# EE2028: Microcontroller Programming and Interfacing

Assignment 2 (Tuesday Lab)


| Project Group 20 | |
|---|---|
| Bui Quang Huy | A0220177M |
| Shin Donghun | A0220616R |


**Table of Contents**



## 1. Introduction and objectives

In this assignment, we were asked to design a prototype of the intelligence sub-system installed in Pixie. It is assumed that the sub-system is running on B-L475E-IOT01A board, while some of the sensors are mounted on Pixie. **Pixie** will use its sensors to monitor the environment and hardware's condition, send updates periodically to Cyrix at the lab, and call for help when it is in serious trouble.

**Pixie** has **TWO** modes of operation: **EXPLORATION** and **BATTLE** modes, and will be transmitting data/messages to Cyrix's Lab. The EXPLORATION mode is the mode in which the two are casually exploring the city, hoping to discover more lost memories. BATTLE mode is the mode in which the drone is detected by the Zurks and it needs to escape with movement or flash the Fluxer. The sensors on board are to measure the various physical quantities. The LED onboard can be used as a mode/state indicator for debugging/monitoring purposes. The user pushbutton initiates mode switching and various actions by Pixie.

In this report, it covers a detailed overview and explanation of codes for users to easily understand the logic of the program. It also includes an OVERLOAD Mode, which is an enhancement we have added, on top of the basic requirements.

## 2. Basic Requirements

### 2.1 Flowchart of Basic Requirements & Single/Double Press

In the following figures, we have designed the flowcharts to have a clear approach to developing the program, before coding. Fig. 1 is Flowchart to achieve basic requirements, and it simplifies the complicated requirements and helps with visualisations for implementation in code.

