SEBM Report

Borja Rodríguez and Daniel Sevilla

Fall 2016

Abstract

In this report we are going to explain what have we done during the lab sessions of SEBM and will include all of the code realized. All the text will be in English so the transverse attitudes can be evaluated easily.

In all the code given, we will try to use always bit operations instead of arithmetical ones (because we have learned in lab that the second ones could produce problems with signed variables). We will also try to use the faster solution (including atomic assignments if necessary) unless the code becomes difficult to read. We will also write some explanations to some parts of the code that have been optimized but may be difficult to understand to an external observer.

P1 - Eclipse Introduction

This first practice aim was to make us being in touch with the essentials tools of the laboratory. Being able to create projects in Ecliplse and compile them in order to work with the STM32 was important. In this practice also we learned how the micro-controller output ports worked.

In this practice we are told about verifying the plate operation, creating new projects and also it explains us some files that are given to us.

For example there are two pins definitions:

- PAD: Name of a port associated pin. It's value goes from 0 to 15.
- BIT: It corresponds to 2^n where n is the bit number.

We are also taught in how to put an INPUT/OUTPUT active or inactive a bit:

- Activate:

  GPIOD->ODR | = BITn : Puts the n bit of D GPIO ODR regiter on.

  GPIOD->BSRR.W = BITn : Puts the n bit of the D GPIO ODR register on atomically. (For $n < 16$)

  PORTx->BSRR.H.set = BITn : Puts the n bit of x port on.

- Desactivate:

  GPIOD->ODR & = (~BITn) : Puts the n bit of D GPIO ODR register off.

  GPIOD->BSRR.W = BITn : Puts the n-16 bit of the D GPIO ODR register off atomically. (For $n > 16$)

  PORTx->BSRR.H.clear = BITn : Puts the n bit of x port off.

Then we are told to create a new function **void ledSequence(void)** that blinks all LEDs alternatively with 0.5s of light for led. The code is the following:

```
1  void ledSequence(void){
2     while(1){
3         i++;
4         switch(i%4){
5             // Turn on the green, orange, red or blue LED
                  using the BSRR register alternatively
6             case 0: (LEDS_PORT->BSRR.H.set)=BIT(
                  GREEN_LED_PAD);
7             break;
8             case 1: (LEDS_PORT->BSRR.H.set)=BIT(
                  ORANGE_LED_PAD);
9             break;
10            case 2: (LEDS_PORT->BSRR.H.set)=BIT(RED_LED_PAD)
                  ;
11            break;
12            case 3: (LEDS_PORT->BSRR.H.set)=BIT(BLUE_LED_PAD
                  );
13            break;
14            }
15
16        // Wait 500ms
17        SLEEP_MS(500);
18
19        switch(i%4){
20            // Turn on the green, orange, red or blue LED
                  using the BSRR register alternatively
21            case 0: (LEDS_PORT->BSRR.H.set)=BIT(
                  GREEN_LED_PAD);
22            break;
23            case 1: (LEDS_PORT->BSRR.H.set)=BIT(
                  ORANGE_LED_PAD);
24            break;
25            case 2: (LEDS_PORT->BSRR.H.set)=BIT(RED_LED_PAD)
                  ;
26            break;
27            case 3: (LEDS_PORT->BSRR.H.set)=BIT(BLUE_LED_PAD
                  );
28            break;
29            }
30
31        // Wait 200ms
32        SLEEP_MS(200);
33    }
34 }
```

After proving that the code worked correctly, we debugged this code watching that the LED's state were changing in the EmbSys Register tab while watching at the ODR GPIOD register.

Finally, we observed the stack frame window where we could see the threads running and the memory position of the funcion running (for example the **main()** was working at 0x200001F6 which meant that we were at the 502th position in the RAM memory (starts at 0x20000000).

P2 - LCD Access

In this practice we learnt how to access, program and use the 2-rowed-16-charactered LCD based on HD44780 of the laboratory plate.

First, we are asked to search if the static logical levels of the LCD and the microcontroller were compatible and, if so, search the noise margins:

$$NMH = min(VIH - VIHmin) = 2,4 - 2 = 0.4V$$
$$NML = min(VILmax - VIL) = 0.8 - 0.4 = 0.4V$$
$$NM = min(NML, NMH) = 0.4V$$

Then, we are taught about how to configure the system ports, about telling us about the important registers:

- MODER register: We can access to it as GPIOx->MODER

  It has 32 bits for 16 lines so n line is governed by 2n and 2n+1 bits.

  As we want to show characters on the LCD we want all the 7 lines in output mode.

Bit 2n+1	Bit 2n	Funcionalitat
0	0	Entrada (Per defecte després de Reset)
0	1	Sortida
1	0	Funcionalitat alternativa
1	1	Mode analògic

Figure 1: Mode Register table

- OTYPER register: We can access to it as GPIOx->OTYPER

  It has 16 bits for 16 lines so n line is governed by n bit. This register just affects the line when it is programmed as an output line.

  We want the outputs in push-pull mode.

Bit n	Funcionalitat
0	Sortida Push-Pull (Per defecte després de Reset)
1	Sortida en drenador obert

Figure 2: Output Type Regiser table

- OSPEEDER register: We can access to it as GPIOx->OSPEEDER

  It has 32 bits for 16 lines so n line is governed by 2n and 2n+1 bits.

  As LCD is made for low speeds, selecting the 2MHz frequency would help to avoid LCD lines noise.

Bit 2n+1	Bit 2n	Funcionalitat
0	0	2 MHz (Per defecte després de Reset)
0	1	25 MHz
1	0	50 MHz
1	1	100 MHz

Figure 3: Output Speed Register table

- PUPDR register: We can access to it as GIPOx->PUPDR

  It has 32 bits for 16 lines so n line is governed by 2n and 2n+1 bits.

  We don't want neither pull-up nor pull-down .

Bit 2n+1	Bit 2n	Funcionalitat
0	0	No *Pull-up* ni *Pull-down* (Per defecte després de Reset)
0	1	*Pull-up*
1	0	*Pull-down*
1	1	Combinació prohibida

Figure 4: Pull Up or Pull Down Register table

After learning about this configuration, we are told to write the function **lcdGPIOInit(void)** in order to configured the 7 lines associated to LCD as low-level push-pull outputs.

The code is the following:

```
1  //Initializes DB7..DB4, RS, E i BL (PE15...PE9)
2  static void lcdGPIOInit(void){
3      //We are aware that some of the initializations are done
           by baseInit(), however, we want to put them all-
           together in order to show that we know all the
           requirements of the LCD to work in that way.
4          GPIOE->MODER |= BIT18|BIT20|BIT22|BIT24|BIT26|BIT28|
               BIT30;
5          GPIOE->MODER &= (~BIT19)&(~BIT21)&(~BIT23)&(~BIT25)
               &(~BIT27)&(~BIT29)&(~BIT31);
6
7          GPIOE->OTYPER &= (~BIT9)&(~BIT10)&(~BIT11)&(~BIT12)
               &(~BIT13)&(~BIT14)&(~BIT15);
8
9          GPIOE->OSPEEDER &= (~BIT18)&(~BIT20)&(~BIT22)&(~
               BIT24)&(~BIT26)&(~BIT28)&(~BIT30);
10         GPIOE->OSPEEDER &= (~BIT19)&(~BIT21)&(~BIT23)&(~
               BIT25)&(~BIT27)&(~BIT29)&(~BIT31);
11
12         GPIOE->PUPDR &= (~BIT18)&(~BIT20)&(~BIT22)&(~BIT24)
               &(~BIT26)&(~BIT28)&(~BIT30);
13         GPIOE->PUPDR &= (~BIT19)&(~BIT21)&(~BIT23)&(~BIT25)
               &(~BIT27)&(~BIT29)&(~BIT31);
14 }
```

Then, the additional exercise of programming a function named **GPIO_ModePushPull** to work generically is proposed. We have done it and the result is the following:

```
1  void GPIO_ModePushPull(GPIO_TypeDef *port, int32_t line){
2      //For the implementation of this function we take into
           account that some of the initializations are done
           by baseInit()
3          (*(int32_t *) port)=((*(int32_t *) port) & (~BIT(2*
               line+1)))|BIT(2*line);
4  }
```

And the resulting code of **lcdGPIOInit(void)** is:

```
1  //Initializes DB7..DB4, RS, E i BL (PE15...PE9)
2  static void lcdGPIOInit(void) {
3          uint32_t count;
4
5          for(count=9;count<16;count++){
6                  GPIO_ModePushPull(&(GPIOE->MODER),count);
7          }
8
9  }
```

After doing this, we are told to write a function that activates or deactivates the LCD backlight. So, the code is:

```
1  void LCD_Backlight(int32_t on){
2          if(on) (GPIOE->BSRR.H.set) = BIT(LCD_BL_PAD);
3          else   (GPIOE->BSRR.H.clear) = BIT(LCD_BL_PAD);
4   }
```

After that, we tried this code to ensure that it worked correctly, and it did. The backlight turned on and off when we wanted.

To change from an 8 bit writing to a 4 bit writing a special initialization has to be done. To make easy this initialization, the code **LCD_Init** is given to us. However, the writing to the LCD are done by an external function **lcdNibble** that we have to implement.

This function has to write the D7...D4 and RS values and then send a 10us enable pulse (E). So, the code is the following:

```
1  // Send 4 bits to the LCD and generates an enable pulse
2  //      nibbleCmd : Bits 3..0 Nibble to send to DB7...DB4
3  //      RS        : TRUE (RS="1")   FALSE (RS="0")
4  static void lcdNibble(int32_t nibbleCmd,int32_t RS){
5          uint32_t i, var;
6          var = LCD_DB4_BIT;
7
8          //We clear the ODR register and put our command
9          (GPIOE->ODR)=((var-1)&(GPIOE->ODR))|(nibbleCmd<<12);
10
11         if(RS) (GPIOE->BSRR.H.set) = LCD_RS_BIT;
12         else   (GPIOE->BSRR.H.clear) = LCD_RS_BIT;
13
14         DELAY_US(10);
15         (GPIOE->BSRR.H.set) = LCD_E_BIT;
16         DELAY_US(10);
17         (GPIOE->BSRR.H.clear) = LCD_E_BIT;
18         DELAY_US(10);
19 }
```

Maybe the line 9 of the code could look strange. It basically clears all the content in the ODR register from bits higher than the D4 corresponding bit because we want to put our command. Then, we shift 12 positions left (the DB4 bit is 12) our 4-information-bit command and place the information in bits 17...14 as we wanted.

Then, we tried this code with the main given to us. We corroborated that it worked by seeing an 'OK' at the LCD.

After that, we are told to write two functions, **LCD_SendChar** and

LCD_SendString. The first one sends 8-bit characters to the Data Register and the second one has a pointer to a character as an argument and calls **LCD_SendChar** as many times as necessary. So, the code is the following:

```
// Send a character to the LCD at the current position
//     car: Charater to send
void LCD_SendChar(char car){
        char hnum, lnum;

        //we take the first 4 bits and last 4 bits of the
            character
        hnum = car >> 4;
        lnum = car & (1<<4 - 1);

        //we send the high bits and then the low bits
        lcdNibble((uint32_t)hnum,1);
        lcdNibble((uint32_t)lnum,1);

        //we wait to avoid problems
        DELAY_US(40);
 }

// Send a string to the LCD at the current position
//     string: String to send
void LCD_SendString(char *string){
    do{
        LCD_SendChar(string++);
    }while(string != '\0');
}
```

After writing these functions, we tried them and write some text in LCD without problems.

Then, we were told to write 3 more functions: **LCD_ClearDisplay**, **LCD_Config** and **LCD_GotoXY**. The first function clears the LCD and goes to (0,0) position. The second one configures the LCD using 3 parameters (disp activates the display, cursor showrs the cursor and blink starts the blinking). And the third one places the cursor at the position (x,y) given these 2 parameters. So the code is the following:

```
// Clear the LCD and set the cursor position at 0,0
void LCD_ClearDisplay(void){
        lcdNibble(0,0);
        lcdNibble(1,0);
        SLEEP_MS(2);
 }

```

```
// Configures the display
//      If Disp is TRUE turn on the display, if not, turns
    off
//      If Cursor is TRUE show the cursor, if not, hides it
//      If Blink is TRUE turn on blinking, if not, deactivate
     blinking
void LCD_Config(int32_t Disp,int32_t Cursor,int32_t Blink){
        //DB3 is always "1"
        int32_t mode = BIT3 + (Disp<<2) + (Cursor<<1) +
            Blink;
        lcdNibble(0,0);
        lcdNibble(mode,0);
        DELAY_US(40);
 }

// Set the cursor at the given position
//    col: Columnn (0..LCD_COLUMNS-1)
//    row: Row     (0..LCD_ROWS-1)
void LCD_GotoXY(int32_t col,int32_t row){
        uint8_t pos;
        pos = (uint8_t)(64*row + col);

        uint8_t hnum, lnum;

        //DB7 is always "1"
        hnum = (pos+BIT7)>>4;
        lnum = pos & (1<<4 - 1);

        //we send the high bits and then the low bits
        lcdNibble((uint32_t)hnum,0);
        lcdNibble((uint32_t)lnum,0);

        //we wait to avoid problems
        DELAY_US(40);
 }
```

The code is easily explained by the image **LCD Instructions**:

Finally, writing a function able of creating custom characters is proposed as an extension, we have done it and the code is the following:

```
 void LCD_CustomChar(int32_t pos,uint8_t *data){
    uint32_t mode i;

    //We clear the non-usable pos bits
    pos = pos & (1<<3 - 1);
    //DB6 is always activated to set de GCRAM adress
    mode = BIT6 + (pos<<3);

```

Instruction	Code RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Description	Execution Time (max) (when f_{op} or f_{osc} is 270 kHz)
Clear display	0	0	0	0	0	0	0	0	0	1	Clears entire display and sets DDRAM address 0 in address counter.	
Return home	0	0	0	0	0	0	0	0	1	—	Sets DDRAM address 0 in address counter. Also returns display from being shifted to original position. DDRAM contents remain unchanged.	1.52 ms
Entry mode set	0	0	0	0	0	0	0	1	I/D	S	Sets cursor move direction and specifies display shift. These operations are performed during data write and read.	37 µs
Display on/off control	0	0	0	0	0	0	1	D	C	B	Sets entire display (D) on/off, cursor on/off (C), and blinking of cursor position character (B).	37 µs
Cursor or display shift	0	0	0	0	0	1	S/C	R/L	—	—	Moves cursor and shifts display without changing DDRAM contents.	37 µs
Function set	0	0	0	0	1	DL	N	F	—	—	Sets interface data length (DL), number of display lines (N), and character font (F).	37 µs
Set CGRAM address	0	0	0	1	ACG	ACG	ACG	ACG	ACG	ACG	Sets CGRAM address. CGRAM data is sent and received after this setting.	37 µs
Set DDRAM address	0	0	1	ADD	ADD	ADD	ADD	ADD	ADD	ADD	Sets DDRAM address. DDRAM data is sent and received after this setting.	37 µs
Read busy flag & address	0	1	BF	AC	AC	AC	AC	AC	AC	AC	Reads busy flag (BF) indicating internal operation is being performed and reads address counter contents.	0 µs

