

Autonomous Target-Tracking Controller for a Drone using a STM32 Microcontroller

**ECE 653 - Embedded and Real-Time Systems
Department of Electrical and Computer Engineering
University of Cyprus**

Course Project

**Submitted To:
Prof. Theocharis Theocharide**

**Group Members:
Adil Farooq
Dimitris Ttofi
Eleni Sotirou**

**Auditing Members:
Muhammad Shahid Noman Siddiqui
Panayiotis Aristodemou**

Contents

Contents

Contents	2
1 Table of Figures	3
2 Introduction.....	4
2.1 Motivation	4
2.2 Objectives	5
3 Project Overview	6
3.1 Bounding Box Information	6
3.2 STM32 Nucleo Board	7
3.3 Quadcopter Basics.....	9
4 Framework overview.....	11
5 Software Implementation	12
5.1 Source code organization	12
5.2 Detail Software Explanation	13
5.2.1 Sending Coordinates to STM32 board.....	13
5.2.2 Reconstructed Coordinates	14
5.2.3 Using STM32 Interrupts.....	15
5.2.4 Sending Drone Commands back to PC	17
6 Results	19
6.1 Performance	19
6.2 STM32 Memory Utilization.....	20
7 Conclusion	22
7.1 Summary and Complications.....	22
7.2 Future Work.....	22
8 References	23
9 Appendix.....	24
9.1 User defined C library	24
9.2 Initial Experiments and Complications	28

1 Table of Figures

Figure 1 Coordinates Representation	6
Figure 2 Nucleo-144 board layout.....	7
Figure 3 Quadcopter Representation.....	9
Figure 4 Roll, Pitch, Yaw and Throttle movements	10
Figure 5 Proposed flow diagram.....	11
Figure 6 State Machine Implementation.....	13
Figure 7 AirSim Environment.....	19
Figure 8 Output log Files.....	20
Figure 9 Memory utilization	20
Figure 10 LED and Board Connection.....	28
Figure 11 External RGB LED Changing Colors	29
Figure 12 Nucleo STM327Z1 UART Connection	30

2 Introduction

Unmanned aerial vehicles (UAVs), which are often called drones or multicopper, have high autonomy and flexibility. However, they have become a popular research topic, as they are important tools not only in the military domain, but in civilian environments, too. Also, they are popular for observational and exploration purposes in indoor and outdoor environments, for data collections, object manipulation or simply as high-tech toys. Additionally, there are many more potential applications. An example is that they could be deployed to explore collapsed buildings to find survivors. The advantage of this application is not only that it has low cost, but it could be done quickly and without risking human lives. Furthermore, using high-resolution cameras, they could be used as flying photographers providing aerial based videos [1]. Also, using a camera for obstacle detection on Micro Aerial Vehicles (MAVs) in unknown areas is still a big challenge [2].

To achieve autonomous flight of a UAV, its position and attitude have to be obtained. Detection and localization come with many challenges, not the least of which is the requirement to maximize the probability of detection of targets while minimizing the rate of false alarm. Flying behavior is able to land and start vertically, stay perfectly still in the air and move in any given direction at any time.

Most UAVs use an inertial measurement unit (IMU) to acquire the attitude and a global positioning system (GPS) to acquire the position. GPS relies on external source for providing vehicle global position information. It is an important tool, but sometimes it is not helpful. In instance, it is not helpful in indoor environments or even outdoors, if there are many obstacles. In addition, it is not reliable at low altitudes, suffers from satellite signal cuts and is a no passive sensing modality. As a result, the UAV loses the information about its location when the signal of GPS is blocked. Thus, to address this problem [3], authors have used visual information that is very important in collision avoidance, if there are a lot of obstacles.

Vision provides information with a pixel-order resolution and has a low-cost hardware. However, increasing video streaming rate can benefit to the control performance. UAVs, therefore, are growing with high resolution camera such as AR.Drones robotic platforms for similar applications [4].

