicious use of the initial task delays.

2.3.12 Portability

3 Objectives

The aim of this lab is to design and implement a cooperate scheduler to accurately provide timeouts and trigger activities. You should add a file for the scheduler implementation and modify the main system call loop to handle timer interrupts.

4 Problem

- Your system should have at least four functions:

- **void SCH_Update(void)**:This function will be updated the remaining time of each tasks that are added to a queue. It will be called in the interrupt timer, for example 10 ms.
- **void SCH_Dispatch_Tasks(void)**: This function will get the task in the queue to run.
- **uint32_t SCH_Add_Task(void (* pFunction)(), uint32_t DELAY, uint32_t PERIOD)**: This function is used to add a task to the queue. It should return an ID that is corresponding with the added task.
- **uint8_t SCH_Delete_Task(uint32_t taskID)**: This function is used to delete the task based on its ID.

You should add more functions if you think it will help you to solve this problem. Your main program must have 5 tasks running periodically in 0.5 second, 1 second, 1.5 seconds, 2 seconds, 2.5 seconds.

5 Demonstration

You should be able to show some test code that uses all the functions specified in the driver interface.

Specifically set up and demonstrate:

- A regular 10ms timer tick.
- Register a timeout to fire a callback every 10ms.
- Then, print the value returned by get_time every time this callback is received.
- Note: Your timestamps must be at least accurate to the nearest 10ms.
- Register another timeout at a different interval in addition to the 500ms running concurrently (i.e. demo more than one timeout registered at a time).
- Before entering the main loop, set up a few calls to SCH_Add_Task. Make sure the delay used is long enough such that the loop is entered before these wake up. These callbacks should just print out the current timestamp as each delay expires.

Note this is not a complete list. The following designs are considered unsatisfactory:

- Only supporting a single timeout registered at a time.

- Delivering callbacks in the wrong order
- O(n) searches in the SCH_Update function.
- Interrupt frequencies greater than 10Hz, if your timer ticks regularly.