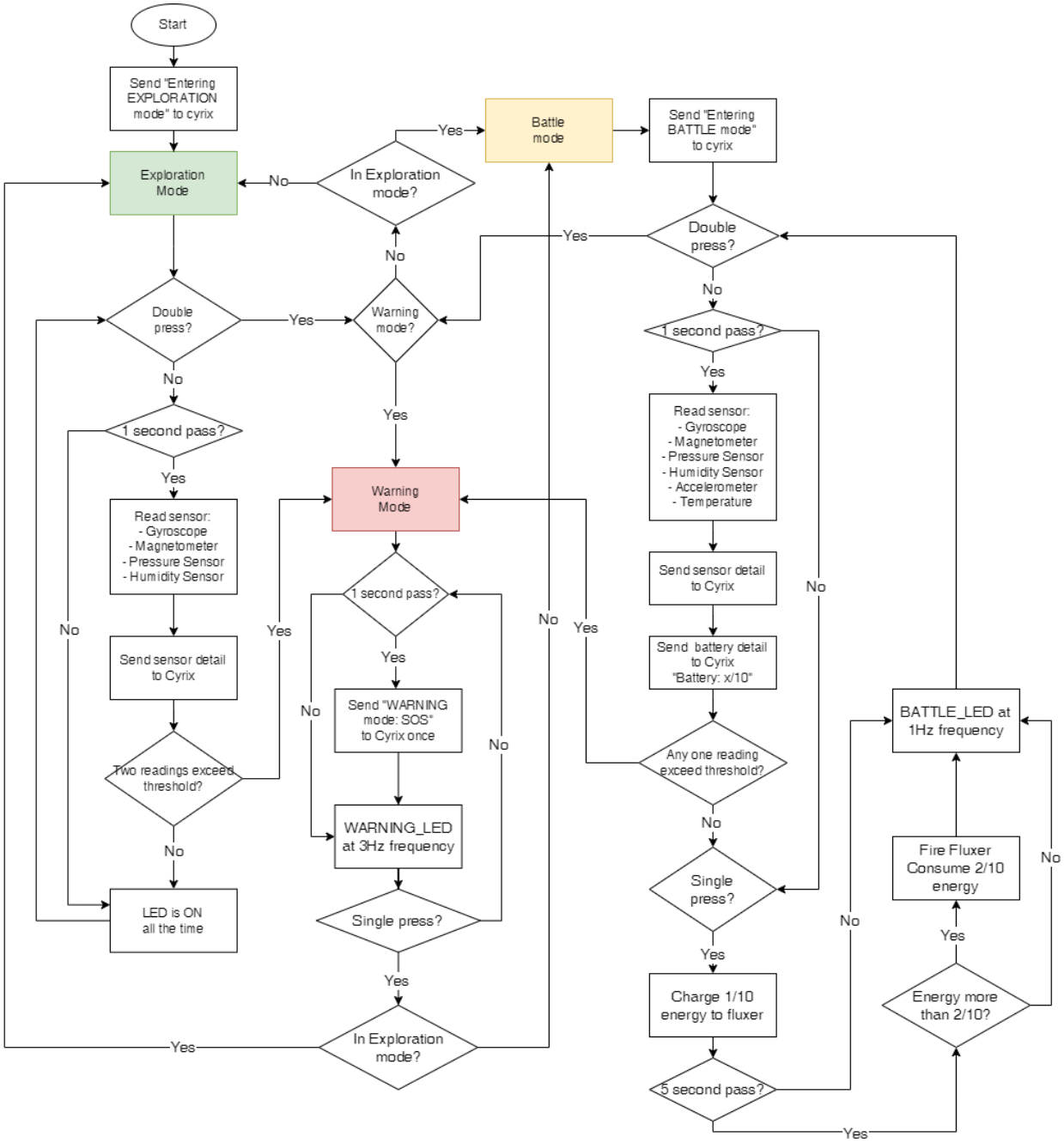



*Fig. 1 Flowchart to achieve basic requirements*

In Fig. 2, It shows a general approach to achieve Single/Double press detections using uwTick. With reference to this flowchart, we implemented the concept in a checkPress Function. For our codes, single/double press detection is done only within the checkPress function.

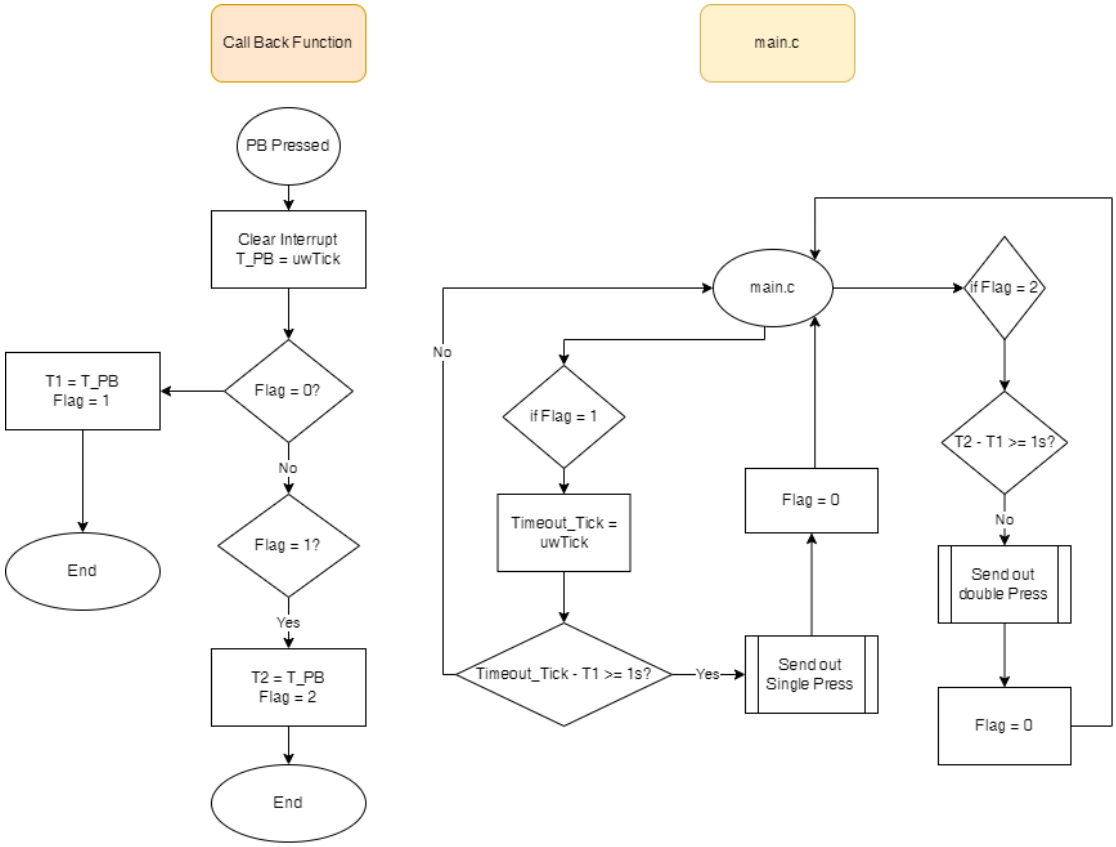
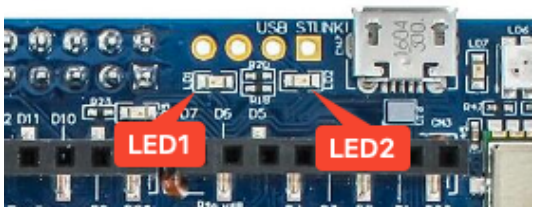