Figure 5: LCD Instructions

```
9       uint8_t hnum;
10      uint8_t lnum;
11
12      //We pass the thata in each matrix adress
13      for(i=0; i<8; ++i){
14          //We take the first 4 bits and last 4 bits of the
                mode
15          hnum = car >> 4;
16              lnum = car & (1<<4 - 1);
17
18          //We send the high bits and then the low bits
19          lcdNibble((uint32_t)hnum,0);
20          lcdNibble((uint32_t)lnum,0)
21
22          DELAY_US(40);
23
24          //We pass the data
25          lcdNibble(data[i] >> 4,1);
```

```
            lcdNibble(data & (1<<4 - 1),1);

            DELAY_US(40);

                ++mode;
            }

            //return to DDRAM
        lcdNibble(8,0);
        lcdNibble(0,0);
        LCD_GotoXY(0,0);
    }
```

With this code we made a funny gif of a pumping heart moving across the LCD and saying WE love SEBM.

```
 int main(void){
    // Basic initializations
    uint8_t data11[8] = {14,17,16,16,8,4,2,1};
    uint8_t data12[8] = {14,31,31,31,15,7,3,1};
    uint8_t data21[8] = {14,17,1,1,2,4,8,16};
    uint8_t data22[8] = {14,31,31,31,30,28,24,16};
    uint8_t reg1, reg2;
    uint8_t i = 0;
    uint8_t j = 0;
    baseInit();
    LCD_Init();
    LCD_GotoXY(4,0);
    LCD_CustomChar(0,data11);
    LCD_CustomChar(1,data21);
    LCD_CustomChar(2,data12);
    LCD_CustomChar(3,data22);
    while(1){
        LCD_GotoXY(i,0);
        LCD_SendString("We");
        LCD_GotoXY(i+1,1);
        LCD_SendString("SEBM");
        LCD_GotoXY(i+4,0);
        if(j < 2){
                reg1 = 0x00;
                reg2 = 0x01;
        }
        else{
                reg1 = 0x02;
                reg2 = 0x03;
        }
        LCD_SendChar(reg1);
        LCD_SendChar(reg2);
        SLEEP_MS(100);
```

```
34              LCD_GotoXY(i+4,0);
35              LCD_SendChar(reg1);
36              LCD_SendChar(reg2);
37              SLEEP_MS(100);
38              LCD_ClearDisplay();
39              SLEEP_MS(50);
40              i++;
41              if((i+5)%16 == 0)
42                      i = 0;
43              j++;
44              j%=4;
45          }
46      }
```

P3 - 3D Accelerometer

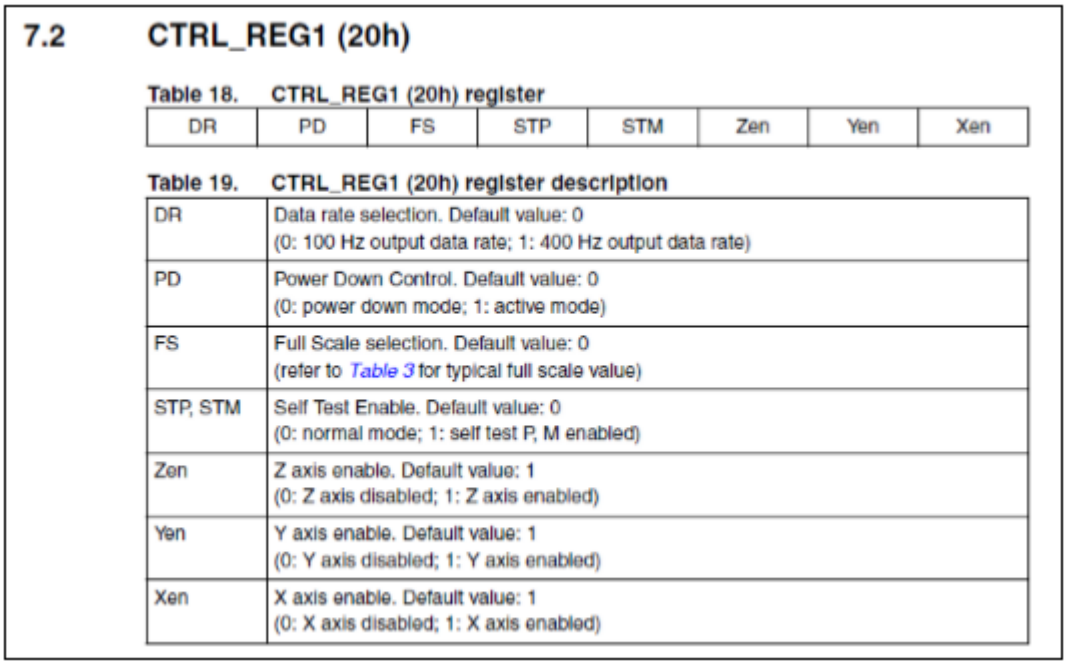
In this practice we are going to learn how to work with the LIS302DL accelerometer of the lab plate and how to use the SPI bus.

We are told about the working mechanism of the accelerometer and also about it's internal registers.

Name	Type	Register address		Default	Comment
		Hex	Binary		
Reserved (Do not modify)		00-0E			Reserved
Who_Am_I	r	0F	000 1111	00111011	Dummy register
Reserved (Do not modify)		10-1F			Reserved
Ctrl_Reg1	rw	20	010 0000	00000111	
Ctrl_Reg2	rw	21	010 0001	00000000	
Ctrl_Reg3	rw	22	010 0010	00000000	
HP_filter_reset	r	23	010 0011	dummy	Dummy register
Reserved (Do not modify)		24-26			Reserved
Status_Reg	r	27	010 0111	00000000	
--	r	28	010 1000		Not Used
OutX	r	29	010 1001	output	
--	r	2A	010 1010		Not Used
OutY	r	2B	010 1011	output	
--	r	2C	010 1100		Not Used
OutZ	r	2D	010 1101	output	

Figure 6: Accelerometer internal registers

We have special interest in understanding the control register:



7.2 CTRL_REG1 (20h)

Table 18. CTRL_REG1 (20h) register

DR	PD	FS	STP	STM	Zen	Yen	Xen

Table 19. CTRL_REG1 (20h) register description

DR	Data rate selection. Default value: 0 (0: 100 Hz output data rate; 1: 400 Hz output data rate)
PD	Power Down Control. Default value: 0 (0: power down mode; 1: active mode)
FS	Full Scale selection. Default value: 0 (refer to *Table 3* for typical full scale value)
STP, STM	Self Test Enable. Default value: 0 (0: normal mode; 1: self test P, M enabled)
Zen	Z axis enable. Default value: 1 (0: Z axis disabled; 1: Z axis enabled)
Yen	Y axis enable. Default value: 1 (0: Y axis disabled; 1: Y axis enabled)
Xen	X axis enable. Default value: 1 (0: X axis disabled; 1: X axis enabled)

Figure 7: Control Register Table

We also have interest in knowing how to enable and disable the clock of our peripheral. All clocks of the microcontroller are governed by the **Reset and cl clock control** (RCC). We have a pointer called **RCC** that points to this structure so we can access to its registers as RCC->REGISTER. For the SPI1 peripheral the register is APB2ENR.

We are told that the SPI1 has a special name **RCC_APB2ENR_SPI1EN**, so we are going to use it instead of BIT12.

The SPI1 block is also pointed by the **SPI1** pointer. The SPI bus works following the MSIO (Master/Slave Input/Output) routine. So in our example, the microcontroller will be the master while the accelerometer will be the slave. The possible inputs of a slave are governed by its Control Slave strobe (nCS). In our example the **CS signal** is connected to **PE3**. The data transmission is made by 2 lines MOSI (PA7) and MISO (PA6). We can also access to the SPI Data Register as SPI-

The clock signal **SCL (PA5)** of the SPI bus is not universal so it is governed by 2 binary signals:

5.3.14 RCC APB2 peripheral clock enable register (RCC_APB2ENR)

Address offset: 0x44

Reset value: 0x0000 0000

Access: no wait state, word, half-word and byte access.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved													TIM11 EN	TIM10 EN	TIM9 EN
													rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reser-ved	SYSCFG EN	Reser-ved	SPI1 EN	SDIO EN	ADC3 EN	ADC2 EN	ADC1 EN	Reserved		USART6 EN	USART1 EN	Reserved		TIM8 EN	TIM1 EN
	rw		rw	rw	rw	rw	rw			rw	rw			rw	rw

Figure 8: Reset and Clock Control Register table

- **CPOL** (Clock Polarity): If CPOL='0' the line is inactive for low-level and if CPOL='1' is inactive for high-level.
- **CPHA** (Clock Phase): If CPAH='0' the data is read at the first clock edge from idle state defined by CPOL. If CPAH='1' the data is read when returning to idle state.

We can access to the SPI Control register as SPI->CR1. It is important to know that the bits CPOL and CPHA has special names **SPI_CR1_CPOL** and **SPI_CR1_CPHA** instead of BIT1 and BIT0.

25.5.1 SPI control register 1 (SPI_CR1) (not used in I^2S mode)

Address offset: 0x00

Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BIDI MODE	BIDI OE	CRC EN	CRC NEXT	DFF	RX ONLY	SSM	SSI	LSB FIRST	SPE	BR [2:0]			MSTR	CPOL	CPHA
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Figure 9: SPI Control Register table

We are told about more configurations of the SPI bus. Knowing that the APB2 is working at 84MHz we can change the transmission rate. We can access to these bits by its special names **SPI_CR1_BR_0 ... SPI_CR1_BR_2**:

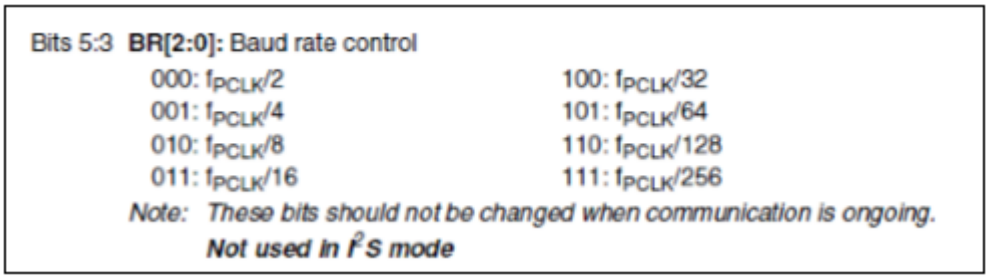


Figure 10: SPI Transmission Rate table

Its also important to know how the data is passed to the accelerometer, we are told that a special register is reserved for this. This regsiter is the **Data Register**, accessible as SPI->DR. As the data is written and read by the same register its important to explicit how this is done:

```
SPI->DR = Data; // Write data to be sent
...             // Wait to end transmission
Data = SPI->DR; // Read received data
```

Now we are told to say the maximum frequency that we can settle in BR that will be compatible with the accelerometer.

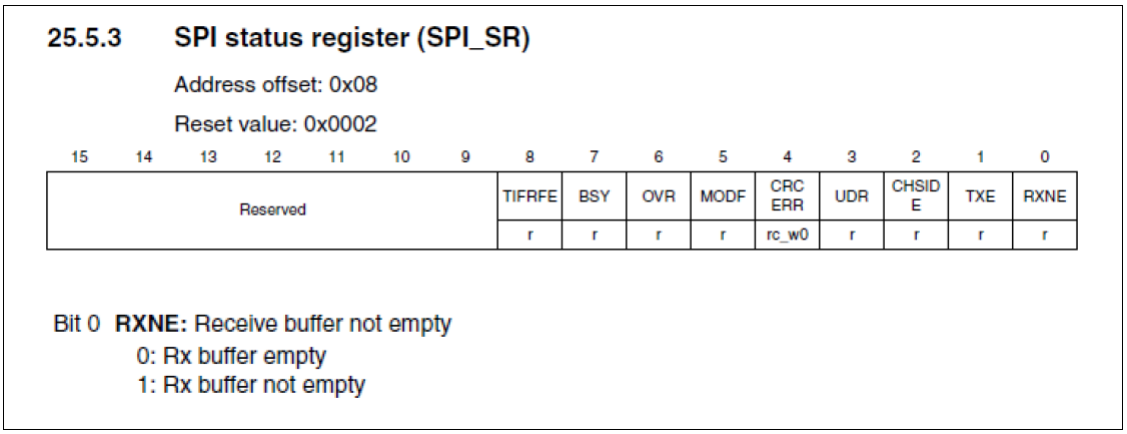
So, as the maximum allowed frequency is 10MHz:

$\frac{84}{(2^x)} < 10 \implies x = 4 \implies f = 5'25MHz \implies BR[2:0] = 011$

After that, we are explained how other bits of the SPI control register work:

- 11. Data frame format (**DFF**): Low for 8-bit data frame and high for 16-bit data frame
- 2. Master selection (**MSTR**): Low for a slave configuration and high for a master configuration.
- 9. Software slave management (**SSM**): Acts like a flag for the software slave management.
- 8. Internal slave select (**SSI**): Force it's value onto the NSS pin when the SSM bit is set.
- 6. SPI enable (**SPE**): Flag for enabling the SPI.

We are also told about the SPI status register:



25.5.3 SPI status register (SPI_SR)

Address offset: 0x08

Reset value: 0x0002

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved							TIFRFE	BSY	OVR	MODF	CRC ERR	UDR	CHSIDE	TXE	RXNE
							r	r	r	r	rc_w0	r	r	r	r

Bit 0 **RXNE:** Receive buffer not empty
0: Rx buffer empty
1: Rx buffer not empty

Figure 11: SPI Status Register

We just care about BIT0 because it who tells us that the buffer is full and the tx is over.

After this explanation we are told to write a function **void initAccel(void)** that initializes the Accelerometer:

```
1  // Init the accelerometer SPI bus
2  void initAccel(void){
3
4          //Enable the SPI peripheric clock of APB2 bus
5          RCC->APB2ENR |= RCC_APB2ENR_SPI1EN;
6
7          //Configure the CR1 register
8          //Configure the CPOL and CPHA registers
9          SPI1->CR1 |= SPI_CR1_CPOL | SPI_CR1_CPHA;
10
11         //Configure the DFF
12         SPI1->CR1 &= (~SPI_CR1_DFF);
13
14         //Configure the BR[2:0]
15         SPI1->CR1 = (SPI1->CR1 | SPI_CR1_BR)&(~SPI_CR1_BR2);
16
17         //Configure the SSM i SSI
18         SPI1->CR1 |= SPI_CR1_SSM | SPI_CR1_SSI;
19
20         //Activate SPE i MSTR in CR1 register
21         SPI1->CR1 |= SPI_CR1_SPE | SPI_CR1_MSTR;
22
23  }
```

Once doing this we are told to see if the initialization of the accelerometer

works correctly and that we use the function **readAccel** given to read the **Who_Am_I** register given (0x3B). The main code stays like this:

```
int main(void){
    // Basic initializations
    char string[20];
    baseInit();
    LCD_Init();
    initAccel();
    LCD_SendString(itoa(readAccel(0x3B,1),string,16));
    while(1)
}
```

And we could see the string in the LCD screen.

