# LIFT CONTROL USING AN ARM-BASED MICROCONTROLLER

ALEJANDRO LÓPEZ RODRÍGUEZ
ANA MARÍA CASANOVA LÓPEZ

*Universidad Politécnica de Valencia*
*ETSID*

JUNE 7, 2021

Escuela Técnica Superior de Ingeniería del Diseño

## Contents

## Summary

This project is devoted to the development of an embedded application which controls an industrial lift that transports heavy loads between two floors.

From the get-go, the main goal of the project was to develop an optimized and efficient application, with proper configuration of the clocks, the interrupts, the IO and a few extra peripherals.

As per the lab sessions, the Keil uVision IDE was used to program the code and manage the files. In addition, STM32CubeMX was also used to configure the internal clocks.

The first step to tackling the problem was to carefully read the requirements and to create a diagram containing the desired workflow.

Once clarified, the peripherals were then configured one-by-one to accommodate for the different requirements. A modular approach was decided upon so as to simplify and organize the implementation.

## Flowchart

There are two main variables which control the functioning of the infinite loop, ***buttonPressFlag*** and ***timer5sEndFlag***, both of them are initialized to false (these are Boolean variables). The first one is set to true when the interrupt attached to the button, EXTI0, is triggered. The second one, as its name states, is set when the 5 second timer (TIM3), overflows. Both are reset in the infinite loop.

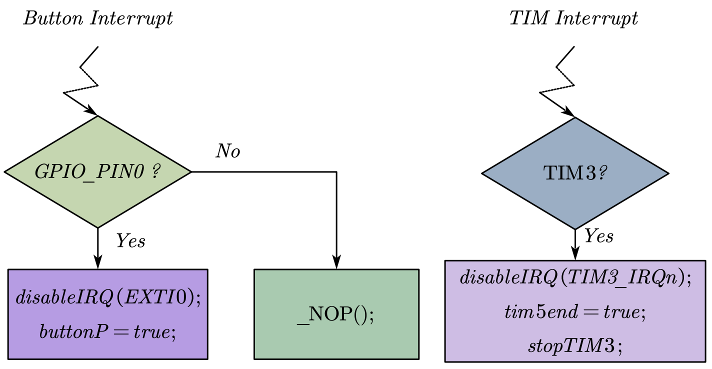

Figure 1: Interrupts workflow

After the main initialization, in the main loop, the system checks whether the ***ButtonPressFlag*** is set. If it is, first the flag is reset, and, depending on the current floor, the next action is to go up or down. Inside the functions *liftUp()* and *liftDown()*, first the doors are closed, then the direction of the movement is decided, the LEDs are set to blink in the correct pattern, and finally, the timer 3 is started. The direction of the lift is sent via the UART.

Then, the ***timer5EndFlag*** is polled. If its value is *true*, the function *liftStop()* is called, which re-enables both the EXTI0 IRQ and the TIM3 IRQ, turns off orange and red LEDs, sets the current floor depending on the lift direction, and opens the doors. In addition, it also sends the current floor through the UART.

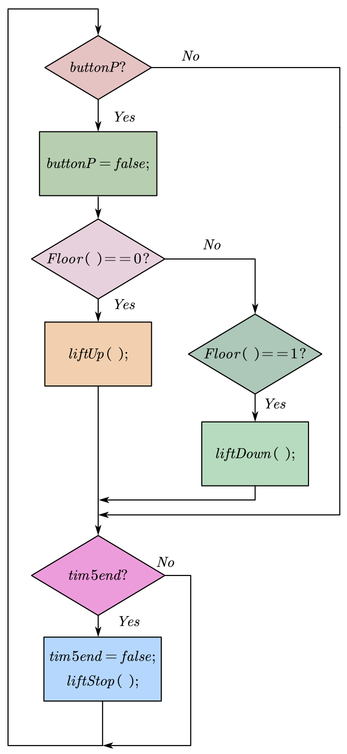

Figure 2: Flow chart describing the operation of the main loop

## 1.1 Files and functions

Instead of defining all the required functions in just one file, we created the *main.c* file following the simple structure explained in previous sections, and a main function file, *lift.c*, which calls different functions from other sub-files to perform the wanted actions.

This is a brief explanation of each file:

- ***button.c*** **:** Initializes the button hardware as well as its interrupt.
- ***doors.c*** **:** Initializes the servomotors hardware and the functions to control the angle of their shaft. As two main positions are desired (open/close), each of them has its independent function.
- ***led.c*** **:** Initializes the LEDs hardware and PWM, includes a function to set the PWM duty cycle, turns off the red and orange LEDs, and sets all LEDs to its initial state. The blue and green ones are controlled by an independent function which turns them on/off depending on the state of the lift (on a floor or moving).
- ***lift.c*** **:** This file is the main nexus. It calls the different functions of the sub-files to control the main actions of the lift, such as initialization, move up/down, stop, retrieve the current floor, etc.
- ***motor.c*** **:** Initializes the stepper motor GPIO.
- ***timer.c*** **:** Configures the timers used in this application, TIM1, TIM3 and TIM4.
- ***uart.c*** **:** Manages the initialization of the UART communication.