6 Submission

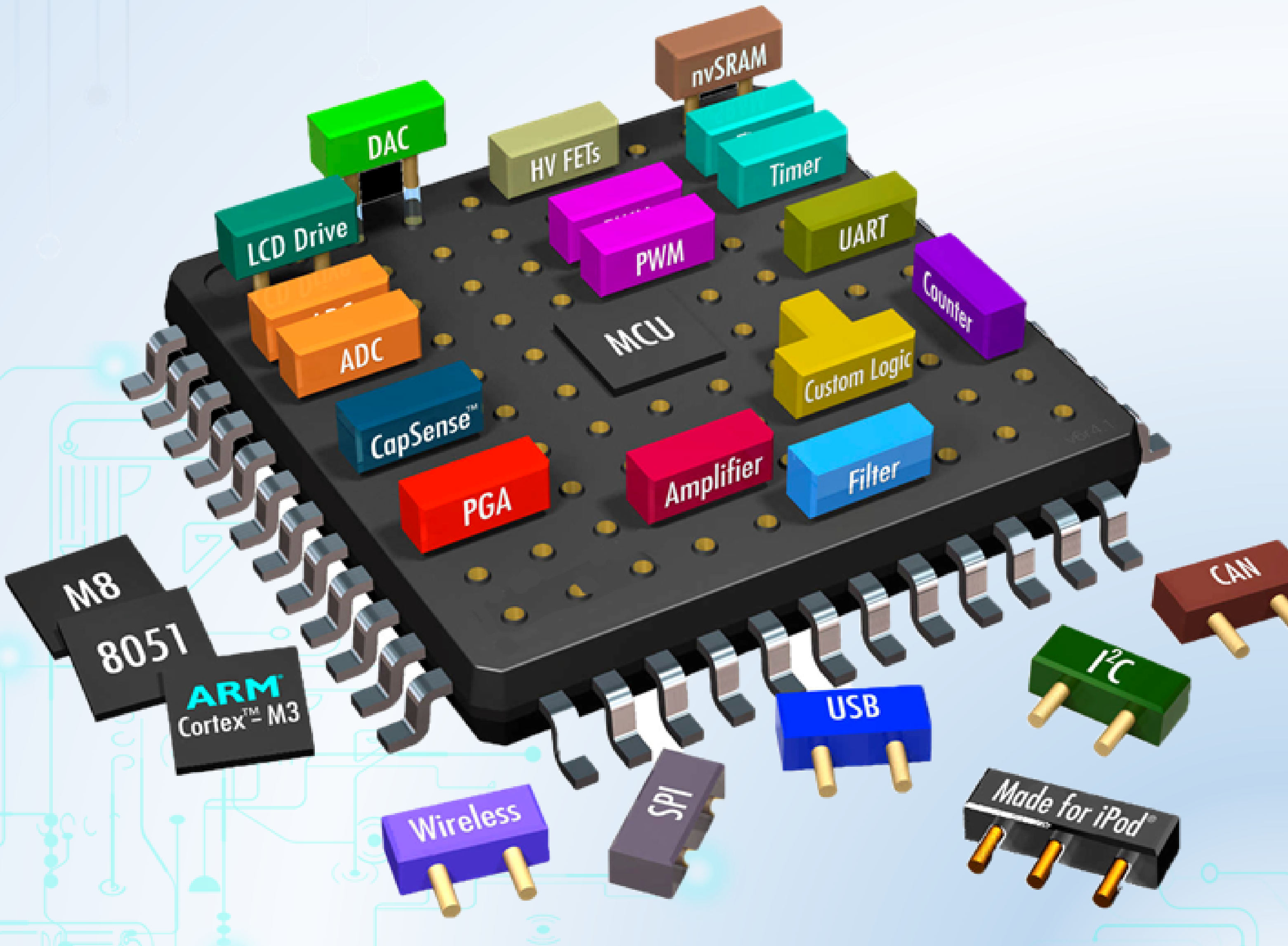
You need to

- Demonstrate your work in the lab class and then
- Submit your source code to the BKeL.

7 References

CHƯƠNG 6

Flow and Error Control in Communication



1 Introduction

Flow control and Error control are the two main responsibilities of the data link layer, which is a communication channel for node-to-node delivery of the data. The functions of the flow and error control are explained as follows.

Flow control mainly coordinates with the amount of data that can be sent before receiving an acknowledgment from the receiver and it is one of the major duties of the data link layer. For most of the communications, flow control is a set of procedures that mainly tells the sender how much data the sender can send before it must wait for an acknowledgment from the receiver.

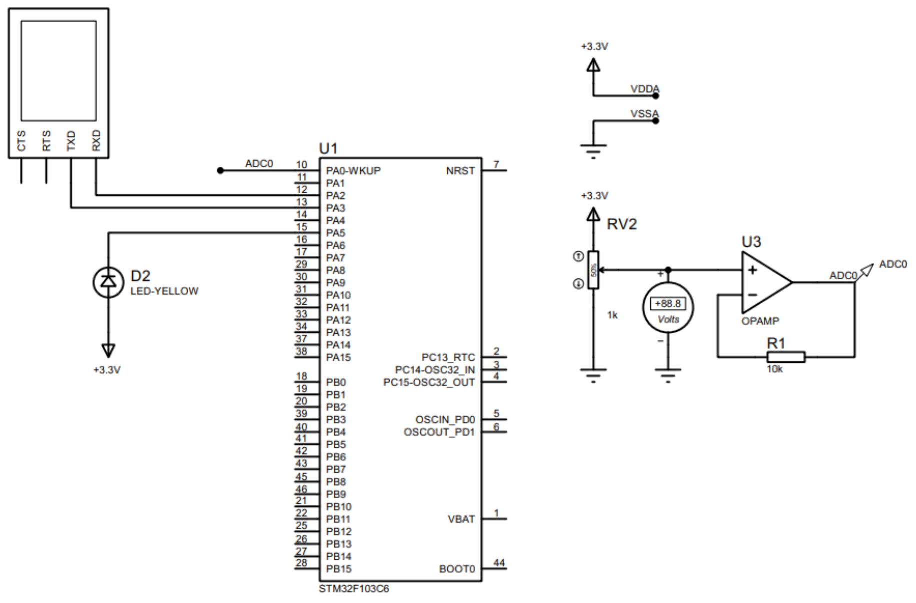
A critical issue, but not really frequently occurred, in the flow control is that the processing rate is slower than the transmission rate. Due to this reason each receiving device has a block of memory that is commonly known as buffer, that is used to store the incoming data until this data will be processed. In case the buffer begins to fill-up then the receiver must be able to tell the sender to halt the transmission until once again the receiver become able to receive.