Now, we want to activate the accelerometer. For doing that, we have to write onto **CTRL_REG1** (0x20) and change the Power Down Control bit (BIT6) and also enable the X, Y Z axis with Xen, Yen and Zen bits respectively. To do that we have written the optional function **writeAccel(int32_t reg, int32_t data)** that allow us to set data in a specific register. Our aim is to set the CTR_REG1 with '0100111' which is equivalent to 39. So the code is the following:

```
void writeAccel(int32_t reg, int32_t data){
    //security mask
    reg&=0x3F;

    //return if we try to write reserved registers
        if (reg<0x0F) return;
        if ((reg<0x20)&&(reg>0x0F)) return;
        if ((reg<0x27)&&(reg>0x23)) return;
        if ((reg<0x30)&&(reg>0x2D)) return;
        //return if we try to write in read-only registers
            outputs x, y and z
        if ((reg==0x29)||(reg==0x2B)||(reg==0x2D)) return;

        // Activates Chip Select
        LIS_CS_PORT->BSRR.H.clear=LIS_CS_BIT;

        // Send the command
        SPI1->DR=reg;

        // Small delay before reading the buffer
        DELAY_US(2);

        // Wait to the receiver buffer to fill
        while (!((SPI1->SR) & BIT0));

```

```
25            SPI1->DR = data;
26
27            // Wait to the receiver buffer to fill
28            while (!((SPI1->SR) & BIT0));
29
30            // Deactivates Chip Select
31            LIS_CS_PORT->BSRR.H.set=LIS_CS_BIT;
32
33    }
```

And the code to add at the end of **initAccel(void)** is:

```
1   //Activate the accelerometer
2   writeAccel(0x20,39);
```

To end this practice we have done a program that shows us the tilt of the X and Y axis in the LCD board. For doing this we are showing two '*'. One is at the first row of the LCD and the second one at the second. This indicators reveal the tilt of the accelerometer in relative terms (with maximum inclination of 45 degrees). Are always visible and are reinforced against noise with making an average of the last N inclinations (the value is set to 15 but it could be any).

The code is the following:

```
1   void visualInclination(void){
2           int32_t x, y, min, max, posx, posy, antx, anty, N;
3           uint32_t i;
4           N = 100;
5           //Min and max are the values expected for 45 and -45
                degrees
6           min = -39;
7           max = 39;
8           while(1){
9                   x = 0;
10                  y = 0;
11                  //We make the N samples average of x and y
                        to avoid noise
12                  for(i=0; i<N; i++){
13                          x += readAccel(0x29,1);
14                          y += readAccel(0x2B,1);
15                  }
16                  x = x/N;
17                  y = y/N;
18
19                  //We make a data transformation to have a
                        integer value between 0 and 15 being 0
                        the value asociated to -45 degrees and
```

```
                    15 the asociated to 45 degrees. We
                    always avoid going out the range [0,15].
20                  posx = 15*(x - min)/(max - min);
21                  if(posx<0)          posx=0;
22                  else if(posx>15)    posx=15;
23                  posy = 15*(y - min)/(max - min);
24                  if(posy<0)          posy=0;
25                  else if(posy>15)    posy=15;
26
27                  //We erase the (*) characters and set them
                        in their corresponding position. Then we
                         actualize the positions.
28                  LCD_GotoXY(antx,0);
29                  LCD_SendChar(' ');
30                  LCD_GotoXY(anty,1);
31                  LCD_SendChar(' ');
32                  LCD_GotoXY(posx, 0);
33                  LCD_SendChar('*');
34                  LCD_GotoXY(posy, 1);
35                  LCD_SendChar('*');
36                  antx = posx;
37                  anty = posy;
38          }
39  }
```

P4 - Temporizing Measures and Interruptions

In this practices, we learned how to work with the interruptions associated with the GPIO ports and to measure the interruptions response times to verify the SPI bus temporizations. First we are told about the interruptions-dealer hardware element of the ARM-Cortex. The **Nested vectored interrupt controller** (NVIC). The interruptions signal generation calls an interruption service routine asynchronously to the rest of the program that evaluates if it has priority and if so, it executes it stopping the rest of the program while doing so.

To make the interruptions work, we first have to tell the program which GPIO lines can generate interruptions. This is done with the System Configuration Controller (SYSCFG):

- First we need to activate its clock. As the peripheral relies on the peripheral bus APB2 we activate it with the RCC->APB2ENR register. As we saw in practice 3 (figure 8) the SYSCFG bit of the APB2EN register is the 14th. It also has the specific name **RCC_APB2ENR_SYSCFGEN**.
- The selection of the allowed interruption-generating lines is made by multiplexers. Each multiplexers takes the 8 lines of its own number. So if the multiplexer is EXTj it takes the lines PAj ... PHj. The control of this multiplexers is done with the registers EXTICR1 ... EXTICR4 and every register has 4 multiplexers referenced inside having a total of 16 multiplexers.

Then we have to program under which conditions this lines can make interruptions, and this is done with the Esternal Interrupt Controller (EXTI).

- The configuration of the habilitation or masking of the line is done by the Interrupt Mask Register (EXTI_IMR) to which we can access as EXTI->IMR.

- We have also to say if the interruptions are done by rising edge, falling edge or both of them. This is done with the Rising and Falling Trigger Selection Register (EXTI_RSTR and EXTI_FSTR) to which we can acces as EXTI-¿RSTR and EXTI-¿FSTR. The specific names of the bits are EXI_RSTR_TRj being j the selected bit.

- Its also important to know if an interruption has been done or not already. For knowing that we have the Pending Register (PR) accessible as EXTI->PR wich tells us if an interruption has been generated or not. Its bit names are EXTI_PR_PRj being j the selected bit.

Finally, the NVIC is the element that receives the interruptions petitions and decide whether doing them or not in terms of priority. To program an interruption inside the NVIC register we will use the **nvicEnableVector** function: NvicEnableVector(Line,CORTEX_PRIORITY_MASK(Priority)). The parameters line and priority are the following:

EXTI GPIO	Línia	Prioritat	Vector RSI
0	EXTI0_IRQn	STM32_EXT_EXTI0_IRQ_PRIORITY	EXTI0_IRQHandler
1	EXTI1_IRQn	STM32_EXT_EXTI1_IRQ_PRIORITY	EXTI1_IRQHandler
2	EXTI2_IRQn	STM32_EXT_EXTI2_IRQ_PRIORITY	EXTI2_IRQHandler
3	EXTI3_IRQn	STM32_EXT_EXTI3_IRQ_PRIORITY	EXTI3_IRQHandler
4	EXTI4_IRQn	STM32_EXT_EXTI4_IRQ_PRIORITY	EXTI4_IRQHandler
5 a 9	EXTI9_5_IRQn	STM32_EXT_EXTI5_9_IRQ_PRIORITY	EXTI9_5_IRQHandler
10 a 15	EXTI15_10_IRQn	STM32_EXT_EXTI10_15_IRQ_PRIORITY	EXTI15_10_IRQHandler

Figure 12: NVIC parameters table

We also have the possibility of completely disabling an interruption with the **nvicDisableVector(Line)** function.

Then, for the NVIC to work correctly, we have to tell him which function it has to attend. We define an interruption vector associated function with the CH_IRQ_HANDLER(Vector) macro. This has always the format:

```
1  CH_IRQ_HANDLER(Vector){
2
3  CH_IRQ_PROLOGUE();
4  //ISR Code
5  CH_IRQ_EPILOGUE();
6  }
```

Now, we are told to write a progam that changes the green led state and actualize the Y axis of the accelerometer. For doing that we have made both

the **interruptTest(Void)** and the RSI functions. The code is the following:

```
volatile int switchFlag=0;
```

```
 void interruptTest(void){
       //Activate the SYSCFG clock
       RCC->APB2ENR |= RCC_APB2ENR_SYSCFGEN;

       //Assign port A to EXTI0
       SYSCFG->EXTICR[0] &= (~SYSCFG_EXTICR1_EXTI0);

       //Enable the EXTI0 interrupt
       EXTI->IMR |= EXTI_IMR_MR0;

       //Acctivate EXTI0 on rising edge
       EXTI->RTSR |= EXTI_RTSR_TR0;
       EXTI->FTSR &= (~EXTI_FTSR_TR0);

       //Clear the pending interrupt flag at EXTI0
       EXTI->PR = EXTI_PR_PR0;

       //Enable EXTI0 at NVIC
       nvicEnableVector(EXTI0_IRQn,CORTEX_PRIORITY_MASK(
           STM32_EXT_EXTI0_IRQ_PRIORITY));

       char string[20];
       LCD_GotoXY(0,0);
       LCD_SendString("Y = ");

       while(1){
               switchFlag = 0;

               //Waiting for the botton to be pressed
               while(!switchFlag);

               LCD_GotoXY(4,0);
               LCD_SendString("    ");
               itoa(((int32_t) readAccel(0x2B,1)),string
                   ,10);
               LCD_GotoXY(4,0);
               LCD_SendString(string);
               //DELAY_US(40);
       }
}
```

```
CH_IRQ_HANDLER(EXTI0_IRQHandler){

```

```
3     CH_IRQ_PROLOGUE();
4
5     //Erase the pending interrupt flag
6     EXTI->PR = EXTI_PR_PR0;
7
8     //Change the green led status
9     LEDS_PORT->ODR^=GREEN_LED_BIT;
10
11    //Activate the flag so that the accelerometer Y axis is
          read
12    switchFlag = 1;
13
14    CH_IRQ_EPILOGUE();
15
16    }
```

Once we have seen that the code works correctly we are told to measure its temporizing. First we are told to measure the interruption latency (time between an interruption is called and its served). We have concluded that this latency is from 247 ns. After that we are doing the temporal verifications of the SPI bus. The following image is the one used to verify the timing:

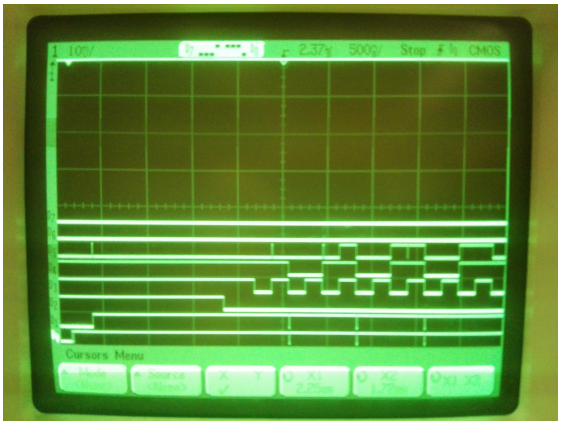


Figure 13: Temporizing of the interruption

Where line 0 is PA0 button, line 1 is PD2 (green LED), line 2 is PE3 (CS), line 3 is PA5 (SCL), line 4 is PA7 (MOSI) and line 5 is PA6 (MISO).

The results are the following:

- Clock frequency : 2.6042MHz
- We have verified that the MOSI line is pointing the accelerometer Y lecture register. At raising edges it shows "101011" that corresponds

to "0x2B". To be sure that te measurement was correct we changed the Y register for the X one and we obtained that it was pointing to "101001" which is "0x29".

- The setup time of the chip select is 540ns >> 5ns as expected.
- The setup time om MOSI is 190ns >> 5ns and the hold time is 190ns >> 15 ns as expected.
- The tv time is 2.4ns << 50ns and the hold time is 190 ns >> 6ns as expected

Now that we are familiarized with the temporizing measures we have done the optional LCD temporizing evaluation. For doing that we will evaluate the HD44780 temporizing.

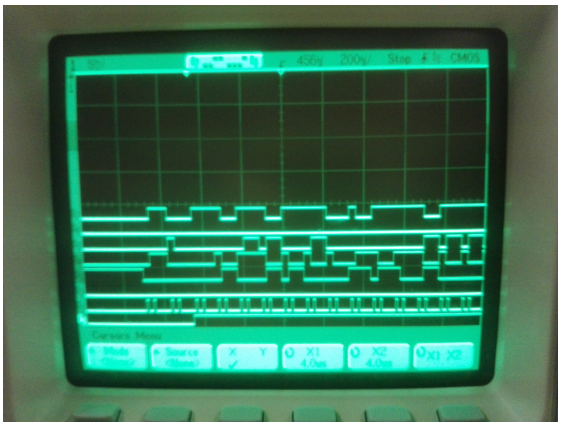


Figure 14: Temporizing of the LCD

Where line 0 i PE11 (RS), line 1 is PE11 (E), line 2 is GND (RW) and lines 3 to 6 are PE12 to PE15 (DB4 to DB7).

The results are the following:

- $t_{as} = 11us > 40ns = t_{as,min}$
- $t_{DSW} = 55us > 90ns = t_{DSW,min}$
- $t_H = 12us > 10ns = t_{H,min}$
- $P_{WEH} = 11us > 230ns = P_{WEH,min}$

C1 - Keyboard Lecture

In this practice we learned how to read a keyboard with a scanning method and also to read the keyboard using interruptions.

First we are told about the keys distribution. They are allocated creating the following matrix:

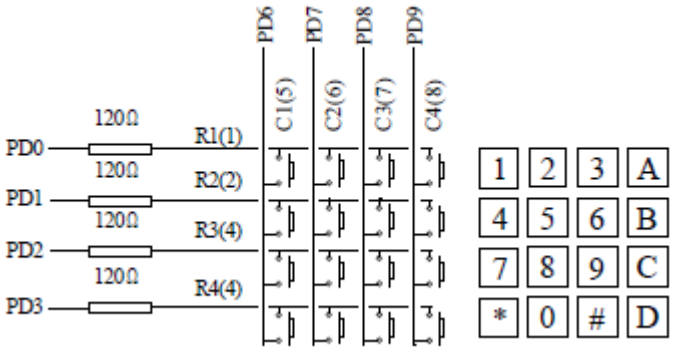


Figure 15: Keys allocation matrix

Allocating the keys in that ways makes that we need just 8 lines in order to read a pressed key. (Its important to notice that this improvement increases as the the number of keys does).

To know which key has been pressed we have to explore the keyboard. The easiest way is to use the rows as open drain output and the columns as pull-up input or vice versa. The pull-up inputs are GPIO normal inputs but their idle state is 1. However, the open drain outputs are 0-forced outputs.

To explore the keyboard we will suppose that we have just a key pressed. We start taking PD0 to '0' and the other outputs PD1 to PD3 to '1'. As the outputs are open drain if there is a '0' in the line it will dominate against the low-1 of the pull-up. When we find a 0 at any line we can conclude that the exploration is over. If its not, we repeat this process with PD1 to PD3.

To make the keyboard exploration we have to read the input bits of the D columns of port D. For doing that we have to acces to the **Input Data Register** (IDR) of the port GPIOD->IDR.