*Fig 2. General Flowchart to identify Single/Double Press*

## 2.2 Detailed implementation

For our implementation, we use both LEDs on the board instead of just one. From now on will be referred to as LED1 (left) and LED2 (right) respectively. In Warning mode/status, LED1 will keep its status according to the previous state before Pixie enters Warning. LED2 will flash when Pixie is in Warning mode.




Summary of sensor reading units:

| Sensor | Raw Unit | Units Used | Threshold Value |
|---|---|---|---|
| Accelerometer (A) | mg | g | 1.9 g |
| Gyrometer (G) | mdps | dps | 10 dps |
| Magnetometer (M) | mgauss/LSB | gauss | -0.4 gauss |

| | | | |
|---|---|---|---|
| Pressure Sensor (P) | hPa | hPa | 1200 hPa |
| Humidity Sensor (H) | % | % | 95 % |
| Temperature Sensor (T) | °C | °C | 39 °C |
| Time-of-Flight Sensor (ToF) | mm | cm | - |

The table below summarises the logic/pseudocodes for the basic requirements of the project

**1. Initialisation of constants in main.h**

**Constants Definitions for easier modifications**

1. Multiple Delays. Eg. #define FLUXER_DELAY 5000
2. Suitable Threshold values (upper limit) for the sensors. Eg. #define TEMP_THRESHOLD 39
3. Others Constants. Eg. #define ONE_SECOND 1000, #define MAX_BATTERY 10

**2. Configurations of the peripherals (in main Function)**

**Configuration of Peripherals**

1. Reset all peripherals & Initialise Systick by calling HAL_Init()
2. Initialise UART by calling UART1_Init() - which contains the logic for UART configurations
3. Initiatlise BSP by calling functions. Eg. BSP_GYRO_Init()
4. Configure GPIO pins by calling MX_GPIO_Init(), which contains configuration logics for LED pins and Push Button Pin. *(refer to part 4)*

**Constantly check of its mode**

5. In an infinite while loop,
    a. Run checkPress() function, which contains logic for Single/Double Press detection and consequences *(refer to parts 5 & 6)*
    b. Conditions to check the mode, and for each mode, a function is called to enter the mode *(refer to 7 & 8)* Eg. if (mode == 'E') {enterExplorationMode();}

**Boot Up Message**

6. Sends EXPLORATION_MSG to Cyrix's Lab when first boot up

**3. Configuration of GPIO Pins (in MX_GPIO_Init Function)**

**Configuration of appropriate pins for input and output**

1. Enable AHB2 Bus for GPIOA/B/C/D. Eg. AHB2_HAL_RCC_GPIOX_CLK_ENABLE()
2. Reset pin to 0 for LED2 and LED 1
3. Proceed to configure LED2_Pin (GPIO-B Pin-14) and LED1_Pin (GPIO-A Pin-5) as GPIO outputs.
    a. Pass in the values for Struct parameters like Pin, Mode, Pull and Speed for each LED.
    b. Complete the configuration by passing GPIOX and address of GPIO_InitStruct to HAL_GPIO_Init function for each LED
4. Similarly, configure for Push Button as BUTTON_EXTI13_Pin (GPIO-C Pin-13) as AF

**4. Double Press and its outcome at different modes (in checkPress Function)**

1. Check conditions to check if double press is detected. Conditions checked is "if uwTick - pressDuration is less than equal to 1s and pressCount tracked is more than equal to 2"

**If conditions satisfied (indeed double press detected),**

2. Reset the pressCount and pressDuration
3. Send out "Double Pressed" message to Cyrix's Lab
4. If current mode is E, switch mode to B. Then, Reset all the ticks, send out Battle Message to Cyrix's Lab and enterBattleMode
5. If current mode is B, switch mode to E. Then, Reset all the ticks, send out Exploratory Message to Cyrix's Lab and enterExploratoryMode

Note: to ensure that the Pixie enters the different modes immediately when the conditions are satisfied, functions like enterExploratoryMode() are written here on top of the one in the main function.