Meanwhile, error control contains both error detection and error correction. It mainly allows the receiver to inform the sender about any damaged or lost frames during the transmission and then it coordinates with the re-transmission of those frames by the sender.

The term Error control in the communications mainly refers to the methods of error detection and re-transmission. Error control is mainly implemented in a simple way and that is whenever there is an error detected during the exchange, then specified frames are re-transmitted and this process is also referred to as Automatic Repeat request(ARQ).

The target in this lab is to implement a UART communication between the STM32 and a simulated terminal. A data request is sent from the terminal to the STM32. Afterward, computations are performed at the STM32 before a data packet is sent to the terminal. The terminal is supposed to reply an ACK to confirm the communication successfully or not.

2 Proteus simulation platform



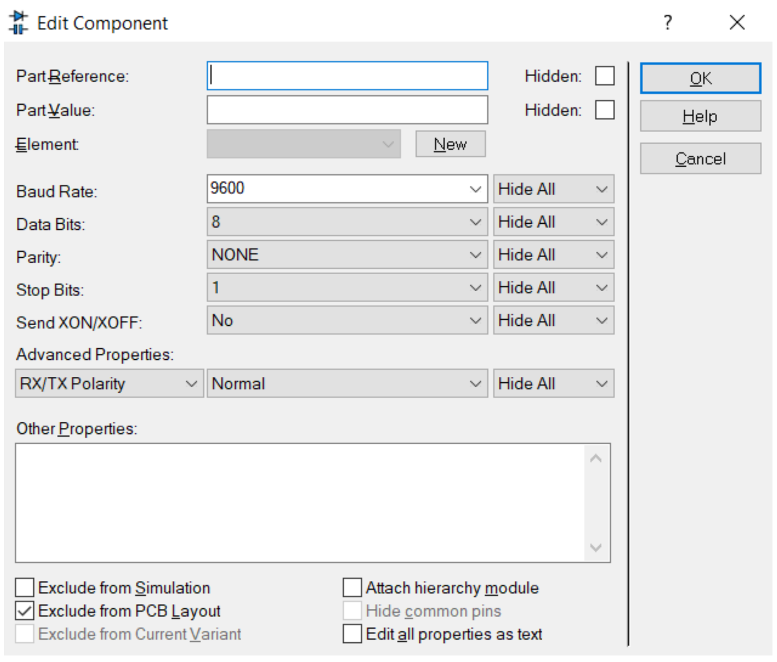
Hình 6.1: Simulation circuit on Proteus

Some new components are listed bellow:

- Terminal: Right click, choose Place, Virtual Instrument, then select VIRTUAL TERMINAL.
- Variable resistor (RV2): Right click, choose Place, From Library, and search for the POT-HG device. The value of this device is set to the default 1k.
- Volt meter (for debug): Right click, choose Place, Virtual Instrument, the select DC VOLTMETER.
- OPAMP (U3): Right click, choose Place, From Library, and search for the OPAMP device.

The opamp is used to design a voltage follower circuit, which is one of the most popular applications for opamp. In this case, it is used to design an adc input signal, which is connected to pin PA0 of the MCU.

Double click on the virtual terminal and set its baudrate to 9600, 8 data bits, no parity and 1 stop bit, as follows:



Part Reference:		Hidden: ☐
Part Value:		Hidden: ☐
Element:		
Baud Rate:	9600	Hide All
Data Bits:	8	Hide All
Parity:	NONE	Hide All
Stop Bits:	1	Hide All
Send XON/XOFF:	No	Hide All
Advanced Properties:		
RX/TX Polarity	Normal	Hide All

Other Properties:

☐ Exclude from Simulation
☑ Exclude from PCB Layout
☐ Exclude from Current Variant
☐ Attach hierarchy module
☐ Hide common pins
☐ Edit all properties as text