2.1 Motivation

The prime motivation of this project is to develop a target tracking controller for a drone quadcopter, able to track a moving target e.g. a car or pedestrian from the bounding box coordinates obtained from a video feed and to follow it keeping a safe distance, on a state-of-the-art stm32 microcontroller.

2.2 Objectives

The objective of our work are as follows:

1. Implement an object tracking controller on a stm32 microcontroller in a 3-D space from 2-D video frames, that is, to develop an algorithm able to follow the movement of the target object through the image's height, width and depth information coming from the drone camera.
2. To control the drone maneuverability so that it follows the observations given by the stm32 flight-controller algorithm, creating a target-tracking implementation for the drone.
3. Verify and test the performance of the algorithm on a virtual open-source simulator called AriSim.

3 Project Overview

The desired quadcopter behavior is achieved by the interaction of two separate modules which provide two different functionalities: The first is the tracking of the target and the second is by giving high-level commands for the control logic for the quadcopter maneuverability. For the first task an object detection system is needed in order to find the target from the image obtained from the drone camera, and a tracking system is needed in order to follow this detected target. For the second task a general controller can be implemented from scratch using the quadcopter high-level commands to be sent to the drone low-level flight controller.

3.1 Bounding Box Information

The input of a target tracker is the bounding box (found by the detector) containing the target to track. To obtain this bounding box information an already trained object detector network is used. This can be done with the help of supervised learning and neural networks. In our case, the bounding box information containing target position coordinates is already provided as a log file with the input video feed dataset.

In general, bounding box coordinates have values that can exceed $2^8 - 1 = 255$ (for example, in a 1280×720 HD video, $0 \leq x_{min}, x_{max} \leq 1280$, and $0 \leq y_{min}, y_{max} \leq 720$). Hence, for transmission and reception, a coordinate can be split into two bytes – the least significant 8 bits of the binary representation of the coordinate form the lower (or least significant) byte, and the next 8 bits form the upper byte. For example, if $x_{max} = 527$, then $(527)_{10} = (1000001111)_2 = (00000010 \ 00001111)_2$. The lower byte is then $(00001111)_2 = (15)_{10}$ and the upper byte is $(00000010)_2 = (2)_{10}$.

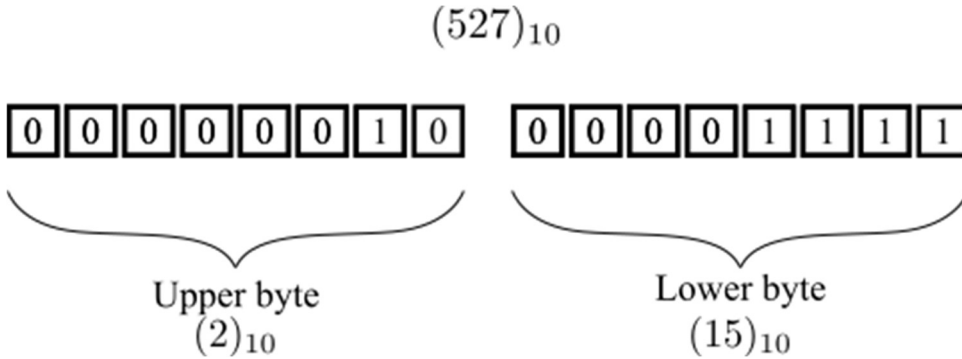


Figure 1 Coordinates Representation

Hence, for any coordinate x ,

Lower byte = remainder of x upon division by 2^8 (256)

Upper byte = quotient of x upon division by 2^8

And $x = (\text{upper byte}) \times 256 + (\text{lower byte})$

3.2 STM32 Nucleo Board

We are required to use a state-of-the-art STM32 Nucleo board for our flight-controller implementation [5]. This Nucleo board can be programming using the free STM32CubeIDE software. Some of the common features of the board included are:

- A 32-bit ARM microcontroller
- 3 user LEDs
- 2 user and reset push-buttons
- 32.768 kHz crystal oscillator
- Flexible power-supply options: ST-LINK, USB VBUS, or external sources
- On-board ST-LINK debugger/programmer with USB re-enumeration capability: mass storage, Virtual COM port, and debug port
- Ethernet compliant with IEEE-802.3-2002
- Board connectors: USB with Micro-AB or USB Type-C® Ethernet RJ45

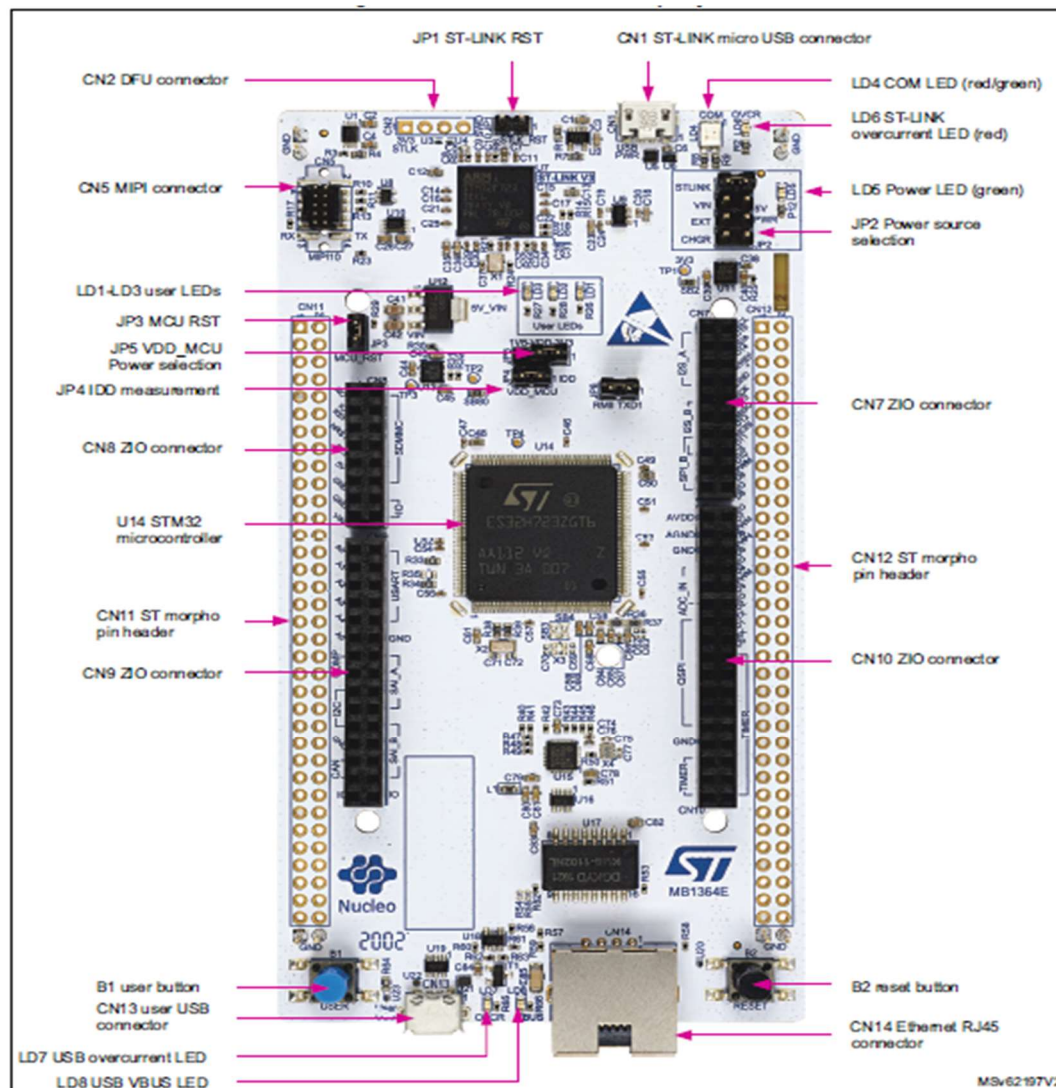


Figure 2 Nucleo-144 board layout

Microcontroller main key features are [6]:

Core

- 32-bit Arm® Cortex®-M7 core with double-precision FPU and L1 cache:
- 16 Kbytes of data and 16 Kbytes of instruction cache
- Frequency up to 400 MHz

Memories

- Up to 2 Mbytes of Flash memory with read-while-write support
- 1 Mbyte of RAM: 192 Kbytes of TCM RAM (inc. 64 Kbytes of ITCM RAM + 128 Kbytes of DTCM RAM for time critical routines), 864 Kbytes of user SRAM, and 4 Kbytes of SRAM in Backup domain
- Dual mode Quad-SPI memory interface running up to 133 MHz

General-purpose input/outputs

- Up to 168 I/O ports with interrupt capability

Low-power consumption

- Total current consumption down to 4 μ A

Clock management

- Internal oscillators: 64 MHz HSI, 48 MHz HSI48, 4 MHz CSI, 32 kHz LSI
- External oscillators: 4-48 MHz HSE, 32.768 kHz LSE
- 3 \times PLLs (1 for the system clock, 2 for kernel clocks) with Fractional mode

Up to 35 communication peripherals

- 4 \times I2Cs FM+ interfaces (SMBus/PMBus)
- 4 \times USARTs/4 \times UARTs (ISO7816 interface, LIN, IrDA, up to 12.5 Mbit/s) and 1 \times LPUART
- 6 \times SPIs, 3 with muxed duplex I2S audio class accuracy via internal audio PLL or external clock, 1 \times I2S in LP domain (up to 133 MHz)
- 4 \times SAls (serial audio interface)
- SPDIFRX interface
- SWPMI single-wire protocol master I/F
- MDIO Slave interface
- 2 \times SD/SDIO/MMC interfaces (up to 125 MHz)
- 2 \times CAN controllers: 2 with CAN FD, 1 with time-triggered CAN (TT-CAN)
- 2 \times USB OTG interfaces (1FS, 1HS/FS) crystal-less solution with LPM and BCD
- Ethernet MAC interface with DMA controller
- HDMI-CEC
- 8- to 14-bit camera interface (up to 80 MHz)

11 analog peripherals

- 3 \times ADCs with 16-bit max. resolution (up to 36 channels, 4.5 MSPS at 12 bits)
- 1 \times temperature sensor
- 2 \times 12-bit D/A converters (1 MHz)

- 2× ultra-low-power comparators
- 2× operational amplifiers (8 MHz bandwidth)
- 1× digital filters for sigma delta modulator (DFSDM) with 8 channels/4 filters

Up to 22 timers and watchdogs

- 1× high-resolution timer (2.5 ns max resolution)
- 2× 32-bit timers with up to 4 IC/OC/PWM or pulse counter and quadrature (incremental) encoder input (up to 200 MHz)
- 2× 16-bit advanced motor control timers (up to 200 MHz)
- 10× 16-bit general-purpose timers (up to 200 MHz)
- 5× 16-bit low-power timers (up to 200 MHz)
- 2× watchdogs (independent and window)
- 1× SysTick timer
- RTC with sub-second accuracy & HW calendar

Debug mode

- SWD & JTAG interfaces
- 4-Kbyte Embedded Trace Buffer

3.3 Quadcopter Basics

A quadcopter is a multirotor helicopter with four actuators (propellers), each providing a force in the body-fixed z-direction and a torque to the body. A general x-configuration representation in body-fix frame B is show in Figure 3 below.