**5. Single Press and its outcome at different modes (in checkPress Function)**

1. Check conditions to check if double press is detected. Conditions checked is "if uwTick - pressDuration is more than 1s and pressCount tracked is exactly 1"

**If conditions satisfied (indeed single press detected),**

2. Reset the pressCount and pressDuration
3. Send out "Single Pressed" message to Cyrix's Lab
4. Check if it is in the battle mode. If it is, increase the battery level by 1.
5. Check if it is in Warning mode. If it is,
    a. Clear thresholdArr[], which has values that are only significant to trigger getting into the warning mode. Hence, in the warning mode, the array has no significance.
    b. Switch off LED2
    c. Change back to the previous mode that it was in by checking the prev_mode (which is saved when Pixie enters the different modes). Reset Ticks.

**6. Inside enterExplorationMode Function**

1. When the mode is 'E,' the code runs enterExplorationMode() to carry out tasks for its mode.
2. For tracking purpose (in order to comeback to the previous mode after going into the warning mode), save the prev_mode as 'E'
3. Switch on LED1 and keep it always on

**For every 1s (using uwTick),**

4. get the necessary data (G, M, P, H)
5. Check if the measured value exceeds the pre-set Threshold values, and update the outcome in thresholdArr[]. Eg. 1→ Exceeded; 0 → Not exceeded.
6. If any 2 elements in thresholdArr have a value '1', enter Warning Mode (while taking the prev_mode as a parameter, so that it can be returned to the previous mode) - this is computed by a for loop
7. When no warning, send out the G, M, P, H data readings are sent to Cyrix's Lab.

**7. Inside enterBattleMode Function**

1. When the mode is 'B' the code runs enterBattleMode() to carry out tasks for its mode

2. For tracking purpose (in order to comeback to the previous mode after going into the warning mode), save the prev_mode as 'B'
3. LED 1 Blink at 1Hz
4. Fluxer is Fired every 5s by calling fireFluxer function (refer to …)
5. //saving B as a prev_mode so that after warning mode is cleared, it can go back to the

**For every 1s (using uwTick),**

6. Send Battery Level information to Cyrix's Lab
7. Get the necessary data (A, G, M, T, P, H)
8. If any element in thresholdArr have a value '1', enter Warning Mode (while taking the prev_mode as a parameter, so that it can be returned to the previous mode) - this is computed by a for loop
9. When no warning, send out the A, G, M, T, P, H data readings are sent to Cyrix's Lab.

**8. Inside enterWarningMode Function**

(take note that it takes in the argument "prev_mode" so that it can go back to the previous mode)

1. Ensure that LED1 blinks (Battle Mode) or stays on (Exploratory Mode) depending on the previous mode
2. LED2 blinks at 3Hz when in Warning Mode
3. Send Warning Message to Cyrix's Lab every 1s

**9. Inside fireFluxer Function -** *for Battle Mode*

1. At every 5s, when battery level is equal or more than 2, fluxer is fired and battery level decreases by 2
2. Send Fluxer Message to Cyrix's Lab

**10. Inside blinkLED1 (***Battle mode)* **& blinkLED2 Functions (***Warning mode)*

**For blinkLED1 function**, LED blink at 1Hz. This is achieved by comparing the difference of uwTick and led1Tick with BATTLE_LED_DELAY (500). Note that led1Tick is the uwTick when the LED is toggled.

**For blinkLED2 function**, LED blinks at 3Hz. This is achieved by comparing the difference of uwTick and led1Tick with BATTLE_LED_DELAY (167).

**11. HAL_GPIO_EXTI_Callback Function -** *Call Back Function for Interrupts*

Inside the function, it contains logic to eliminate possibilities of debouncing issues.