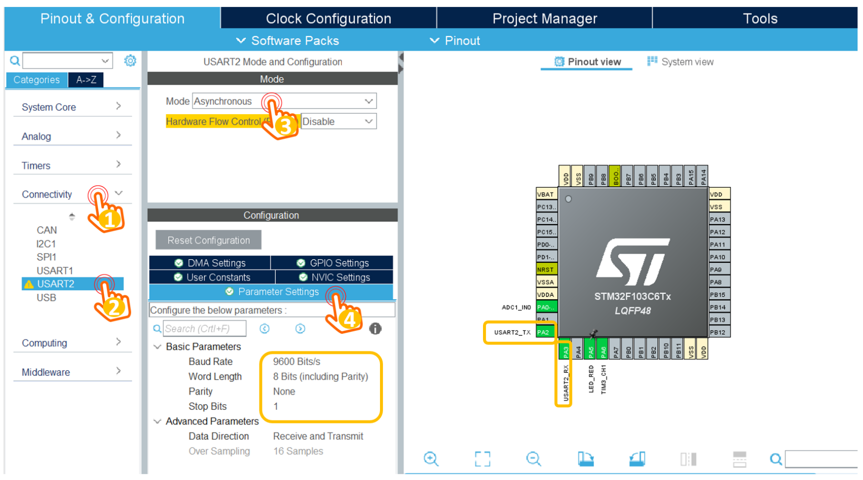
Hình 6.2: Terminal configuration

3 Project configurations

A new project is created with following configurations, concerning the UART for communications and ADC input for sensor reading. The pin PA5 should be an GPIO output, for LED blinky.

3.1 UART Configuration

From the ioc file, select **Connectivity**, and then select the **USART2**. The parameter settings for UART channel 2 (USART2) module are depicted as follows:



USART2 Mode and Configuration

Mode

Mode: Asynchronous
Hardware Flow Control (RS232): Disable

Configuration

Reset Configuration

DMA Settings | GPIO Settings | User Constants | NVIC Settings | Parameter Settings

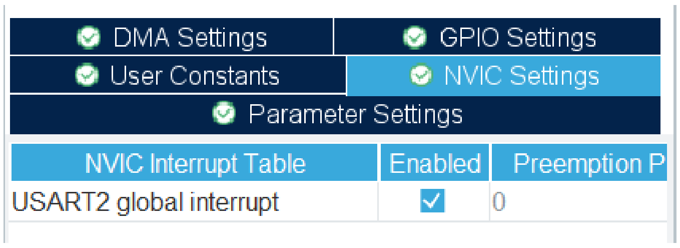
Configure the below parameters :

Basic Parameters	
Baud Rate	9600 Bits/s
Word Length	8 Bits (including Parity)
Parity	None
Stop Bits	1
Advanced Parameters	
Data Direction	Receive and Transmit
Over Sampling	16 Samples

Hình 6.3: UART configuration in STMCube

The UART channel in this lab is the Asynchronous mode, 9600 bits/s with no Parity and 1 stop bit. After the uart is configured, the pins PA2 (Tx) and PA3(Rx) are enabled.

Finally, the NVIC settings are checked to enable the UART interrupt, as follows:

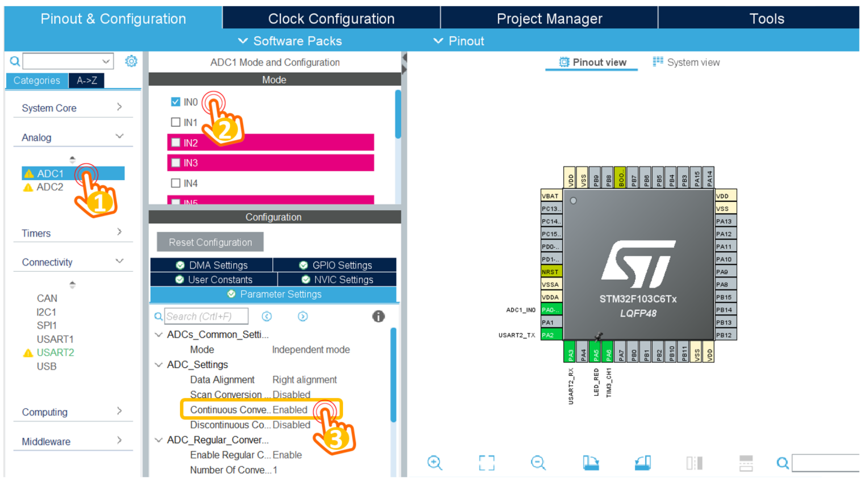


NVIC Interrupt Table	Enabled	Preemption P
USART2 global interrupt	✓	0

Hình 6.4: Enable UART interrupt

3.2 ADC Input

In order to read a voltage signal from a simulated sensor, this module is required. By selecting on **Analog**, then **ADC1**, following configurations are required:



ADCs_Common_Setti...
Mode: Independent mode
ADC_Settings
Data Alignment: Right alignment
Scan Conversion ...: Disabled
Continuous Conve..: Enabled
Discontinuous Co...: Disabled
ADC_Regular_Conver...
Enable Regular C...: Enable
Number Of Conve...: 1

Hình 6.5: Enable UART interrupt

The ADC pin is configured to PA0 of the STM32, which is shown in the pinout view dialog.