All of these files are strictly necessary for the proper operation of our application. As it can be seen on the project, inside "Core/Src" there are more files: *stm32f4xx_ hal_ msp.c*, which is necessary for the MSP initialization and de-initialization; *stm32f4xx_ it.c*, which is indispensable for interrupts; *system_ stm32f4xx.c*, that initializes the system; *stm32f4xx_ hal_ uart.c*, essential for UART module operation (this file would be, initially, inside the HAL drivers folder but since from the point of view of the CubeMX this peripheral is unused, it does not generate this file. To solve this problem, we got this file from another project in which we had configured the UART from CubeMX and included it in our source code folder.)

Inside "Core/Inc", apart from the header files of the named files, we also find *stm32f4xx_ hal_ conf.h*, which defines which peripherals are active.

If "Generate code" is clicked in CubeMX, in the file *stm32f4xx_ hal_ conf.h*, the UART and Timers peripherals will **NOT** be enabled. To solve this, it is necessary to uncomment lines 65 and 66 of said file.

The set of functions shown in the sub-files were organized in the following way. (We also developed more functions, but these are the main ones used in *lift.c*).

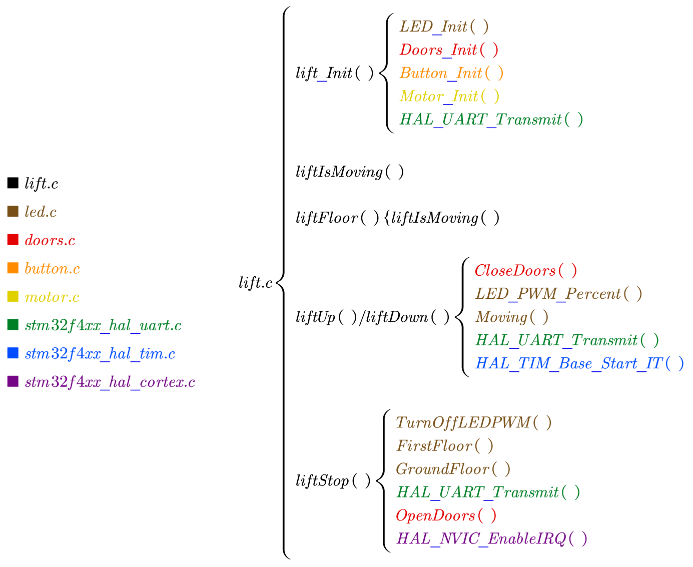

Figure 3: File dependency tree

## Clocks and peripherals configuration

As in this project, more peripherals than the strictly necessary ones were used, we will firstly introduce the essential ones and then, the rest.

### 2.1 Minimum configuration

As the problem states, the input consists of a button, and the outputs are 4 LEDs, which are already on the STM32F407 Discovery board.

The button, as stated before, must produce an interrupt, due to this, the EXTI0 interrupt was configured to trigger following the press of the blue button.

The green LED and the blue one were connected to simple push-pull outputs. No external resistors were required for this application. As the frequency did not matter in this case, we configured it as low.

The other two LEDs, orange and red, were configured to be controlled by Pulse Width Modulation. Since both share the same period, 1 second, they were configured with the same timer, in this case, TIM4. On the actual pin configuration, these pins had to be configured as alternate function push-pull. The internal resistor and speed parameters were configured mimicking the previously defined LED configuration. These LEDs in particular needed an extra configuration step: on the enumerated type variable *GPIO_ InitStruct, Alternate*, the timer that controls them had to be configured as well for them to work as expected.

In order to control the duty cycle of the LEDs, a special function was developed, which takes the LED pin as an input, as well as the desired duty cycle, and both computes and writes the required pulse value to the CCRx register.

```c
void LED_PWM_Percent(uint16_t LED, uint8_t value){
    uint32_t Pulse = 0;
    /* Calculate the Pulse value given a duty cycle percentage */
    if(value > 100) Pulse = 9999;
    else if(value == 0) Pulse = 0;
    else Pulse = (uint32_t)(value * 100 - 1);
    /* Update the PWM duty cycle by modifying the CCRx value */
    if(LED == LD3_Pin) /* Orange LED */
        TIM4->CCR2 = Pulse;
    else if(LED == LD5_Pin) /* Red LED */
        TIM4->CCR3 = Pulse;
}
```

Regarding the internal clock, the configuration was practically the same as the one that we used for the different lab sessions, namely, setting the internal clock to 100 MHz, and the APB1 and APB2 clocks to 1/2 of that.