The code that initializes the keyboard is **initKeyboard** and the one that reads the key doing the process wrote before is **readKeyboard**:

```
1   void initKeyboard(void){
2      //We set the PD0 to PD3 (rows) as open drain.
3           (GPIOD->MODER) |= BIT(0)|BIT(2)|BIT(4)|BIT(6);
4           (GIPOD->OTYPER) |= BIT(0)|BIT(1)|BIT(2)|BIT(3);
5           //We set the PD6 to PD9 (columns) as pull-up.
6           (GPIOD->PUPDR) |= BIT(12)|BIT(14)|BIT(16)|BIT(18);
7   }
```

```
1   int32_t readKeyboard(void){
2           //Just to remember it ODR (output) and IDR (input)
3
4           uint32_t j, k;
5
6           //We put all the rows to '1'
7           (GPIOD->ODR) |= BIT(0)|BIT(1)|BIT(2)|BIT(3);
8
9           for(j=0; j<4; j++){
10                  //We prove row by row searching any '0' in a
                        column
11                  (GPIOD->ODR) &= (~BIT(j));
12                  //We wait in order to don't have problems in
                        the reading
13                  DELAY_US(10);
14                  for(k=6; k<10; k++){
15                          //If the column is '0' we set the
                                value
16                          if(!((GPIOD->IDR)&BIT(k))){
17                                  (GPIOD->ODR) |= BIT(j);
18                                  //We return the value with
                                        the codification
                                        proposed
19                                  return(4*j+k-6);
20                          }
21                  }
22                  //If this row does not match, we put it
                        again to '1'
23                  (GPIOD->ODR) |= BIT(j);
24          }
25          return -1;
26  }
```

To be able to show if the keys were really pressed we implemented the function **writeKeyboard** in order to show if any key was pressed and that was being refreshed in intervals of half a second to make a user-confortable experience. The code was the following:

```
void writeKeyboard(void){
        char keys[]={'1','2','3','A','4','5','6','B','7','8'
            ,'9','C','*','0','#','D'};
        int32_t prev = 0;
        while(1){
                SLEEP_MS(500);
                int32_t aux_key = readKeyboard();
                LCD_GotoXY(prev,0);
                LCD_SendChar(' ');
                LCD_GotoXY(aux_key,0);
                LCD_SendChar(keys[aux_key]);
                prev = aux_key;
                int_key = 32;
        }
}
```

Then an optional work is proposed. We should implement the function **readMultikey** that returns a code that puts the key-code high. The codification used is as simple as put the key-pressed bit high (that means that if the keys number 3 and 4 [which key is 3 or 4 is selected arbitrary] the code is 11000) . So the code is the following:

```
int32_t readMultikey(void){
        //Just to remember it ODR (output) and IDR (input)

        int32_t j, k, sum = 0;

        //We put all the rows to '1'
        (GPIOD->ODR) |= BIT(0)|BIT(1)|BIT(2)|BIT(3);

        for(j=0; j<4; j++){
                //We prove row by row searching any '0' in a
                    column
                (GPIOD->ODR) &= (~BIT(j));
                //We wait in order to don't have problems in
                    the reading
                DELAY_US(10);
                for(k=6; k<10; k++){
                        //If the column is '0' we set the
                            value
                        if(!((GPIOD->IDR)&BIT(k))){
                                (GPIOD->ODR) |= BIT(j);
```

```
                                    //We return the value with
                                        the codification
                                        proposed
                                    sum |= BIT(4*j+k-6);
                            }
                    }
                    //If this row does not match, we put it
                        again to '1'
                    (GPIOD->ODR) |= BIT(j);
            }
            return sum;
    }
```

After that we are proposed to handle the interpretation of the keyboard by using interruptions. For doing that we have to take into account that every key pressed can make an interruption. The necessary code for doing this are the **initConfigKeyboard** (configurates the keyboard to kork with interruptions) and the RSI of the keyboard. The code is the following:

```
volatile int32_t int_key=-1
```

```
void intConfigKeyboard(void){

        //Activate the SYSCFG clock
        RCC->APB2ENR |= RCC_APB2ENR_SYSCFGEN;

        //We put all the rows to '0' in order to be able to
            detect
        (GPIOD->ODR) &= (~BIT(0))&(~BIT(1))&(~BIT(2))&(~BIT
            (3));

        //We put as interruption lines 6 to 9
        SYSCFG->EXTICR[1] |= SYSCFG_EXTICR2_EXTI6_PD |
            SYSCFG_EXTICR2_EXTI7_PD;
        SYSCFG->EXTICR[2] |= SYSCFG_EXTICR3_EXTI8_PD |
            SYSCFG_EXTICR3_EXTI9_PD;

        //Enable the EXTI6, EXTI7, EXTI8 and EXTI9 interrupt
        EXTI->IMR |=  BIT(6)|BIT(7)|BIT(8)|BIT(9);
        //Acctivate EXTI6, EXTI7, EXTI8 and EXTI9 on falling
            edge
        EXTI->RTSR &= (~BIT(6))&(~BIT(7))&(~BIT(8))&(~BIT(9)
            );
        EXTI->FTSR |=  BIT(6)|BIT(7)|BIT(8)|BIT(9);

        //Clear the pending interrupt flag at EXTI6, EXTI7,
            EXTI8 and EXTI9
```

```
20          EXTI->PR = BIT(6)|BIT(7)|BIT(8)|BIT(9);
21
22          //Enable EXTI6, EXTI7, EXTI8 and EXTI9 at NVIC
23          nvicEnableVector(EXTI9_5_IRQn,CORTEX_PRIORITY_MASK(
                STM32_EXT_EXTI5_9_IRQ_PRIORITY));
24  }
```

```
1   CH_IRQ_HANDLER(EXTI9_5_IRQHandler){
2
3    CH_IRQ_PROLOGUE();
4
5    //Mask the interruptions
6    (EXTI->IMR) &= (~BIT(6))&(~BIT(7))&(~BIT(8))&(~BIT(9));
7
8    //Read the key
9    %int_key = readKeyboard();
10   int_key = readMultikey();
11
12   //Erase the pending interrupt flag
13   EXTI->PR = BIT(6)|BIT(7)|BIT(8)|BIT(9);
14
15   //We put all the rows to '0' in order to be able to detect
16   (GPIOD->ODR) & (~BIT(0))&(~BIT(1))&(~BIT(2))&(~BIT(3));
17
18    //Enable the EXTI6, EXTI7, EXTI8 and EXTI9 interrupt
19    EXTI->IMR |= BIT(6)|BIT(7)|BIT(8)|BIT(9);
20   }
21
22   CH_IRQ_EPILOGUE();
23   }
```

Now, to be able to show when the keys were pressed, the function writeKeyboard was not working anymore so the new code was:

```
1   void writeKeyboard(void){
2           char keys[]={'1','2','3','A','4','5','6','B','7','8'
                ,'9','C','*','0','#','D'};
3           int32_t prev = 0;
4           while(1){
5                   while(int_key==-1);
6                   int32_t aux_key = int_key;
7                   LCD_GotoXY(prev,0);
8                   LCD_SendChar(' ');
9                   LCD_GotoXY(aux_key,0);
10                  LCD_SendChar(keys[aux_key]);
11                  prev = aux_key;
12                  int_key = -1;
13          }
```

```
}
```

Of course, we needed to write a function to show when more than one key was pressed and that was able to decode the codification done in **readMultikey**. For this reason, the function **writeMultikey** was developed. The code is the following:

```
void writeMultikey(void){
        char keys[]={'1','2','3','A','4','5','6','B','7','8'
            ,'9','C','*','0','#','D'};
        int32_t i = 0;
        while(1){
                while(int_key==-1);
                int32_t aux_key = int_key;
                LCD_ClearDisplay();
                for(i=0;i<16;i++){
                        if((aux_key & BIT(i)) == BIT(i)){
                                LCD_GotoXY(i,0);
                                LCD_SendChar(keys[i]);
                        }
                }
                int_key = -1;
        }
}
```

Finally, we decided to do the optional part of a calculator. Our calculator was able to add, subtract, multiply and divide numbers of any number of digits. The algorithm of the calculator is the following:

- 1. Read a digit from the first number.
- 2. If another number key is pressed, it is understood as another digit of the first number. This goes on until a non-numerical key is pressed. Eg: If I first tap 1 and then tap 2 the resulting number is 12.
- 3. If the ' #' key is pressed, the number changes to be negative. And then return to step 2.
- 4. If the key is A, B, C or D it is understood as addition, subtraction, multiplication or division operation and its respective symbol is shown. Then you cannot go to step 2 anymore.
- 5. Read a digit from the second number.
- 6. If another number key is pressed, it is understood as another digit of the first number. This goes on until a non-numerical key is pressed. Eg: If I first tap 1 and then tap 2 the resulting number is 12.

- 8. Any no-numerical key is understood as an equal demanding. So the result is shown. Then go back to step 1.

We have set a delay between key reading of 150ms just to make a more comfortable user experience while taping the keys.

```
void calculator(void){
        int32_t cont=0;
        int32_t neg=0;
        int32_t num1=1;
        int32_t num2=1;
        int32_t res;
        int32_t op=1;
        int32_t key=-1;
        int32_t delay=150;
        char data[16];
        char keys[]={'1','2','3','A','4','5','6','B','7','8'
            ,'9','C','*','0','#','D'};
        while(1){
                LCD_ClearDisplay();
                LCD_GotoXY(0,0);
                num1=1;
                num2=1;
                //1st number introduction
                do{
                        key=readKeyboard();
                        SLEEP_MS(delay);
                }while(key==-1);
                if(keys[key]>='0' && keys[key]<='9'){
                        num1=keys[key]-'0';
                        itoa(num1,data,10);
                        LCD_SendString(data);
                        cont++;
                }
                while(keys[key]>='0' && keys[key]<='9'){
                        do{
                                key=readKeyboard();
                                SLEEP_MS(delay);
                        }while(key==-1);
                        if(keys[key]>='0' && keys[key]<='9')
                            {
                                num1=10*num1+keys[key]-'0';
                                itoa(num1,data,10);
                                LCD_GotoXY(0,0);
                                LCD_SendString(data);
                                cont++;
                        }
                }
```

```
41
42                      //negative number
43                      while(keys[key]=='#'){
44                              num1*=-1;
45                              cont += ((neg++)%2);
46                              itoa(num1,data,10);
47                              LCD_GotoXY(0,0);
48                              LCD_SendString(data);
49                              LCD_SendChar(' ');
50                              do{
51                                      key=readKeyboard();
52                                      SLEEP_MS(delay);
53                              }while(key==-1);
54                      }
55
56                      LCD_SendChar(' ');
57                      cont++;
58
59                      //operation introduction
60                      while(keys[key]<'A' && keys[key]>'D'){
61                              do{
62                                      key=readKeyboard();
63                                      SLEEP_MS(delay);
64                              }while(key==-1);
65                      }
66
67                      while(keys[key]>='A' && keys[key]<='D'){
68                              do{
69                                      if(keys[key]=='A'){
70                                              LCD_GotoXY(cont,0);
71                                              LCD_SendChar('+');
72                                              op=1;
73                                      }
74                                      else if(keys[key]=='B'){
75                                              LCD_GotoXY(cont,0);
76                                              LCD_SendChar('-');
77                                              op=2;
78                                      }
79                                      else if(keys[key]=='C'){
80                                              LCD_GotoXY(cont,0);
81                                              LCD_SendChar('*');
82                                              op=3;
83                                      }
84                                      else if(keys[key]=='D'){
85                                              LCD_GotoXY(cont,0);
86                                              LCD_SendChar('/');
87                                              op=4;
88                                      }
89                                      else if(keys[key]=='#'){
```

```
90                                          num1*=-1;
91                                          itoa(num1,data,10);
92                                          LCD_GotoXY(0,0);
93                                          LCD_SendString(data)
                                                ;
94                                          cont += ((neg++)%2);
95                                  }
96                                  key=readKeyboard();
97                                  SLEEP_MS(delay);
98                          }while(key==-1);
99                  }
100
101                 //2nd number introduction
102                 //do{
103                 //      key = readKeyboard();
104                 SLEEP_MS(delay);
105                 //}while(key==-1);
106                 if(keys[key]>='0' && keys[key]<='9'){
107                         num2=keys[key]-'0';
108                         itoa(num2,data,10);
109                         LCD_GotoXY(0,1);
110                         LCD_SendString(data);
111                 }
112                 while(keys[key]>='0' && keys[key]<='9'){
113                         do{
114                                 key = readKeyboard();
115                                 SLEEP_MS(delay);
116                         }while(key==-1);
117                         if(keys[key]>='0' && keys[key]<='9')
                                {
118                                 num2=10*num2+keys[key]-'0';
119                                 itoa(num2,data,10);
120                                 LCD_GotoXY(0,1);
121                                 LCD_SendString(data);
122                         }
123                 }
124
125
126                 //after the calculation every key acts as an
                        equal and the
127                 //result is shown
128                 if(op==1){
129                         res=num1+num2;
130                 }
131                 else if(op==2){
132                         res=num1-num2;
133                 }
134                 else if(op==3){
135                         res=num1*num2;
```

```
136                     }
137                     else if(op==4){
138                             res=num1/num2;
139                     }
140
141                     LCD_ClearDisplay();
142                     LCD_GotoXY(0,0);
143                     itoa(res,data,10);
144                     LCD_SendString("= ");
145                     LCD_SendString(data);
146                     cont=0;
147
148                     //we wait until any key is pressed
149                     do{
150                             key = readKeyboard();
151                             SLEEP_MS(delay);
152                     }while(key==-1);
153             }
154
155     }
```

C2 - A/D - Converter

In this practice have learned how to use the A/D converter of the STM32F407. We will use the ADC1, which is a 12-bit resolution A/D converter.

The microcontroller ADC structure is the following, where the supply and positive reference voltage are 3V, and the ground and negative reference voltage are 0V.

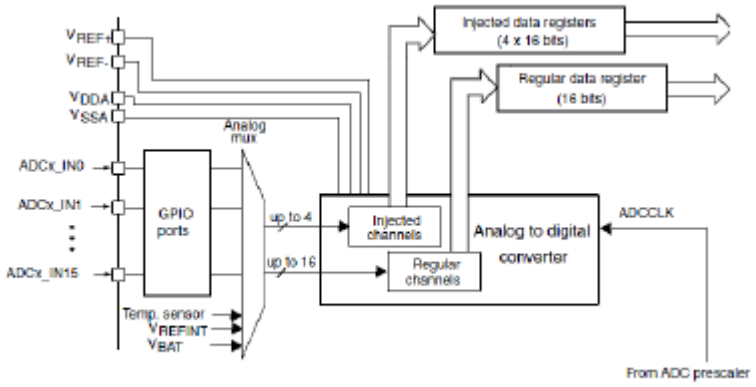


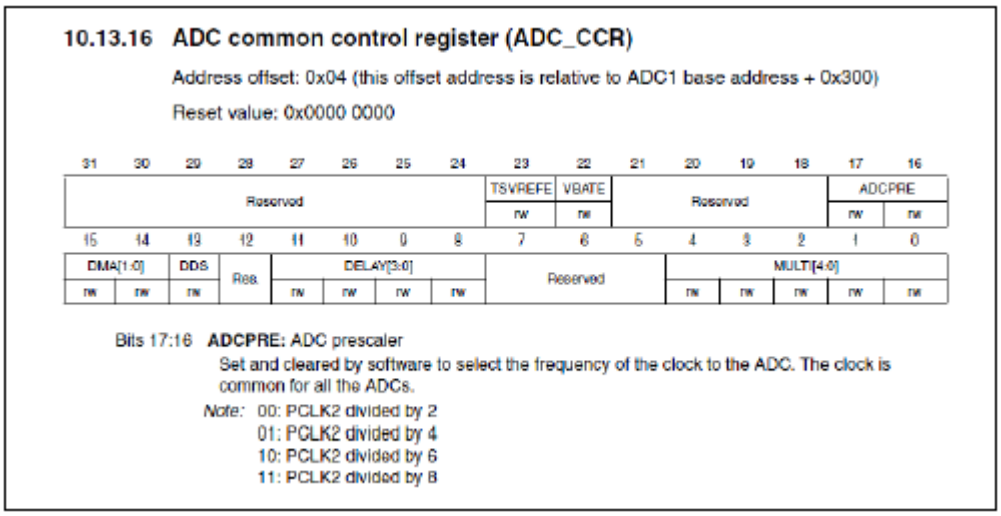
Figure 16: ADC schematic

We have 19 inputs for the ADC:

- 16 are GPIO pins associated that can work as analog inputs
- a temperature sensor measurer (IN16)
- a reference voltage (IN17)
- a backup pile voltae (IN18)

1) To configure the ADC1 is important to activate the peripheric clock. This ADC is held by the APB2 bus so we have to work with the RCC->APB2ENR register but now we have to activate the **ADC1EN** bit.