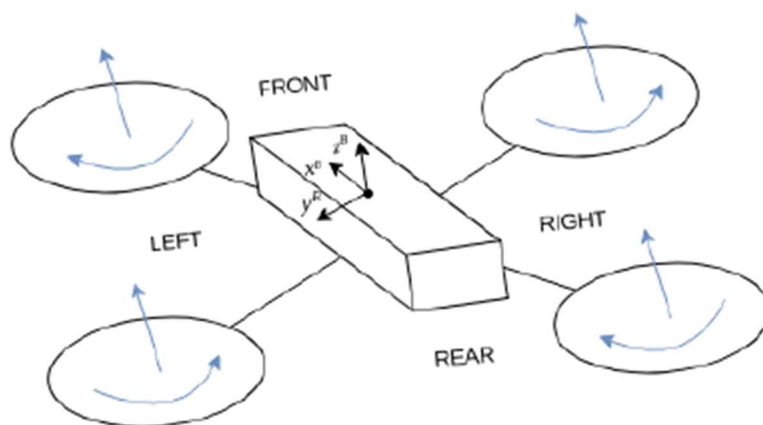


Figure 3 Quadcopter Representation

Movements are obtained by changing the roll, pitch and yaw angles, and by changing the vertical Speed. For example, moving forward, both the front rotors have to decrease their speed while increase the rear ones. The same concept applies for backward, left and right movements by changing the corresponding rotors speed as shown in Figure 4.

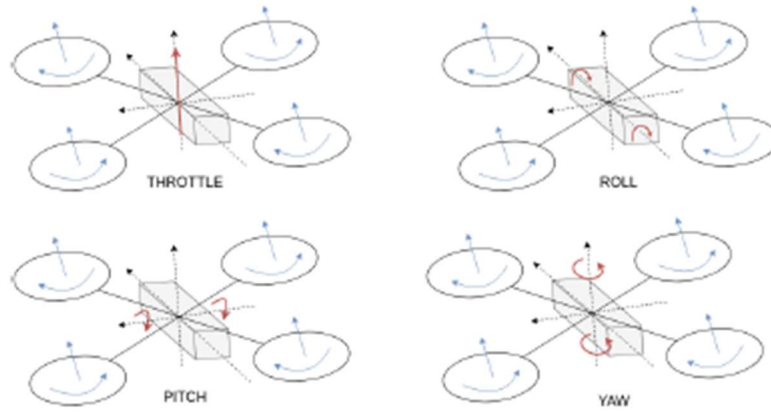


Figure 4 Roll, Pitch, Yaw and Throttle movements

All quadrotors have two coordinate frames: the inertial frame (earth-fixed frame) and the body-fixed frame. This represents a system of six degrees of freedom (x, y, z , roll, pitch, yaw), controlled by adjusting the rotational speeds of the four rotors. In addition to this, our system has four inputs i.e. U_1 : throttle, U_2 : roll, U_3 : pitch and U_4 : yaw to produce the 6-DOF output movements.

In this work, we are interested to implement a high-level controller on a stm32 microcontroller to support the onboard-flight computer (auto-pilot) for autonomous flight eliminating the need of external pilot.

4 Framework overview

Previously, we have seen physically how the quadcopter can be controlled by adjusting the speed of its rotors (propellers). Here, we will propose a framework to implement our project. To start with, we are provided with the object detection and tracking in the form of bounding box coordinates as output of drone camera video feed, we now need to create a software (controller algorithm) on the stm32 microcontroller able to keep track of a target through the quadcopter's movement, be able to recover from tracking failure, and move the quadcopter accordingly to the target's movements, with the use of open-source python API to easily send the commands to the drone in AirSim simulator. A flow diagram is shown in Figure 5.