**12. Multiple getXXXXData Functions**

1. Run XYZData getXXXData() functions for A, G, M
   (XYZData is a user defined struct, where it contains x, y, z float values)
   Eg. XYZData getAccelerometerData()
2. Run float getXXXData() functions for T, P, H. Eg. float getHumidityData(){...}
3. Inside the getXXXData, There is a logic to convert the raw values of the sensors to readable values with units
   Eg. (for Accelerometer reading, raw data is in mg. Convert to g for our case.)
   data.x = (float)data_arr[0] / 1000.0f;

### 2.3 Significant problems encountered

The most significant problems met while developing this section is the management of multiple chaining logic. The process of abstracting functions outside into different libraries and tracking them down is rather cumbersome and slow. Sometimes the IDE also failed to link the function to its correct place of definition. From this experience, we learn that while the logic could be simple, organising the codebase becomes not just a good practice but a must have if we want to write anything more than a hundred lines of code. Arguably our code base is still not the cleanest it can be due a lack of abstraction into their own library, but we feel that for a small scale project like this, having all the code in one place just allow for faster debugging and developing under time constraint.

## 3. Enhancement (Addition of Overload Mode)

### 3.1 Overview & Flowchart of Overload Mode implementation

Overload mode is an enhancement of additional logic added to give Pixie the ability to aim and shoot at Zurk freely. The overload mode use the VL53L0X Time of Fly sensor (ToF) on the B-L475E-IOT01A board to measure the distance of an object on top of the board, in this case the user hand, to aim Pixie's Fluxer in real life.

To enter the Overload mode, Pixie needs to be in battle mode and the fluxer battery energy must be equal to 10 when first entered and more than 10 in subsequent entry. When Pixie enters Overload mode, LED1 will be off and LED2 will be on all the time. Zurk will randomly spawn at a certain altitude, either below or above the 25 cm mark. Users can put their hand on top of the board to start aiming. The ToF will measure the distance from the board to the hand to determine the aiming altitude. To shoot, the user just needs to move their hands away from the board like a recoil action from shooting a gun. The last non-zero altitude reading will be used as the aiming altitude. If the user hits the Zurk, a new Zurk will be spawned and the cycle repeats. If the user misses the Zurk, the Zurk will successfully attack Pixie and force Pixie to enter the Warning mode.

After entering the Warning mode, Pixie behaves as normal according to the basic requirement. Single press in this mode will return Pixie back to Battle mode. All other functionality retained.

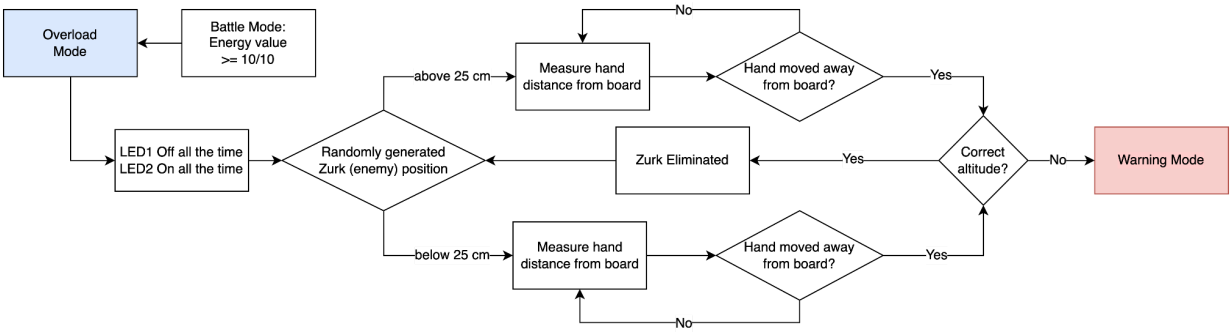



*Fig 3. Flowchart of Overload mode*

### 3.2 Detailed Implementation of Overload Mode implementation

Indicator to this would be LED1 being set as always OFF and LED 2 being set as always ON.