Finally, the PA5 is configured as a GPIO output, connected to a blinky LED.

4 UART loop-back communication

This source is required to add in the main.c file, to verify the UART communication channel: sending back any character received from the terminal, which is well-known as the loop-back communication.

```c
/* USER CODE BEGIN 0 */
uint8_t temp = 0;

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart){
  if(huart->Instance == USART2){
    HAL_UART_Transmit(&huart2, &temp, 1, 50);
    HAL_UART_Receive_IT(&huart2, &temp, 1);
  }
}
/* USER CODE END 0 */
```

Program 6.1: Implement the UART interrupt service routine

When a character (or a byte) is received, this interrupt service routine is invoked. After the character is sent to the terminal, the interrupt is activated again. This source code should be placed in a user-defined section.

Finally, in the main function, the proposed source code is presented as follows:

```c
int main(void)
{
  HAL_Init();
  SystemClock_Config();

  MX_GPIO_Init();
  MX_USART2_UART_Init();
  MX_ADC1_Init();

  HAL_UART_Receive_IT(&huart2, &temp, 1);

  while (1)
  {
    HAL_GPIO_TogglePin(LED_RED_GPIO_Port, LED_RED_Pin);
    HAL_Delay(500);
  }

}
```

Program 6.2: Implement the main function

5 Sensor reading

A simple source code to read adc value from PA0 is presented as follows:

```c
uint32_t ADC_value = 0;
while (1)
{
  HAL_GPIO_TogglePin(LED_RED_GPIO_Port, LED_RED_Pin);
  ADC_value =  HAL_ADC_GetValue(&hadc1);
HAL_UART_Transmit(&huart2, (void *)str, sprintf(str, "%d\n"
    , ADC_value), 1000);
  HAL_Delay(500);
}
```

Program 6.3: ADC reading from AN0

Every half of second, the ADC value is read and its value is sent to the console. It is worth noticing that the number ADC_value is convert to ascii character by using the sprintf function.

The default ADC in STM32 is 13 bits, meaning that 5V is converted to 4096 decimal value. If the input is 2.5V, ADC_value is 2048.

6 Project description

In this lab, a simple communication protocol is implemented as follows:

- From the console, user types **!RST#** to ask for a sensory data.
- The STM32 response the ADC_value, following a format **!ADC=1234#**, where 1234 presents for the value of ADC_value variable.
- The user ends the communication by sending **!OK#**

The timeout for waiting the **!OK#** at STM32 is 3 seconds. After this period, its packet is sent again. **The value is kept as the previous packet**.

6.1 Command parser

This module is used to received a command from the console. As the reception process is implement by an interrupt, the complexity is considered seriously. The proposed implementation is given as follows.

Firstly, the received character is added into a buffer, and a flag is set to indicate that there is a new data.

```c
#define MAX_BUFFER_SIZE  30
uint8_t temp = 0;
uint8_t buffer[MAX_BUFFER_SIZE];
uint8_t index_buffer = 0;
uint8_t buffer_flag = 0;
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart){
  if(huart->Instance == USART2){

    //HAL_UART_Transmit(&huart2, &temp, 1, 50);
    buffer[index_buffer++] = temp;
    if(index_buffer == 30) index_buffer = 0;

    buffer_flag = 1;
    HAL_UART_Receive_IT(&huart2, &temp, 1);
  }
}
```

Program 6.4: Add the received character into a buffer

A state machine to extract a command is implemented in the while(1) of the main function, as follows:

```
while (1){
    if(buffer_flag == 1){
        command_parser_fsm();
        buffer_flag = 0;
    }
}
```

Program 6.5: State machine to extract the command

The output of the command parser is to set **command_flag** and **command_data**. In this project, there are two commands, **RTS** and **OK**. The program skeleton is proposed as follows:

```
while (1){
    if(buffer_flag == 1){
        command_parser_fsm();
        buffer_flag = 0;
    }
    uart_communiation_fsm();
}
```

Program 6.6: Program structure

6.2 Project implementation

Students are proposed to implement 2 FSM in seperated modules. Students are asked to design the FSM before their implementations in STM32Cube.