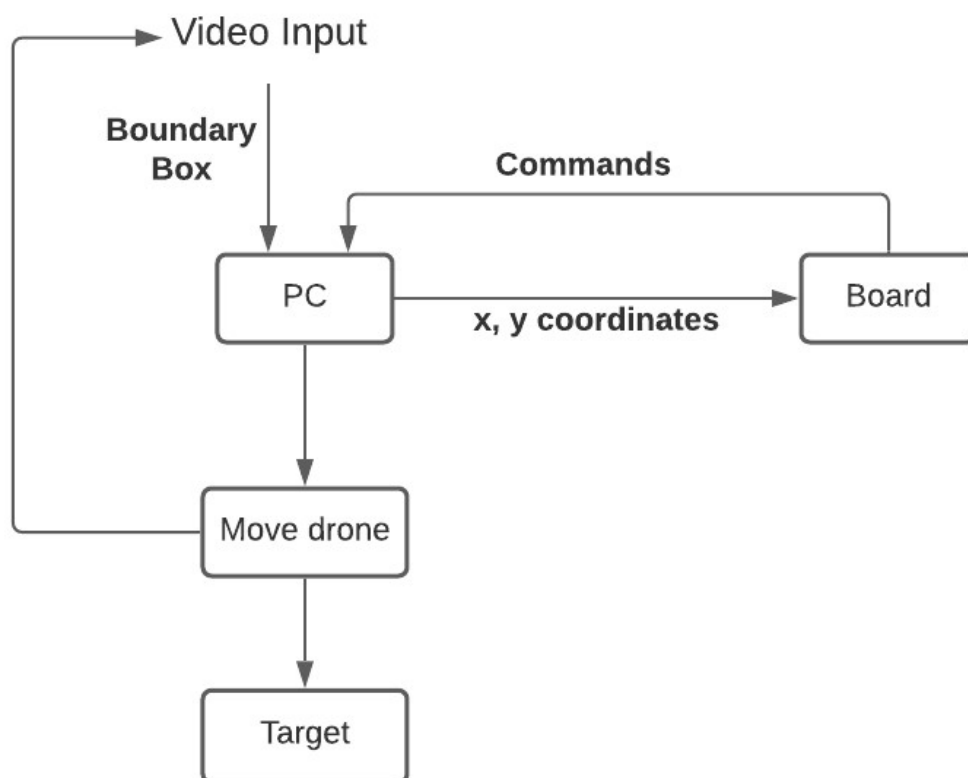


Figure 5 Proposed flow diagram

The drone sends every few milliseconds an image frame captured from its camera, which is collected as our dataset and is then passed to a PC where bounding box coordinates of the target are found of each frame, from which we will be able to send the commands to the drone that will make it go to a certain position. With the help of x, y coordinates and the target height information a flight-controller is implemented, to send commands for the drone movement.

We have used the following 5- integer array scheme to send commands to the drone as tabulated in table 1.

Table- I

Array Index	Integer Value	Commands
0	0	No movement in X-axis
	1	Move Forward at V_x
	2	Move Backward at $-V_x$
1	0	No movement in Y-axis
	1	Move Right at V_y
	2	Move Left at $-V_y$
2	0	No movement in Z-axis
	1	Move Down at V_z
	2	Move Up at $-V_z$
3	0	No Rotation
	1	Roll Right at D°
	2	Roll Left at $-D^\circ$
4	0	Ascending
	1	Landing

Where, $V_x = 2 \text{ m/s}$, $V_y = 2 \text{ m/s}$, $V_z = 5 \text{ m/s}$ and $D^\circ = 5 \text{ degrees}$

5 Software Implementation

In this section we will explain how the theoretical framework from the previous chapter has been implemented in C and Python.

5.1 Source code organization

The controller software for STM32 microcontroller created for this project is written in C while communication with the PC is done through serial port is written in python 3.7. We have used HAL library provided my STM32 for serial communication developed in STM32CubeIDE environment. The computer used is Dell Optiplex 5060 with Intel® Core™ i5-8400 CPU @ 2.80GHz. The file size for the implemented controller is 65 MB while the parsing of data and executing the commands done in python has a file size of under 10MB.