2) Then we have to fix the clock frequency. The APB2 bus works at 84MHz and is divided by the ADC prescaler. To control this prescaler we have to acces to the **Common Control Register** (CCR), accessible as ADC->CCR.



10.13.16 ADC common control register (ADC_CCR)

Address offset: 0x04 (this offset address is relative to ADC1 base address + 0x300)

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								TSVREFE	VBATE	Reserved				ADCPRE	
								rw	rw					rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DMA[1:0]		DDS	Res.	DELAY[3:0]				Reserved			MULTI[4:0]				
rw	rw	rw		rw	rw	rw	rw				rw	rw	rw	rw	rw

Bits 17:16 **ADCPRE:** ADC prescaler

Set and cleared by software to select the frequency of the clock to the ADC. The clock is common for all the ADCs.

Note: 00: PCLK2 divided by 2
01: PCLK2 divided by 4
10: PCLK2 divided by 6
11: PCLK2 divided by 8

Figure 17: ADC Common Control Register

The ADCPRE bits can be referenced with its special names ADC_CCR_ADCPRE_0 and ADC_CCR_ADPRE_1.

As the frequency of the ADC should be between 30 and 36MHz with 3V supply, we should work with a prescaler of 4 (17th low and 16th bit high).

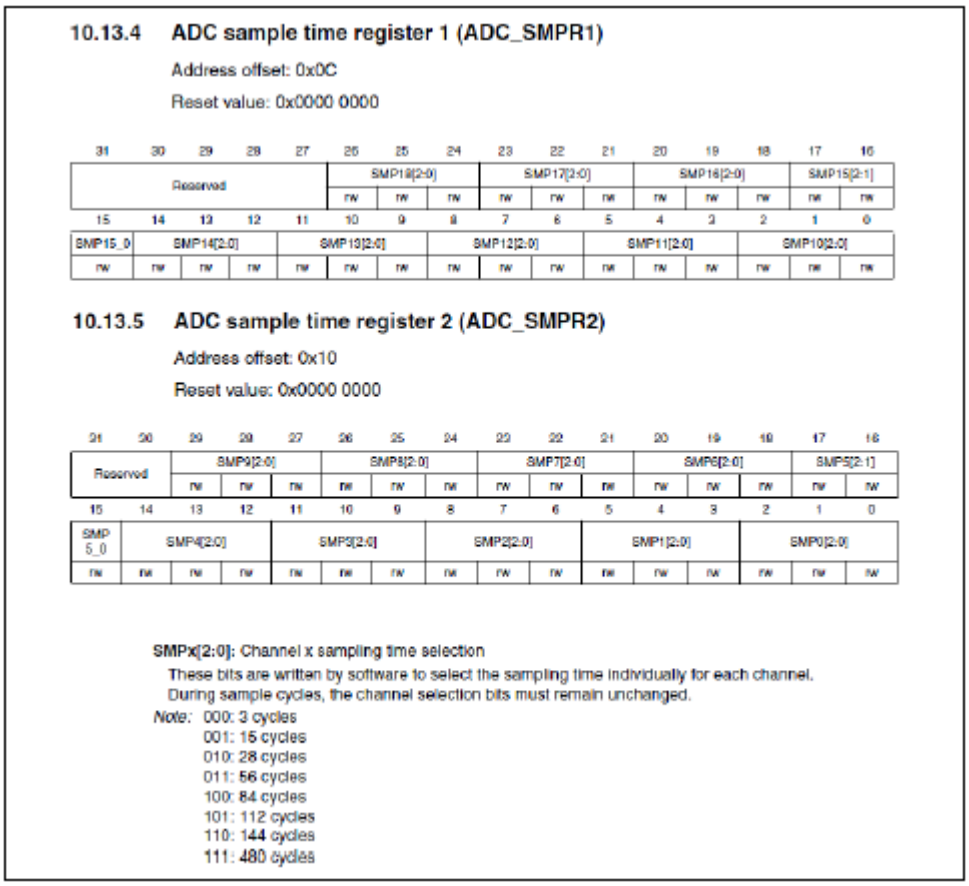
3) Then its important to determine the sample time. This is adjusted by the **ADC Sample Time Registers 1,2** (SMPR1 and SMPR2)

The bits of these registers are accesible as ADC_SMPR1_SMP10, ADC_SMPR1_SMP10_0, ADC_SMPR2_SMP2 and ADC_SMPR1_SMP10_1.

We are told to use the IN8 in this practice so we want to select an appropriate sample time, for doing so, we will use the following ADC connection model and will impose a half of a bit precision.

The time constant at the ADC input will be (in the worst of the scenarios):

$$\tau = R_{max}C = 16120\Omega 4pF = 64.48ns$$



10.13.4 ADC sample time register 1 (ADC_SMPR1)

Address offset: 0x0C

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved					SMP18[2:0]			SMP17[2:0]			SMP16[2:0]			SMP15[2:1]	
					rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SMP15_0	SMP14[2:0]			SMP13[2:0]			SMP12[2:0]			SMP11[2:0]			SMP10[2:0]		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

10.13.5 ADC sample time register 2 (ADC_SMPR2)

Address offset: 0x10

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved		SMP9[2:0]			SMP8[2:0]			SMP7[2:0]			SMP6[2:0]			SMP5[2:1]	
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SMP 5_0	SMP4[2:0]			SMP3[2:0]			SMP2[2:0]			SMP1[2:0]			SMP0[2:0]		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

SMPx[2:0]: Channel x sampling time selection

These bits are written by software to select the sampling time individually for each channel.
During sample cycles, the channel selection bits must remain unchanged.

Note: 000: 3 cycles
001: 15 cycles
010: 28 cycles
011: 56 cycles
100: 84 cycles
101: 112 cycles
110: 144 cycles
111: 480 cycles

Figure 18: ADC Sample Time Registers

The sample time is:

$$\frac{V_{DD}}{2^{13}} = V_{DD}e^{-\frac{T_S}{\tau}} \implies T_S = 581ns$$

As we have see, the clock period is $T_{clk} = 47.62ns$ and therefore the number of cycles are $\lceil \frac{T_S}{T_{clk}} \rceil = 13$. And the superior number of cycles available was 15. However, as we wanted to be sure of the correct behaviour of the system we also tried to put 28 cycles and we observed that the performance of the system was better. For this reason we kept the value of 28 cycles.

4) Then, its important to configure the GPIO PB0 as an analog input. This is done as we did in P2.

5) Finally, its mandatory to activate the ADC. It is done by puting the ADON (BIT0) of the **ADC Control Register 2** (CR2) accessible as

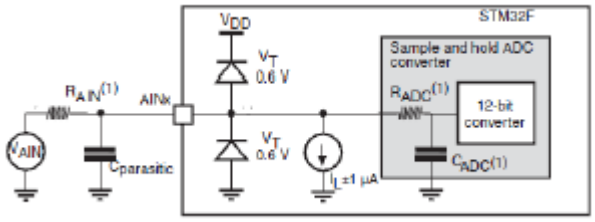


Figure 19: ADC connection model

ADC1->CR2.

So, this configuration is done with the following code:

```
void initADC(void){
        //Activate the peripheric clock.
        //ADC1EN is APB2ENR 8th bit.
        (RCC->APB2ENR) |= RCC_APB2ENR_ADC1EN;

        //For 3V of supply the frequency is between 30 and
            36MHz.
        //As fpclk = 84MHz we should work with fpclk/4.
            Which means that
        //we need the value of ADCPRE = 01 (16th and 17th
            bits).
        (ADC->CCR) |= BIT16;
        (ADC->CCR) &= (~BIT17);

        //Lecture time to ensure a stable value before
            converting it.
        (ADC1->SMPR2) = ((ADC1->SMPR2)|BIT25)&(~BIT24)&(~
            BIT26);         //8th channel 010 -> 28 CYCLES
        (ADC1->SMPR1) |= BIT20|BIT19|BIT18;
                            //16th channel (more than 10us)
        (ADC1->SMPR1) |= BIT23|BIT22|BIT21;
                            //17th channel (more than 10us)

        //Activate temperature sensor (TSVREFE: BIT23)
        (ADC->CCR) |= BIT23;

        //Activate analog mode (GPIOB->MODER = 11)
        //Disconnect it's push-pull (GPIOB->OTYPER =
            0000000000000000)
        (GPIOB->MODER) |= BIT0|BIT1;
        (GPIOB->OTYPER) &= (~BIT0)&(~BIT2)&(~BIT3)&(~BIT4)
            &(~BIT5)&(~BIT6)&(~BIT7)&(~BIT8);
```

```
24          (GPIOB->OTYPER) &= (~BIT9)&(~BIT10)&(~BIT11)&(~BIT12
                )&(~BIT13)&(~BIT14)&(~BIT15);
25
26          //Activate the ADC (ADC->CR2 ADON: BIT0)
27          (ADC1->CR2) |= BIT0;
28  }
```

As we see, the lines 14, 15 and 18 have nothing to be with what we have told before. But will make sense in a while.

To read...

6) The conversions regular sequence is programmed with the **ADC Regular Sequence Registers 1,2,3** (SQR1, SQR2 and SQR3).

The first register SQR1 accessible as ADC1->SQR1 is a 4-bit length register where the regular sequence length is selected. As we just want a conversion we will fix it as default 0000.

The SQ1 champ of the SQR3 register, accessible as ADC1->SQR3 is where we will put the number of the channel to convert.

7) Then we have to activate the **SWSTART** bit of the ADC Control Register 2 (CR2). This is the BIT30.

8) To know when the conversion is over we can read the **ADC Status Register** (SR) accessible as ADC1->SR. So, to read the result we will wait until the EOC bit high. This is the BIT1.

The code to configure the lecture of the IN8 is:

```
1   int32_t readIN8(void){
2           //Activate the 8 channel conversion (ADC1->SQR3 :
                01000)
3           (ADC1->SQR3)=((ADC1->SQR3)|BIT3)&(~BIT4)&(~BIT2)&(~
                BIT1)&(~BIT0);
4
5           //Start conversion (ADC1->CR2 SWSTART: BIT30)
6           (ADC1->CR2) |= BIT30;
7
8           //Wait until the conversion ends (ADC->SR EOC = 1)
9           while(((ADC1->SR)&BIT1)==0);
10
11          //return value
12          int32_t res = (ADC1->DR);
13          return res;
14  }
```

Then, is proposed to do (optionally) a more general function (**readChannel**) to be able to read any channel.

```
1   int32_t readChannel(int32_t channel){
```

```
2           //Activate the chosen channel
3           (ADC1->SQR3)=((ADC1->SQR3)&(~BIT3)&(~BIT4)&(~BIT2)
                &(~BIT1)&(~BIT0))^channel;
4
5           //Start conversion (ADC1->CR2 SWSTART: BIT30)
6           (ADC1->CR2) |= BIT30;
7
8           //Wait until the conversion ends (ADC->SR EOC = 1)
9           while(((ADC1->SR)&BIT1)==0);
10
11          //return value
12          int32_t res = (ADC1->DR);
13          return res;
14  }
```

Now we are told to read the temperature with the ADC. We are able to do that using the IN16. The temperature response is the following:

$$T(^oC) = 25^o + \frac{V_{sense} - V_{25}}{Avgslope}$$

To do this we have to adjust the IN16 lecture channel as 10us or more (this corresponds to the 14 ine of initADC. We also have to activate the temperature sensor of the Common Control Register using the **TSVREFE** bit, which is BIT23 (this corresponds to the line 18 of initADC).

Then we are told to write the function **readT** that reads the temperature. The code is the following:

```
1   int32_t readT(void){
2           int32_t T, temp, Vs;
3           temp = readChannel(16);
4           //Knowing that Vdd = 3V and the ADC values go from 0
                to 4095
5           //We convert both Vs and T to be able to show the
                result
6           //We multiply for 10000 and then divide for 100 to
                avoid errors and work
7           //with integers (for example because V25 = 0.76V)
8           Vs=30000*temp/4095;
9           T=250000+400*(Vs-7600);
10          return T/1000;
11  }
```

To show the results, the code in the main function was:

```
1   int main(void){
2       // Basic initializations
3       char string[10];
4       baseInit();
```

```
5       LCD_Init();
6       initADC();
7       while(1){
8           LCD_ClearDisplay();
9           int32_t val, dec;
10          val = readT();
11          dec = val\%10;
12          val = val/10;
13          LCD_SendString(itoa(val, string, 10));
14          LCD_SendChar(',');
15          LCD_SendString(itoa(dec, string, 10));
16          SLEEP_MS(200);
17      }
18  }
```

Finally, we want to make precise conversions, for this reason we are told to use the IN17 (connected to a reference voltage) to use this voltage to make a function that helps us to read this voltage. For doing this we also need to set a sample time limit of 10 us (line 15 of the initADC). The code was the following:

```
1   int32_t readVdd(void){
2           //We use the typical reference voltage 1.21V
3           int32_t Vdd;
4           Vdd=1210*4096/readChannel(17);
5           return Vdd;
6   }
```

And the code for the visualization was:

```
1   int main(void){
2       // Basic initializations
3       char string[10];
4       baseInit();
5       LCD_Init();
6       initADC();
7       while(1){
8           LCD_ClearDisplay();
9           int32_t val, dec;
10          val = readVdd();
11          dec = val\%1000;
12          val = val/1000;
13          LCD_SendString(itoa(val, string, 10));
14          LCD_SendChar(',');
15          LCD_SendString(itoa(dec, string, 10));
16          LCD_SendString(" V");
17          SLEEP_MS(200);
18      }
```

```
}
```

Finally, an optional adjustment is demanded. We are told to use the readVdd function to calibrate the voltage measure and have a better implementation of the readT function. To do this we have to change a little bit the code, the change is:

```
int32_t readT(void){
        int32_t T, temp, Vs;
        temp = readChannel(16);
        //Knowing that Vdd = 3V and the ADC values go from 0 to 4095
        //We convert both Vs and T to be able to show the result
        //We multiply for 10000 and then divide for 100 to avoid errors and work
        //with integers (for example because V25 = 0.76V)
        Vdd = readVdd()*1000;
        Vs=Vdd*temp/4095;
        T=250000+400*(Vs-7600);
        return T/1000;
}
```

C3 - Encoder

In this practice we will learn how to use a quadrature encoder to give input values to the system. We will also use the interruption RSI routines as in practice 4.

A quadrature encoder is a rotary gadget that is able to record relative rotation movements. The encoder gives a pulse response when it rotates so we can determine the rotation angle by counting the pulses.