| **1. Addition of Overload Mode in main function** |
|---|
| In the while loop, run OverloadMode function when the mode is O |
| **2. Addition to Configuations of the peripherals (in main Function)** |
| Self configure peripheral for Time-of-Flight Sensor. Eg. VL53L0X_Start(&sensor1); |
| **3. Addition to checkPress Function when Single Press Detected** |
| If it is in battle mode and with max battery level, Change to OVERLOAD mode |
| **4. Addition of enterOverloadMode Function** |
| 1. LED1 is set to be always OFF, LED2 is set to be always ON<br>2. Check if enemyDead is 1 (at the start, it is initialised to be 1)<br>**If enemy is dead, spawn new enemy**<br>3. Generate a random position for enemy (either 0-25cm range (0) or 25 and above range (1))<br>4. Depending on which range, transmit messages, "ENEMY_BELOW_MSG" or "ENEMY_ABOVE_MSG"<br>**If enemy is alive,**<br>5. Start aiming for the enemy (using the hand on top of ToF sensor)<br>6. If the hand is correctly placed in the region specified where Zurk has spawned and then makes leaves quickly away from the board (this action is checked every 0.25s), it will kill the enemy. This is achieved by conditioning a previous_distance = "~ at the region" and the current distance = 0.<br>7. Hand Position (Aiming Altitude) is periodically updated to the Cyrix's Lab.<br>8. When the enemy is wrongly aimed, it will cause Zurk to attack Pixie first, causing Pixie to quit Overload Mode and switch to the Warning Mode |

### 3.3 Significant problems encountered

The enhancement was quite a challenge that made us step out of our comfort zone. After careful research, we realise that utilising the VL53L0X Time of Fly sensor is not an easy feed. STM32 made an executive decision to close source their VL53L0X development, meaning a lot of functions to initiate the VL53L0X depend on a deep understanding of their provided VL53L0X library. The problem further intensified when we found out that the provided VL53L0X library by STM32 has no example of how to integrate their library with B-L475E-IOT01A board and only has example to integrate with their Nucleo board family which run on STM32F4XX family instead. A deep search online also yielded no result on interacting with the VL53L0X on B-L475E-IOT01A. Even STM32's own wiki and support packages do not yield any result for a guide to interact with the VL53L0X. This leads us to believe that there is no known publicly available and easy way to interact with the ToF sensor.

We were stuck for a very long time. Given our inexperience and lack of knowledge and the time constraint reading STM32 VL53L0X detailed documents prove to be an almost infeasible challenge. However, we managed to figure out a way to interact with the VL53L0X under time constraints. We realise that an interaction with the VL53L0X on B-L475E-IOT01A is unnecessary, we actually only need to see an example of how VL53L0X can be interacted using a STM32L4XX family, the same chip family on our board. With this knowledge, we change our research direction and search for implementation of VL53L0X with STM32L4XX instead. It is still a hard challenge regardless. However, we managed to find one repository on Github of a person who managed to get his VL53L0X to work on the Nucleo_L476RG board: *JC-Toussaint/Nucleo_L476RG_VL53L0X_v1*. Arguably the only such repository on the entirety of Github.

Thanks to this we manage to extract a slim down version of the VL53L0X library and integrate it with our board. By changing some register address to suit our usage and reading the internal working of the slim down library which is much easier than the original API provided by STM32, we manage to initialise the VL53L0X on our board and get back rather accurate value from it.

This incident let us learn about the importance and significance of documentation and open source resources. To our surprise, not all boards come with an easily integrated and tested library for all the peripherals on it, which is rather strange but something we have to learn to expect. In the case of the B-L475E-IOT01A board, the VL53L0X library and sound sensor integration are not provided with the board, so to use them, users would have to dig rather deep into their documentation. If not for leveraging the knowledge and experience of open source projects, we might not have made it in time for the demo day. It is then important for us to contribute back to the open source community for the next person who will be scratching their head and wondering how to interact with the VL53L0X on the B-L475E-IOT01A board.

## 4. Conclusion

This project taught us not only the principles and practices of embedded systems engineering but also very in depth about the approach we should take in order to achieve the requirements defined in the design. For example, drawing of flowcharts really allowed us to keep track of the program logic, and helped tremendously with the debugging issues and simplification of complex model.

Furthermore with the interesting context about the project “No Way Home”, we could think more deeply about possible features to be add on in order to fit into the context. This innovative addition allowed us to design the additional feature from scratch, involving a series of challenges including coming up with a flowchart and finding a right configuration for the sensor we would want to use, and implementing it.

Through this project, we have gained confidence in developing embedded system software, and also became comfortable with referring to datasheets for different purposes.