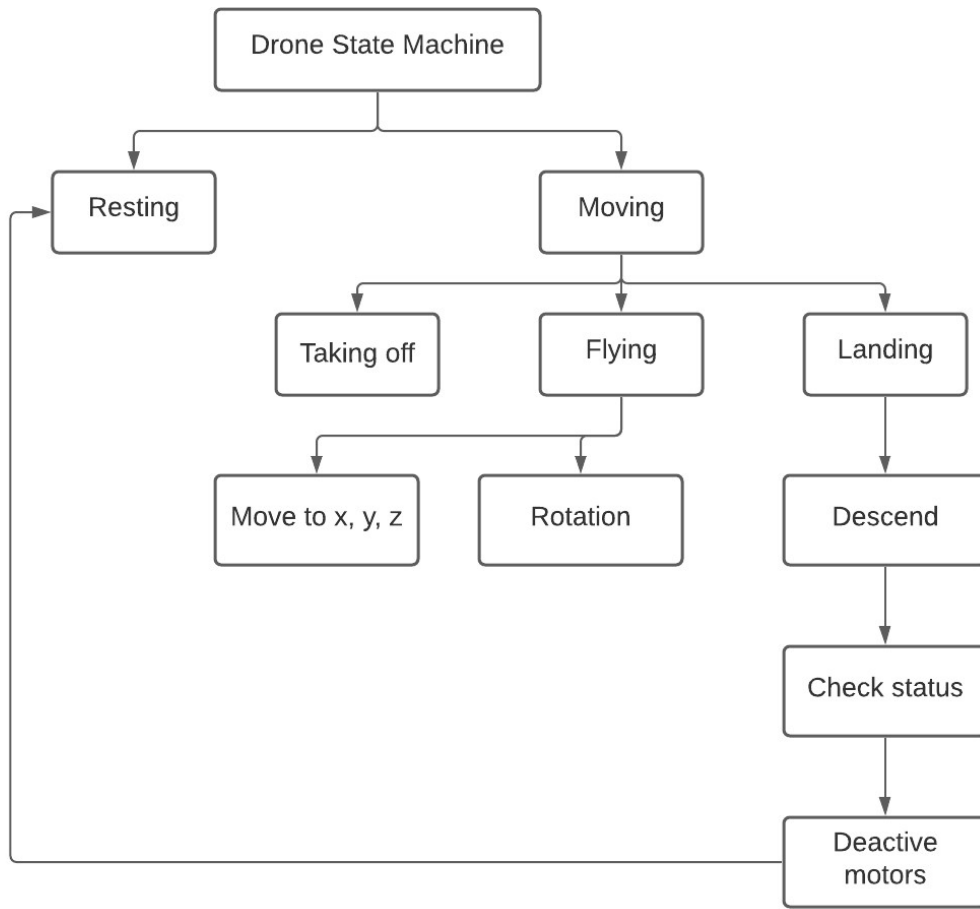


Figure 6 State Machine Implementation

5.2 Detail Software Explanation

In this section we will explain how the codes are implemented from the previous framework shown in Figure 5.

5.2.1 Sending Coordinates to STM32 board

As we had explained in the start we are provided with bounding box information as x , y coordinates. However, to send them from PC to the STM32 board we need to make the highest value less than 256. For this we form lower and upper bytes for each coordinate value. These lower and upper bytes are then transmitted by Python and received by the STM32 board. Python splits each of the coordinates into two bytes (there are thus 8 bytes in total for $x_{min}, y_{min}, x_{max}, y_{max}$) using the function `send_coords()` :

```
def send_coords(coords, port):
    dataTx = [0]*9
    dataTx[1], dataTx[0] = divmod(coords.x_min, 256)
    dataTx[3], dataTx[2] = divmod(coords.y_min, 256)
    dataTx[5], dataTx[4] = divmod(coords.x_max, 256)
    dataTx[7], dataTx[6] = divmod(coords.y_max, 256)
    dataTx[8] = SYNC_BYTE
```

(The function `divmod(a, b)` returns the quotient and remainder when `a` is divided by `b`.)