The operation of the encoder is well specified with the following two figures:

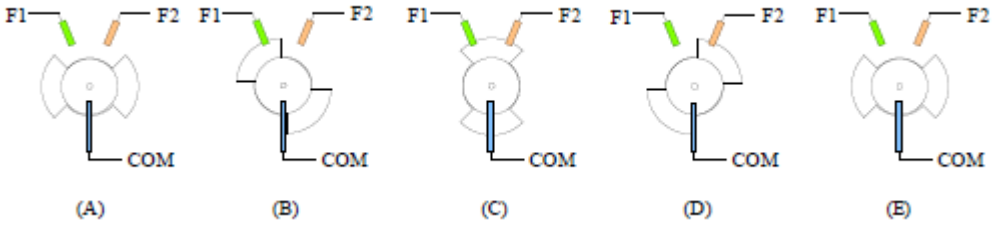


Figure 20: Encoder circuit commutation

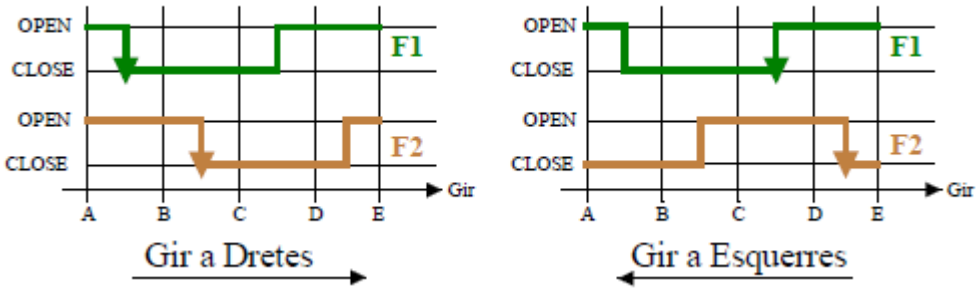


Figure 21: Encoder rotation response

It's important to clarify that the encoder has 24 stop points and that the outputs of it are connected to PA1 and PA2 through two 120 ohms resistances. The encoder only forces the '0' so we will need **pull-up** GPIO inputs.

Then we are told to write a code that does the following:

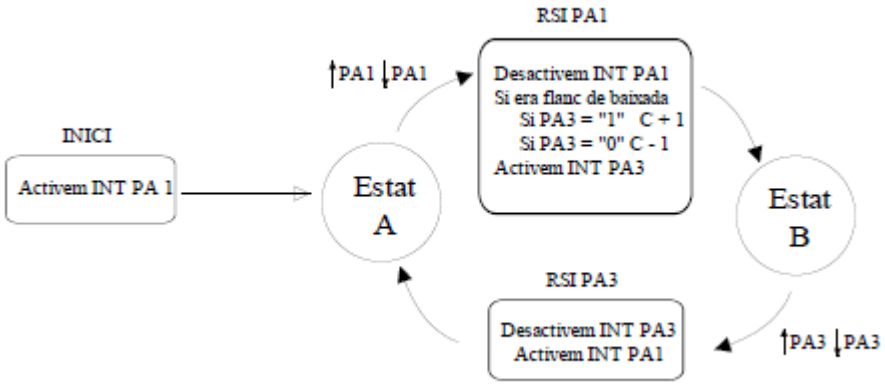


Figure 22: Encoder code schematic

So, first, we have to initialize and prepare the encoder interruptions:

```
1  #include "Base.h"      // Basic definitions
2  #include "mcuconf.h"
3  #include "encoder.h"
4  #include "lcd.h"
5
```

```
6  volatile int32_t count = 0;
7
8  void initEncoder(void){
9          //GPIO PA1 i PA3 com entrades digitals amb Pull-Up
10         GPIOA->MODER &= (~BIT2)&(~BIT3)&(~BIT6)&(~BIT7);
11         GPIOA->PUPDR |= BIT2|BIT6;
12         GPIOA->PUPDR &= (~BIT3)&(~BIT7);
13
14         //Activate the SYSCFG clock
15         RCC->APB2ENR |= RCC_APB2ENR_SYSCFGEN;
16
17         //Assign port PA1 and PA3 to EXTI1 and EXTI3
18         SYSCFG->EXTICR[0] &= (~SYSCFG_EXTICR1_EXTI1)&(~
               SYSCFG_EXTICR1_EXTI3);
19
20         //Configure EXTI1 on rising and falling edge
21         EXTI->RTSR |= EXTI_RTSR_TR1|EXTI_RTSR_TR3;
22         EXTI->FTSR |= EXTI_FTSR_TR1|EXTI_RTSR_TR3;
23
24         //Clear the pending interrupt flag at EXTI1 and
               EXTI3
25         EXTI->PR = EXTI_PR_PR1|EXTI_PR_PR3;
26
27         //Enable ONLY the EXTI1 interrupt
28         EXTI->IMR |= EXTI_IMR_MR1;
29
30         //Enable EXTI1 and EXTI3 at NVIC
31         nvicEnableVector(EXTI1_IRQn,CORTEX_PRIORITY_MASK(
               STM32_EXT_EXTI1_IRQ_PRIORITY));
32         nvicEnableVector(EXTI3_IRQn,CORTEX_PRIORITY_MASK(
               STM32_EXT_EXTI3_IRQ_PRIORITY));
33 }
34
35 CH_IRQ_HANDLER(EXTI1_IRQHandler){
36  CH_IRQ_PROLOGUE();
37
38  //Disable the EXTI1 interrupt
39  EXTI->IMR &= (~EXTI_IMR_MR1);
40
41  //Erase the pending interrupt flag
42  EXTI->PR = EXTI_PR_PR1;
43
44  if(((GPIOA->IDR)&BIT1)!=BIT1){
45          if(((GPIOA->IDR)&BIT3)==BIT3) count++;
46          else count--;
47  }
48
49  //Enable the EXTI3 interrupt
50  EXTI->IMR |= EXTI_IMR_MR3;
```

```
51
52    //Clear the pending interrupt flag at EXTI3
53    EXTI->PR = EXTI_PR_PR3;
54
55    CH_IRQ_EPILOGUE();
56    }
57
58   CH_IRQ_HANDLER(EXTI3_IRQHandler){
59    CH_IRQ_PROLOGUE();
60
61    //Disable the EXTI3 interrupt
62    EXTI->IMR &= (~EXTI_IMR_MR3);
63
64    //Erase the pending interrupt flag
65    EXTI->PR = EXTI_PR_PR3;
66
67    //Enable the EXTI1 interrupt
68    EXTI->IMR |= EXTI_IMR_MR1;
69
70    //Clear the pending interrupt flag at EXTI1
71    EXTI->PR = EXTI_PR_PR1;
72
73    CH_IRQ_EPILOGUE();
74    }
```

And the main program to show what we are doing is:

```
1    int main(void)
2     {
3     // Basic initializations
4     char string[11];
5     baseInit();
6     LCD_Init();
7     initEncoder();
8     extern volatile int32_t count;
9     while(1){
10            LCD_ClearDisplay();
11            LCD_SendString(itoa(count, string, 10));
12            SLEEP_MS(200);
13       }
14    }
```

R1 - ChibioOS/RT Introduction

In this practice we will learn how to create different simultaneous process with ChibiOS/RT. First we are told about what a OS is and what a RTOS is. We will skip this introduction because it is perfectly explained in the Practice Manual.

In our microcontroller, when we run the OS there is only a process and it has a **NORMALPRIO** priority. Every additional process will be its child. Its important to know that we will fix priorities relative to the main program priority. All priorities must rely between **LOWPRIO** and **HIGHPRIO**.

The creation of a child process is done as follows:

```
chThdCreateStatic(wsp, size, priority, thread_function, arg)
    ;
```

Where:

- wsp : Pointer to a working area dedicated to the thread stack. E.g:

```
    static WORKING_AREA(wsp,128);
```

- size : Working area size. E.g:

```
    size = sizeof(wsp)
```

- priority : Priority of the new process.
- $\text{thread}_{function}$: $Function of the new process. E.g$:

```
    static msg_t thread_function(void *arg)
```

- arg : pointer to an argument passed to the tread function. It can be NULL.

Then we are given a code (**test2threads**) that turns on the blue led and waits 300ms until it turns it off. Meanwhile another thread blinks the orange led wit 500ms delay instead.

Some experiments are proposed to us. First, we change the child process priority to **NORMALPRIO+1** which makes the child process more important making the blue led not blinking because all the microcontroller is working on the child thread. After that we do the same thing but with **NORMALPRIO-1**, this makes an inverse effect. Now, only the blue led is blinking. Another proposal is to change the **busyWait()** function calls for calls to the **SLEEP_MS()** macro. This macro calls the **chThdSleep** function that takes out the control process from the CPU during some clock cycles which allows other process to run while the prior one is sleeping. If we exchange this functions we don't really see many difference because the program was working all right before, but we are aware that with more process and complexity in the program, this change could improve the code behaviour.

Then we are asked to generate a function that blinks the blue, orange, green and red leds with different delays. Also an optional part is proposed, it is said to try to use only one function for all the different leds and take advantage of the last parameter of the **chThdCreateStatic** function. The resulting code is:

```
1  / Working area for the childs threads
2  static WORKING_AREA(waChild1,128);
3  static WORKING_AREA(waChild2,128);
4  static WORKING_AREA(waChild3,128);
5
6  static msg_t thChild(void *arg);
7
8  void test4threads(void) {
9   // Basic system initialization
10  baseInit();
11
12  // Creation of the void pointer to run different process
        with the same function
13  uint32_t x1, x2, x3;
14  x1=1;
15  x2=2;
16  x3=3;
17  void* thread1 = &x1;
18  void* thread2 = &x2;
19  void* thread2 = &x3;
```

```
20
21    // Child thread creation
22    chThdCreateStatic(waChild1, sizeof(waChild1), NORMALPRIO,
          thChild, thread1);
23    chThdCreateStatic(waChild2, sizeof(waChild2), NORMALPRIO,
          thChild, thread2);
24    chThdCreateStatic(waChild3, sizeof(waChild3), NORMALPRIO,
          thChild, thread3);
25
26    while (TRUE){
27            // Turn on blue LED using BSRR
28            (LEDS_PORT->BSRR.H.set)=BLUE_LED_BIT;
29
30            // Pause
31            SLEEP_MS(300);
32
33            // Turn off blue LED using BSRR
34            (LEDS_PORT->BSRR.H.clear)=BLUE_LED_BIT;
35
36            // Pause
37        SLEEP_MS(300);
38     }
39    }
40
41
42  // Child threads that bliks the orange, green and red LEDs
43  static msg_t thChild(void *arg) {
44        if(*(int *) arg==1){
45         while (TRUE) {
46           // Turn on orange LED using BSRR
47           (LEDS_PORT->BSRR.H.set)=ORANGE_LED_BIT;
48
49           // Pause
50           SLEEP_MS(500);
51
52           // Turn off orange LED using BSRR
53           (LEDS_PORT->BSRR.H.clear)=ORANGE_LED_BIT;
54
55           // Pause
56           SLEEP_MS(500);
57           }
58        } else if (*(int *) arg==2){
59         while (TRUE) {
60           // Turn on green LED using BSRR
61           (LEDS_PORT->BSRR.H.set)=GREEN_LED_BIT;
62
63           // Pause
64           SLEEP_MS(400);
65
```

```
66            // Turn off gren LED using BSRR
67            (LEDS_PORT->BSRR.H.clear)=GREEN_LED_BIT;
68
69            // Pause
70            SLEEP_MS(400);
71            }
72        } else if (*(int *) arg==3){
73         while (TRUE) {
74            // Turn on red LED using BSRR
75            (LEDS_PORT->BSRR.H.set)=RED_LED_BIT;
76
77            // Pause
78            SLEEP_MS(700);
79
80            // Turn off red LED using BSRR
81            (LEDS_PORT->BSRR.H.clear)=RED_LED_BIT;
82
83            // Pause
84            SLEEP_MS(700);
85            }
86        }
87        return 0;
88    }
```

With this code we see how the different leds are blinking with different frequencies.

R2 - Mutual Elements Sharing

In this practice we will learn about the exclusive-usable common elements. To be able to arbitrate the access to the common elements we will define and use the **Mutex** element-synchronization.

In ChibiOS/RT we have synchronization elements as **BinarySemafore**. The principal functions of these semafors are:

- **void chBSemInit (BinarySemaphore *pbsm, taken)** : Initializes the associated variable to a semaphore saying if its initial state is free or taken.
- **msg_t chBSemWait(BinarySemaphore *pbsm)** : Tries to take a semaphore. If its free, the process will continue, if its not, the process will wait until the semaphore gets free.
- **void chBSemSignal(BinarySemaphore *pbsm)** : It leaves the semaphore free again.

To see if we understand how the semaphores work (and after being given an example) we are told to write a code with 2 semaphores that blinks orange led at every iteration and also blinks the blue led every 2 iterations. The code that does it is the following:

```
1  // Binary semaphores
2  BinarySemaphore   semaf;
3  BinarySemaphore   semaf2;
4
5  // Working area for the semaphore child thread
6  static WORKING_AREA(waSem,128);
7  static WORKING_AREA(waSem2,128);
8
```

```
9  // Child threads function prototypes
10 static msg_t thSem(void *arg);
11 static msg_t thSem2(void *arg);
12
13 // Process syncronization example that uses semaphores
14 void semaphoreExample(void) {
15  // Global initialization
16  baseInit();
17
18  // Initializes the semaphore as not taken
19  chBSemInit(&semaf,FALSE);
20  chBSemInit(&semaf2,FALSE);
21
22  // Creates child threads
23  chThdCreateStatic(waSem1, sizeof(waSem1), NORMALPRIO+1,
        thSem, NULL);
24  chThdCreateStatic(waSem2, sizeof(waSem2), NORMALPRIO+1,
        thSem2, NULL);
25
26  while(1) {
27      SLEEP_MS(300);          // Wait 300ms
28      chBSemSignal(&semaf);   // Send signal to child
29      }
30  }
31
32
33 // Child thread that changes the state of the orange LED
34 // at each semaphore syncronization
35 static msg_t thSem(void *arg) {
36  int32_t led2= 0;
37  while(1) {
38          chBSemWait(&semaf);    // Wait for syncronization
39      (LEDS_PORT->ODR)^=ORANGE_LED_BIT; // Toggle orange LED
40      led2 = (led2+1)%2;
41          if(!led2) chBSemSignal(&semaf2);
42  }
43
44  return 0;
45 }
46
47 static msg_t thSem2(void *arg) {
48  while(1){
49          chBSemWait(&semaf2);   // Wait for syncronization
50          (LEDS_PORT->ODR)^=BLUE_LED_BIT; // Toggle blue LED
51      }
52
53  return 0;
54  }
```

We are told about the dangers of deadlocks and priority inversion problems due to bad synchronising. For this reason, is better to use **Mutexes** instead of semaphores to control the access to resources. The principal functions of mutexes are:

- **void chMtxInit (Mutex *pmtx)** : Initializes a free-state mutex
- **void chMtxLock (Mutex *pmtx)** : Tries to take a mutex. If its free, the process will continue, if its not, the process will wait until the mutex gets free.
- **Mutex * chMtxUnlock(Mutex *pmtx)** : It leaves the last mutex taken by the process.

Now, we are given a code called **mutexExample** where 2 processes try to write at LCD (the main program writes the first LCD line and the child tries to write in the second). Of course, they both try to write some-when at the same time and some glitches are shown in the LCD screen due to the bad resources access control (in fact there is not any access control).