As we needed a 1s period PWM, these values for the prescaler and counter were used:

$$\text{TIM4 (T} = 1\text{ s)} \rightarrow \begin{cases} \text{TIM4 CLK} = \dfrac{50\text{ MHz}}{\text{Presc}} = \dfrac{50\text{ MHz}}{5000} = 10\text{ kHz} \\[2ex] \text{Counter period} = ((\text{TIM4 CLK}) \cdot \text{T} - 1) = 999 \end{cases}$$

In order to count the 5 seconds, which correspond to the time that it takes for the lift to move, timer 3 was chosen.

$$\text{TIM3 (T} = 5\text{ s)} \rightarrow \begin{cases} \text{TIM3 CLK} = \dfrac{50\text{ MHz}}{\text{Presc}} = \dfrac{50\text{ MHz}}{50000} = 1\text{ kHz} \\[2ex] \text{Counter period} = ((\text{TIM3 CLK}) \cdot \text{T} - 1) = 4999 \end{cases}$$

## 2.2 Extra peripherals

Two servomotors and one stepper motor were also used to increase the complexity of the project. For communications, the aforementioned UART peripheral was used in order to send information about the current action/set of actions.

To operate the servomotors, a PWM signal was used, the minimum pulse width of which was 0.75 ms; the maximum 2.25 ms. Its frequency, as per the manufacturer datasheet [1] was set to 50 Hz. The timer chosen to generate this signal was TIM1. Since we had two servomotors, we used two of the timer's channels (Connected to pins PE9 and PE11.)

In order to control the angle, a special function was developed which takes the servomotor pin as an input, as well as the desired angle, and both computes and writes the required PWM pulse value to the corresponding CCRx register.

```c
void Servo_PWM_Angle(uint16_t Servo, uint8_t angle){
    uint32_t Pulse = 0;
    /* Calculate the Pulse value given a duty cycle percentage */
    if(angle > 180) Pulse = 224;
    else if(angle == 0) Pulse = 74;
    else Pulse = (uint32_t)(angle * (5.0/6.0) + 74);
    /* Update the PWM duty cycle by modifying the CCRx value */
    if(Servo == Servo1_Pin) /* Orange LED */
        TIM1->CCR1 = Pulse;
    else if(Servo == Servo2_Pin) /* Red LED */
        TIM1->CCR2 = Pulse;
}
```

The timer 1 configuration was configured as follows:

$$\text{TIM4 (T = 1 s)} \rightarrow \begin{cases} \text{TIM4 CLK} = \dfrac{50\text{ MHz}}{\text{Presc}} = \dfrac{50\text{ MHz}}{5000} = 10\text{ kHz} \\ \text{Counter period} = ((\text{TIM4 CLK}) \cdot \text{T} - \ 1) = 999 \end{cases}$$

For the stepper motor to work properly, it needed a Look-Up Table and 1 ms wait between step changes. As the SysTick timer generates an interrupt every 1 ms, we decided to take advantage of this predefined interruption, reducing the necessary number of extra timers. Depending on the direction of the motor, the index of the LUT gets increased or decreased in each call. This can be observed in the following code:

```c
void SysTick_Handler(void){
  HAL_IncTick();

  if(liftIsMoving() && DirUP)
    (step_index == 7)? step_index = 0 : step_index++;
  else if(liftIsMoving() && DirDOWN)
    (step_index == 0)? step_index = 7 : step_index--;

  GPIOB->ODR &= ~(0xF << 12);
  GPIOB->ODR |= step_positions[step_index] << 12;
}
```

The UART was configured in order to have an asynchronous communication, with a baud rate of 9600, which is a common value for this type of communications.

In order to make the microcontroller transmit information, we used the HAL function designed for this purpose. This is an example of its usage:

```c
unsigned char mssg[]="Initialization complete\n";
HAL_UART_Transmit(&huart2,mssg,sizeof(mssg)/sizeof(mssg[0]),0xffff);
```

Summing up, all the pins configurations can be found below:

- Blue Pushbutton
  - **PA0:** GPIO_EXTI0, no pull-up no pull-down.
- UART Communication
  - **PA2, PA3:** USART2_TX, USART2_RX.
- Stepper Motor
  - **PB12, PB13, PB14, PB15:** GPIO_Output, Push-Pull, no internal resistor, output speed low.
- Servomotor
  - **PE9:** TIM1_CH1.
  - **PE11:** TIM1_CH2.
- Floor Number LEDs
  - **PD12 (Green Led), PD15 (Blue Led)**: GPIO_Output, Push-Pull, no internal resistor, output speed low.
- State LEDs
  - **PD13 (Orange Led):** TIM4_CH2.
  - **PD14 (Red Led):** TIM4_CH3.

## Conclusion

This project is the culmination of months of work on this subject. We have applied the knowledge obtained in class to the best of out abilities, with gratifying results. In the end, everything worked as expected, even though we spent more time than we would like to admit debugging the code. This was due to the ordeal of configuring the peripherals without the aid of CubeMX, the "vanishing" of documents like *stm32f4xx_hal_uart.c* and the need to uncomment a couple of lines from other files, which took a while to figure out.

That being said, we are really happy with how the project turned out in the end. In **this link** you can find the project's GitHub repository, and, in **this one** the video of the presentation and operation of the project. We hope that you like it!

## References

[1] *Parallax Standard Servo*, 900-00005, Rev. 2.2, Parallax INC., Oct. 2011. [Online]. Available: https://www.parallax.com/package/parallax-standard-servo-downloads/.