To fix this problem, we are told to arbitrate the access to LCD with the **Mutex** method. The code is the following:

```
1  // Working area for the child thread
2  static WORKING_AREA(waThEM,128);
3
4  // Child thread function prototype
5  static msg_t thMtx(void *arg);
6
7  // Test where two threads access to the LCD
8  // The main thread write digits 0..9 on the first LCD line
9  // The child thread write letters A..Z on the second LCD
       line
10 void mutexExample(void){
11  int32_t x,i,car;
12
13  // Global initialization
14  baseInit();
15
16  // Initialize the LCD
17  LCD_Init();
18
19  // Clear the LCD and turn off the backlight
20  LCD_ClearDisplay();
21  LCD_Backlight(0);
22
```

```
23    //Mutex initialization
24    chMtxInit(&mutex_1);
25
26    // Creates a child thread
27    chThdCreateStatic(waThEM, sizeof(waThEM), NORMALPRIO,thMtx,
          NULL);
28
29    // First digit to show
30    car='0';
31
32    // Infinite loop
33    while (1){
34       // 20 times for each digit
35       for(i=0;i<20;i++){
36          for(x=0;x<LCD_COLUMNS;x++){
37                // For each column on the LCD...
38                // Reserve
39                     chMtxLock(&mutex_1);
40                     // Jump to that column
41                     LCD_GotoXY(x,0);
42                     // Write the digit
43                     LCD_SendChar(car);
44                     // Set free
45                chMtxUnlock();
46                // Some delay
47                     DELAY_US(17000);
48                }
49           // Go to next digit
50       if (++car>'9') car='0';
51       }
52    }
53
54  // Child thread entry point
55  // Write letters on the second row of the LCD
56  static msg_t thMtx(void *arg){
57   int32_t x,i,car;
58
59   // First char to show
60   car='A';
61
62   // Infinite loop
63   while (1){
64     // 20 times for each char
65     for(i=0;i<20;i++){
66        for(x=0;x<LCD_COLUMNS;x++){
67              // For each LCD column
68              // Reserve
69                  chMtxLock(&mutex_1);
70                  // Jump to that column
```

```
71                  LCD_GotoXY(x,1);
72                  // Write the char
73                  LCD_SendChar(car);
74                  // Set free
75                  chMtxUnlock();
76                  // Some delay
77                  DELAY_US2(10000);
78              }
79            // Go to next char
80        if (++car>'Z') car='A';
81      }
82    }
```

A1 - Inter-Plates Communication

As we finished the optional and mandatory practices before finishing all the established lab sessions we decided to try a communication between the plates with our lab-mates that had also finished.

For doing so, we used the EXT1 communications wire that was given to us which implemented the following connections:

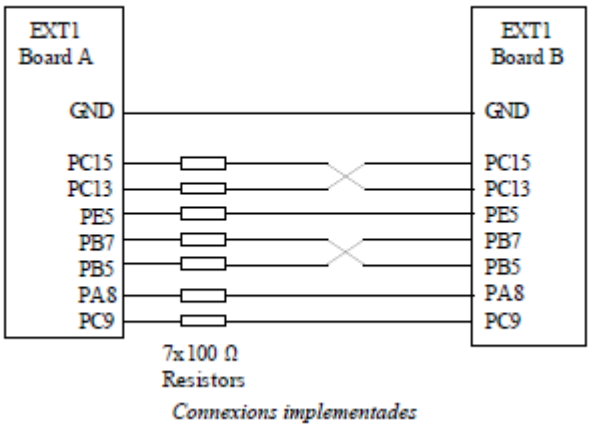


Figure 23: Implemented EXT1 wire connections

As we had to choose between one of the communications we decided to use the synchronous full-duplex communication. Also, as implementing a communication protocol can be a little bit difficult, we decided to opt to a easy-readable code instead of a high-efficient one this time. For this reason, we made a clarifing header file:

```
/*
 * comms.h
 *
 *  Full duplex serial synchronous communication
 *
 */

#ifndef COMMS_H_
#define COMMS_H_

#define COMMS_PORT GPIOC    // Communications port
#define CLK_PORT GPIOB      // Clocks port

#define TX_PIN    15
#define RX_PIN    13

#define CLK_TX     7
#define CLK_RX     5

#define MAX_LENGTH    256 // Maximum length of a message (256
     bits)

#define T    1000  // Clk period (in us)

void initComms(void); // Initialization of the board to
    communicate between boards

void sendMessage(char *string); // Send a string to the
    receiver


#endif /* COMMS_H_ */
```

Thanks to that, it will be easier to understand the following code, that implements the full-duplex communication:

```
/*
 * comms.c
 *
 *  Created on: 29/04/2016
 *      Author: marc.de.cea
 */


#include "Base.h"      // Basic definitions
#include "comms.h"
#include "lcd.h"

static void GPIO_ModePushPull(GPIO_TypeDef *port, int32_t
```

```
    linia, int32_t speed){
14          // 1st step: modify MODER register to set OUTPUT
                mode
15          (port->MODER) = ((port->MODER) & (~BIT(2*linia + 1))
                ) | (BIT(2*linia));
16
17          // 2nd step: modify OTYPER register to select PUSH-
                PULL mode
18          (port->OTYPER) = (port->OTYPER)&(~BIT(linia));
19
20          // 3rd step: modify OSPEEDR register to set the
                working speed
21          switch(speed){
22                  case (2):
23                   (port->OSPEEDR) = (port->OSPEEDR) & (~BIT
                         (2*linia)) & (~BIT(2*linia + 1));
24                   break;
25
26                  case (25):
27                   (port->OSPEEDR) = ((port->OSPEEDR) & (~BIT
                         (2*linia + 1))) | (BIT(2*linia));
28                   break;
29
30                  case (50):
31                   (port->OSPEEDR) = ((port->OSPEEDR) | (BIT
                         (2*linia + 1))) & (~BIT(2*linia));
32                   break;
33
34                  default:
35                   (port->OSPEEDR) = (port->OSPEEDR) | (BIT(2*
                         linia)) | (BIT(2*linia + 1));
36          }
37
38
39          // 4th step: modify the PUPDR register to set no
                pull-up/pull-down resistors
40          (port->PUPDR) = (port->PUPDR) & (~BIT(2*linia)) & (~
                BIT(2*linia + 1));
41
42
43          // 5th step: set the output to '0' (modify ODR
                register)
44          (port->BSRR.H.clear) = BIT(linia);
45   }
46
47
48  static void GPIO_ModeInput(GPIO_TypeDef *port, int32_t linia
        ){
49          // 1st step: modify MODER register to set INPUT mode
```

```
50          (port->MODER) = ((port->MODER) & (~BIT(2*linia + 1))
                ) & (~BIT(2*linia));
51
52          // 2nd step: modify the PUPDR register to no pull up
                 nor pull down
53          (port->PUPDR) = ((port->PUPDR) & (~BIT(2*linia))) &
                (~BIT(2*linia + 1));
54
55   }
56
57  // Working area for the child threads
58  static WORKING_AREA(clkThread,128);
59
60  // Binary semaphore to sinchronize tx_clk and bit
        transmition
61  BinarySemaphore   semafC;
62
63  // Variable to tell to the clock generation Thread to change
        status
64  volatile int32_t clk_generate = 0;
65
66  static void clkGeneration(void){
67
68          while (1){
69
70                  if (clk_generate){
71
72                          // Genarte the clock by toggling
                                between high and low values
73                          CLK_PORT->ODR = CLK_PORT->ODR ^ BIT(
                                CLK_TX);
74
75                          if ( ((CLK_PORT->ODR >> CLK_TX ) & 1
                                )== 0){
76                                  // It is a falling edge -->
                                        The data has to be
                                        transmitted
77                                  // Wake up the sending
                                        thread with the
                                        semaphore
78                                  chBSemSignal(&semafC);
79
80                          }
81
82
83                          DELAY_US(T);
84
85                  } else {
86
```

```
87                          // In the idle state, CLK_TX is high
88                          CLK_PORT->ODR = CLK_PORT->ODR | BIT(
                                CLK_TX);
89                  }
90          }
91
92  }
93
94  void initComms(){
95
96          // Initilaize the LCD
97          LCD_Init();
98
99          // Child thread creation. This child thread
                generates the clock in transmission mode
100         chThdCreateStatic(clkThread, sizeof(clkThread),
                NORMALPRIO, clkGeneration, NULL);
101
102         // TX and CLK-TX have to be configured as Push-Pull
                output
103         GPIO_ModePushPull(COMMS_PORT, TX_PIN, 0);
104         GPIO_ModePushPull(CLK_PORT, CLK_TX, 0);
105
106         // RX and CLK_RX as inputs with no pull-up nor pull-
                down
107         GPIO_ModeInput(COMMS_PORT, RX_PIN);
108         GPIO_ModeInput(CLK_PORT, CLK_RX);
109
110         // Configure interruption of RX_CLK and RX pin
111         // Activate SYSCFG clock
112         (RCC->APB2ENR) = (RCC->APB2ENR |
                RCC_APB2ENR_SYSCFGEN);
113
114         //EXTI13: Assign the port B (clk port) to the
                external interrupt line EXTI5. With the & we are
                 erasing the 4 bits that were
115         // previously indicating which port was assigned,
                and with the | we are
116         // indicating the new assigned port
117         (SYSCFG->EXTICR[3]) = (SYSCFG->EXTICR[3] & (~
                SYSCFG_EXTICR4_EXTI13)) |
                SYSCFG_EXTICR4_EXTI13_PC;
118
119         // Enable interruption lines
120         (EXTI->IMR) = (EXTI->IMR) | EXTI_IMR_MR13;
121
122         // Enable interrupts of RX_Pin at both edges (for
                synchronization)
123         (EXTI->RTSR) = (EXTI->RTSR) | EXTI_RTSR_TR13;
```

```
124             (EXTI->FTSR) = (EXTI->FTSR) | EXTI_FTSR_TR13;
125
126             // Erase any pending interrupt flag
127             (EXTI->PR) = EXTI_PR_PR13;
128
129             // Enable the vector that manages the interruptions
                    of line 13 (RX_PIN)
130             nvicEnableVector(EXTI15_10_IRQn ,
                    CORTEX_PRIORITY_MASK(
                    STM32_EXT_EXTI10_15_IRQ_PRIORITY));
131
132             // Set intial values of TX and CLK-TX
133             CLK_PORT->ODR = CLK_PORT->ODR | BIT(CLK_TX);
134             COMMS_PORT->ODR = COMMS_PORT->ODR | BIT(TX_PIN);
135
136             // Initializes the semaphore as not taken
137             chBSemInit(&semafC,FALSE);
138
139     }
140
141     // Send a string to the receiver
142     void sendMessage(char *string){
143
144
145             // Indicate the start of a message by creating a
                    falling edge before the
146             // clk generation
147             COMMS_PORT->ODR = COMMS_PORT->ODR & (~BIT(TX_PIN));
148
149             // Activate the clk modfying clk_generate
150             clk_generate = 1;
151
152             char send; // char to send at a given moment
153
154             int32_t i;
155
156             DELAY_US(100);
157
158             // Send the message
159             while (*string){
160
161                     // Send the char bit by bit
162                      send = *string++;
163
164
165                      for (i = 0; i<8; i++){
166
167                              chBSemWait(&semafC);
168                              if ( ((send>>i)&1) == 0){
```

```
169                                     COMMS_PORT->ODR = (
                                            COMMS_PORT->ODR & (~BIT
                                            (TX_PIN)));
170                             } else {
171                                     COMMS_PORT->ODR = (
                                            COMMS_PORT->ODR | BIT(
                                            TX_PIN));
172                             }
173
174                     }
175             }
176
177             // Send the EOL sign (0xFF)
178             for (i = 0; i<8; i++){
179
180                     chBSemWait(&semafC);
181                     COMMS_PORT->ODR = (COMMS_PORT->ODR | BIT(
                            TX_PIN));
182
183             }
184
185             // Deactivate the clk
186             clk_generate = 0;
187
188             // Indicate the end of the message with a rising
                    edge after the clk deactivation
189             COMMS_PORT->ODR = COMMS_PORT->ODR | BIT(TX_PIN);
190
191     }
192
193     /********************* ISR FUNCTIONS
            *****************************
194     Those functions should never be called directly or
            indirectly from
195     the main program. They are automatically called by the NVIC
196     interrupt subsystem.
197     ***************************************************************
            */
198
199
200     // Global volatile variable to store the received message
201     int8_t received_message[MAX_LENGTH];
202     volatile int8_t *current_pos = received_message;
203
204
205     // Process the received message and show it on the LCD
            screen
206     static void processMessage(void){
207
```

```
208             char message[MAX_LENGTH/8];
209             char aux;
210 
211             // Iterate over every char in the string and write
                    it using
212             // the function LCD_SendChar
213 
214             // Put the pointer at the initial position of the
                    array
215             current_pos = received_message;
216 
217             // We know that the message has finished when the
                    value if the int8 in the
218             // array is greater than 2
219             int32_t counter_char = 0;
220             int32_t counter_string = 0;
221 
222             // Our EOL sign is 11111111 = 0xFF
223             while (aux != 0xFF){
224 
225                     aux = aux | (*current_pos<<counter_char);
226             counter_char = (counter_char + 1)%8;
227 
228             if (counter_char == 0){
229                     message[counter_string] = aux;
230                     aux = 0;
231                     counter_string++;
232             }
233 
234                     current_pos++;
235              }
236 
237             // We have to remove the EOL sign
238             message[counter_string] = 0;
239 
240             LCD_SendString(message);
241 
242     }
243 
244     // EXTI5 ISR
245     // Associated to a receiving message
246     CH_IRQ_HANDLER(EXTI9_5_IRQHandler) {
247 
248        CH_IRQ_PROLOGUE();
249 
250        // Start of the ISR code
251 
252        // Disable the interrupt
253        (EXTI->IMR) = (EXTI->IMR & ~EXTI_IMR_MR5);
```

```
254
255     // Clear all the interrupt flags (just in case)
256     (EXTI->PR) = EXTI_PR_PR5;
257
258
259     // If the interruptions is triggered by the clk, we can
            assure we only have to read the bit
260     // So read it
261     int32_t bit = (COMMS_PORT->IDR) >> RX_PIN;
262     *current_pos = bit;
263     current_pos++;
264
265     // Enable the interrupts again
266     // Line 5
267     (EXTI->IMR) = (EXTI->IMR) | EXTI_IMR_MR5;
268
269     // Clear all the interrupt flags (just in case)
270     (EXTI->PR) = EXTI_PR_PR5;
271
272     // End of the ISR code
273     CH_IRQ_EPILOGUE();
274
275   }
276
277  // Interrupt associated to the RX line
278  CH_IRQ_HANDLER(EXTI15_10_IRQHandler){
279
280   CH_IRQ_PROLOGUE();
281
282    // Start of the ISR code
283
284    // Disable the interrupt
285    (EXTI->IMR) = (EXTI->IMR & ~EXTI_IMR_MR13);
286
287     // Clear all the interrupt flags (just in case)
288     (EXTI->PR) = EXTI_PR_PR13;
289
290     // Differentiate if it is a rising or a falling edge
291
292     if ( ((CLK_PORT->IDR >> CLK_RX) & 1 )== 1) {
293
294             if ((((COMMS_PORT->IDR) >> RX_PIN) & 1) == 0) {
295
296                     // If it is a falling edge and CLK_RX is
                            high --> Start sign
297
298                     // Activate interrupts of RX_CLK
299
300                     //EXTI5: Assign the port B (clk port) to
```

```
                        the external interrupt line EXTI5.
                        With the & we are erasing the 4 bits
                        that were
301                      // previously indicating which port
                             was assigned, and with the | we
                             are
302                      // indicating the new assigned port
303                      (SYSCFG->EXTICR[1]) = (SYSCFG->
                             EXTICR[1] & (~
                             SYSCFG_EXTICR2_EXTI5)) |
                             SYSCFG_EXTICR2_EXTI5_PB;
304                      // Enable interrupt line
305                      (EXTI->IMR) = (EXTI->IMR) |
                             EXTI_IMR_MR5;
306                      // Enable interrupt of clk at rising
                              edges
307                      (EXTI->RTSR) = (EXTI->RTSR) |
                             EXTI_RTSR_TR5;
308                      // Erase any pending interrupt flag
309                      (EXTI->PR) = EXTI_PR_PR5;
310                      nvicEnableVector(EXTI9_5_IRQn ,
                             CORTEX_PRIORITY_MASK(
                             STM32_EXT_EXTI5_9_IRQ_PRIORITY )
                             );
311
312                      // Clear the previous message (
                             setting the pointer to the
                             initial position of memory)
313                      current_pos = received_message;
314
315                  } else {
316
317                      // If it is a rising edge and CLK_RX
                              is high --> End sign
318                      // Process  the message and show it
319                      processMessage();
320
321                  }
322      }
323
324      // Enable the interrupts again
325      // Line 13
326      (EXTI->IMR) = (EXTI->IMR) | EXTI_IMR_MR13;
327
328      // Clear all the interrupt flags (just in case)
329      (EXTI->PR) = EXTI_PR_PR13;
330
331    // End of the ISR code
332    CH_IRQ_EPILOGUE();
```

```
333
334   }
```

We also tried another full-duplex communication system using the bit-banging method. As you will be able to see, there is no much difference between the two realizations.

```
1    /*
2   * comms_b.h
3   *
4   *   Full duplex serial synchronous communication using bit
        banging
5   *
6   */
7
8   #ifndef COMMS_B_H_
9   #define COMMS_B_H_
10
11  #define COMMS_PORT GPIOC    // Communications port
12  #define CLK_PORT GPIOB      // Clocks port
13
14  #define TX_PIN    15
15  #define RX_PIN    13
16
17  #define CLK_TX     7
18  #define CLK_RX     5
19
20  #define MAX_LENGTH    256 // Maximum length of a message (256
        bits)
21
22  #define T    1000   // Clk period (in us)
23
24  void initComms(void); // Initialization of the board to
        communicate between boards
25
26  void sendMessage(char *string); // Send a string to the
        receiver
27
28  #endif /* COMMS_B_H_ */
```

```
1    /*
2   * comms_b.c
3   *
4   *  Created on: 16/05/2016
5   *      Author: usuario
6   */
7
8
```

```
9
10  #include "Base.h"      // Basic definitions
11  #include "comms.h"
12  #include "lcd.h"
13
14  static void GPIO_ModePushPull(GPIO_TypeDef *port, int32_t
        linia, int32_t speed)
15   {
16          // 1st step: modify MODER register to set OUTPUT
                mode
17          (port->MODER) = ((port->MODER) & (~BIT(2*linia + 1))
                ) | (BIT(2*linia));
18
19          // 2nd step: modify OTYPER register to select PUSH-
                PULL mode
20          (port->OTYPER) = (port->OTYPER)&(~BIT(linia));
21
22          // 3rd step: modify OSPEEDR register to set the
                working speed
23          switch(speed){
24                  case (2):
25                   (port->OSPEEDR) = (port->OSPEEDR) & (~BIT
                         (2*linia)) & (~BIT(2*linia + 1));
26                   break;
27
28                  case (25):
29                   (port->OSPEEDR) = ((port->OSPEEDR) & (~BIT
                         (2*linia + 1))) | (BIT(2*linia));
30                   break;
31
32                  case (50):
33                   (port->OSPEEDR) = ((port->OSPEEDR) | (BIT
                         (2*linia + 1))) & (~BIT(2*linia));
34                   break;
35
36                  default:
37                   (port->OSPEEDR) = (port->OSPEEDR) | (BIT(2*
                         linia)) | (BIT(2*linia + 1));
38          }
39
40
41          // 4th step: modify the PUPDR register to set no
                pull-up/pull-down resistors
42          (port->PUPDR) = (port->PUPDR) & (~BIT(2*linia)) & (~
                BIT(2*linia + 1));
43
44
45          // 5th step: set the output to '0' (modify ODR
                register)
```

```
46          (port->BSRR.H.clear) = BIT(linia);
47   }
48
49
50  static void GPIO_ModeInput(GPIO_TypeDef *port, int32_t linia
        ){
51          // 1st step: modify MODER register to set INPUT mode
52          (port->MODER) = ((port->MODER) & (~BIT(2*linia + 1))
                ) & (~BIT(2*linia));
53
54          // 2nd step: modify the PUPDR register to no pull up
                 nor pull down
55          (port->PUPDR) = ((port->PUPDR) & (~BIT(2*linia))) &
                (~BIT(2*linia + 1));
56
57   }
58
59
60  void initComms(){
61
62          // TX and CLK-TX have to be configured as Push-Pull
                output
63          GPIO_ModePushPull(COMMS_PORT, TX_PIN, 0);
64          GPIO_ModePushPull(CLK_PORT, CLK_TX, 0);
65
66          // RX and CLK_RX as inputs with no pull-up nor pull-
                down
67          GPIO_ModeInput(COMMS_PORT, RX_PIN);
68          GPIO_ModeInput(CLK_PORT, CLK_RX);
69
70          // Configure interruption of RX_CLK and RX pin
71          // Activate SYSCFG clock
72          (RCC->APB2ENR) = (RCC->APB2ENR |
                RCC_APB2ENR_SYSCFGEN);
73
74          //EXTI13: Assign the port C (RX port) to the
                external interrupt line EXTI13. With the & we
                are erasing the 4 bits that were
75          // previously indicating which port was assigned,
                and with the | we are
76          // indicating the new assigned port
77          (SYSCFG->EXTICR[3]) = (SYSCFG->EXTICR[3] & (~
                SYSCFG_EXTICR4_EXTI13)) |
                SYSCFG_EXTICR4_EXTI13_PC;
78
79          // Enable interruption lines
80          (EXTI->IMR) = (EXTI->IMR) | EXTI_IMR_MR13;
81
82          // Enable interrupts of RX_Pin at both edges (for
```

```
            synchronization)
83          (EXTI->RTSR) = (EXTI->RTSR) | EXTI_RTSR_TR13;
84          (EXTI->FTSR) = (EXTI->FTSR) | EXTI_FTSR_TR13;
85
86          // Erase any pending interrupt flag
87          (EXTI->PR) = EXTI_PR_PR13;
88
89          // Enable the vector that manages the interruptions
                of line 13 (RX_PIN)
90          nvicEnableVector(EXTI15_10_IRQn ,
                CORTEX_PRIORITY_MASK(
                STM32_EXT_EXTI10_15_IRQ_PRIORITY));
91
92          // Set initial values of TX and CLK-TX (both high)
93          CLK_PORT->ODR = CLK_PORT->ODR | BIT(CLK_TX);
94          COMMS_PORT->ODR = COMMS_PORT->ODR | BIT(TX_PIN);
95
96  }
97
98
99  // Sends a string to the receiver using bit banging
100 void sendChar(char car){
101
102         // Send every bit of the char
103         int32_t i = 0;
104         for (int i = 0; i++; i<8){
105
106                 // Generate falling edge of the clk
107                 CLK_PORT->ODR = CLK_PORT->ODR & (~BIT(CLK_TX
                        ));
108
109                 // Write the bit in the TX_pin
110                 if (car>>i & 1 == 0){
111                         COMMS_PORT->ODR = COMMS_PORT->ODR &
                                (~ BIT(TX_PIN));
112                 } else {
113                         COMMS_PORT->ODR = COMMS_PORT->ODR |
                                BIT(TX_PIN);
114                 }
115
116                 // Wait
117                 DELAY_US(T/2);
118
119                 // Generate rising edge of the clk
120                 CLK_PORT->ODR = CLK_PORT->ODR | BIT(CLK_TX);
121
122                 // Wait
123                 DELAY_US(T/2);
124         }
```

```
125
126 }
127
128
129 // Send a string to the receiver using bit banging
130 void sendMessage(char *string){
131
132
133         // Indicate the start of a message by creating a
                falling edge before the
134         // clk generation
135         COMMS_PORT->ODR = COMMS_PORT->ODR & (~BIT(TX_PIN));
136
137         while (*string) {
138                 //Send the string char by char using the
                        function
139                 // sendChar
140                 sendChar(*string++);
141         }
142
143         // Send the EOL sign (0xFF)
144         sendChar(0xFF);
145
146         // Indicate the end of the message with a rising
                edge after the clk deactivation
147         COMMS_PORT->ODR = COMMS_PORT->ODR & (~BIT(TX_PIN));
148         DELAY_US(100);
149         COMMS_PORT->ODR = COMMS_PORT->ODR | BIT(TX_PIN);
150
151 }
152
153 /********************* ISR FUNCTIONS
        ****************************
154 Those functions should never be called directly or
        indirectly from
155 the main program. They are automatically called by the NVIC
156 interrupt subsystem.
157 ***************************************************************
        */
158
159 // Global volatile variable to store the received message
160 int8_t received_message[MAX_LENGTH];
161 volatile int8_t *current_pos = received_message;
162
163
164 // Process the received message and show it on the LCD
        screen
165 static void processMessage(void){
166
```

```
167             char message[MAX_LENGTH/8]; // Received message as a
                    string
168             char aux; // Helper variable that stores the char
                    that is currently being
169                               // built form the bits stored in
                                      receibed_message
170
171             // Put the pointer at the initial position of the
                    array
172             current_pos = received_message;
173
174
175             int32_t counter_char = 0; // Counter to know how
                    many bits of the current char we have already
                    created
176             int32_t counter_string = 0; // Counter to know how
                    many chars of the string we have already
                    obtained
177
178             // Our EOL sign is 11111111 = 0xFF
179             while (aux != 0xFF){
180
181                     aux = aux | (*current_pos<<counter_char);
182             counter_char = (counter_char + 1)%8;
183
184             // Check if we have a complete char
185             if ( (counter_char == 0) & (counter_string > 0)){
186                     // We have a complete char --> We have to
                            put it in the string
187                     message[counter_string] = aux;
188                     aux = 0;
189                     counter_string++;
190             }
191
192                     current_pos++;
193              }
194
195             // We have to remove the EOL sign (as we don't want
                    it to be shown in the LCD)
196             message[counter_string] = 0;
197
198             LCD_SendString(message);
199
200     }
201
202     // EXTI5 ISR
203     // Associated to a receiving message, allows us to read the
            bit after a
204     // rising edge of the rx_clk
```

```
205  CH_IRQ_HANDLER(EXTI9_5_IRQHandler){
206
207     CH_IRQ_PROLOGUE();
208
209     // Start of the ISR code
210
211     // First of all, check if the interrupt comes from
212     // the rx_clk pin
213
214     if (( (EXTI->PR >> CLK_RX) & 1) == 0) {
215             // Interruption does not come from the RX_CLK line
                    --> Ignore it
216             // Clear the interrupt flags and return
217             int32_t pr_state = (EXTI->PR);
218             (EXTI->PR) = pr_state;
219             return;
220     }
221
222     // If we are here, the interrupt comes from the rx_clk
223     // We have to read the bit value
224
225     // Disable the interrupt
226     (EXTI->IMR) = (EXTI->IMR & ~EXTI_IMR_MR5);
227
228     // Clear all the interrupt flags (just in case)
229     (EXTI->PR) = EXTI_PR_PR5;
230
231     // Read the bit value
232     int32_t bit = ((COMMS_PORT->IDR) >> RX_PIN) & 1;
233     *current_pos = bit;
234     current_pos++;
235
236     // Enable the interrupts again
237     // Line 5
238     (EXTI->IMR) = (EXTI->IMR) | EXTI_IMR_MR5;
239
240     // Clear all the interrupt flags (just in case)
241     (EXTI->PR) = EXTI_PR_PR5;
242
243     // End of the ISR code
244     CH_IRQ_EPILOGUE();
245
246  }
247
248  // Interrupt associated to the RX line (that allows us to
        detect the start and the
249  // end of a reception)
250  CH_IRQ_HANDLER(EXTI15_10_IRQHandler){
251
```

```
252     CH_IRQ_PROLOGUE();
253
254     // Start of the ISR code
255
256     // First of all, check if the interrupt comes from
257     // the RX pin
258
259     if (( (EXTI->PR >> RX_PIN) & 1) == 0) {
260             // Interruption does not come from the RX line -->
                    Ignore it
261             // Clear the interrupt flags and return
262             int32_t pr_state = (EXTI->PR);
263             (EXTI->PR) = pr_state;
264             return;
265     }
266
267     // If we are here, the interrupt is caused by the RX pin
268
269     // Disable the interrupt
270     (EXTI->IMR) = (EXTI->IMR & ~EXTI_IMR_MR13);
271
272      // Clear all the interrupt flags (just in case)
273      (EXTI->PR) = EXTI_PR_PR13;
274
275      // Check if it is a start sign or a end sign. In both
           cases
276      // the rx_clk has to be high
277      if ( ((CLK_PORT->IDR >> CLK_RX) & 1 )== 1) {
278
279              if ((((COMMS_PORT->IDR) >> RX_PIN) & 1) == 0) {
280
281                      // If it is a falling edge and CLK_RX is
                            high --> Start sign
282                      // Activate interrupts of RX_CLK (in
                            order to read the bit at the right
                            time)
283
284                      //EXTI5: Assign the port B (clk port) to
                            the external interrupt line EXTI5.
285                      (SYSCFG->EXTICR[1]) = (SYSCFG->EXTICR[1]
                            & (~SYSCFG_EXTICR2_EXTI5)) |
                            SYSCFG_EXTICR2_EXTI5_PB;
286                      // Enable interrupt line
287                      (EXTI->IMR) = (EXTI->IMR) | EXTI_IMR_MR5;
288                      // Enable interrupt of clk at rising
                            edges
289                      (EXTI->RTSR) = (EXTI->RTSR) |
                            EXTI_RTSR_TR5;
290                      // Erase any pending interrupt flag
```

```
291                         (EXTI->PR) = EXTI_PR_PR5;
292                         nvicEnableVector(EXTI9_5_IRQn ,
                                CORTEX_PRIORITY_MASK(
                                STM32_EXT_EXTI5_9_IRQ_PRIORITY ));
293
294                             // Clear the previous message (
                                    setting the pointer to the
                                    initial position of memory)
295                             current_pos = received_message;
296
297                     } else {
298
299                             // If it is a rising edge and CLK_RX
                                     is high --> End sign
300                             // Process  the message and show it
301                             processMessage();
302                     }
303       }
304
305       // Enable the interrupts again
306       // Line 13
307       (EXTI->IMR) = (EXTI->IMR) | EXTI_IMR_MR13;
308
309       // Clear all the interrupt flags (just in case)
310       (EXTI->PR) = EXTI_PR_PR13;
311
312     // End of the ISR code
313     CH_IRQ_EPILOGUE();
314
315     }
```

As this time we had to work with another lab group, we clarified every code step in the program, so we don't think that it will be necessary to comment what the code does as it is self-explained. Finally, we want to stand out that the communication, actually, did not work perfectly. We were able to send the messages and decode them only a few percent of the times. We strongly think that a problem in the decodification of the signal has been done (**processMessage** method) and we would have loved to fix it if we have had more time.

Conclusions

We want to stand out that have learned a lot about programming a MCU and also about the behaviour of it. Its also remarkable that we have learned while having fun, for this reason we consider that this laboratory is a very good docent method for the students to learn interactively. We think that maybe the evaluation should be a little different, taking more into account what the student does in the lab instead of weighing the lab exam so much (unless if it is made using the PC). It's possible for a student to have a bad day or maybe be messed up in paper instead of using eclipse. Also, the fact of not having the manual does not help to evaluate the student abilities, because being a good programmer and understanding well the subject does not mean to memorize a practice or the manual contents.

To sum up, we consider this as a very interesting lab and would like to point out the efficiency in terms of understanding the subject of it. For this reason, we consider that a taking-into-consideration practice for this course could be increasing the lab number of hours.