The `SYNC_BYTE` is for synchronizing. It indicates that a set of 8 bytes is now complete and to be transmitted. The 8 bytes (and the 9th `SYNC_BYTE`) are then sent to the STM32 board by the following snippet:

```
for i in range(len(dataTx)):
    if i < len(dataTx) - 1 and dataTx[i] == SYNC_BYTE:
        dataTx[i] -= 1
    if dataTx[i] < 0:
        dataTx[i] = 0;

port.write(bytes([dataTx[i]]))
```

We need to make sure that none of the bytes of the coordinates are equal to `SYNC_BYTE`. The above snippet takes care of this.

5.2.2 Reconstructed Coordinates

The 8 bytes are received by the STM32 board and the 4 coordinates are reconstructed by the following snippets:

For reconstruction, the function `get_bbox_coords()` is used (defined in `pycomm_qc.c`):

```
coords get_bbox_coords(uint8_t *Rx)
{
    // Two bytes form a coordinate. If a coordinate is x, then lower byte
    = x % 2^8 = x % 256,
    // and upper byte = quotient of x upon division by 2^8 = floor(x /
    256). So, for reconstructing
    // x, we use the formula x = 256*(upper byte) + (lower byte)
    coords XY;
    XY.x_min = Rx[1]*256 + Rx[0];
    XY.y_min = Rx[3]*256 + Rx[2];
    XY.x_max = Rx[5]*256 + Rx[4];
    XY.y_max = Rx[7]*256 + Rx[6];
    return XY;
}
```

For reception, the following snippet is used (in `stm32h7xx_it.c`):

```
uint8_t ch; //Byte received
```

```
HAL_UART_Receive(&huart3, &ch, 1, 100); //For receiving a single byte.
Byte by byte reception
```

```
dataRx[count] = ch;
count++;
```

When STM32 receives SYNC_BYTE, it understands that a complete array of 8 bytes has been received, and then it performs the reconstruction and other calculations based on the bounding box coordinates. As per the calculations, it creates the command integer array (containing 5 integers: the 1st integer denoting command about motion along x-axis, 2nd integer denoting command about motion along y-axis, 3rd integer denoting command about motion along z-axis, 4th integer denoting command about rotation, and 5th denoting command about landing). And then it sends the command integer array (stored as `dataTx`) to Python. This is handled by the following snippet:

```
if (ch == SYNC_BYTE)
{
    //Reception of 4 coordinates (as 8 bytes) ended
    count = 0;
    bbXY = get_bbox_coords(dataRx);
    //Computation of quadcopter quantities based on bounding box
coordinates and setting
    //the command integer array (having 5 integers as 5 bytes) in
dataTx
    compute(bbXY, dataTx);

    //Transmitting the command 5-integer array (as 5 bytes) to
Python
    HAL_UART_Transmit(&huart3, dataTx, 5, 100);
```

5.2.3 Using STM32 Interrupts

There are three possible ways to receive data with serial port via universal asynchronous receiver-transmitter (UART).

1. Polling
2. Interrupt
3. DMA

From the above methods, only the interrupt methods worked properly for duplex communication on the STM32 board. We tried with both other methods, i.e. polling and DMA but they didn't work after initial few executions.

For this, Interrupt handling is done using the functions `__HAL_UART_ENABLE_IT(huart, UART_IT_RXNE)` and `USART3_IRQHandler()`. The function `__HAL_UART_ENABLE_IT(huart, UART_IT_RXNE)` enables interrupts, and is called inside the function `HAL_UART_MspInit(UART_HandleTypeDef* huart)` defined in the file `stm32h7xx_hal_msp.c`.

```

void HAL_UART_MspInit(UART_HandleTypeDef* huart)
{
    ...
    ...
    ...

    /* USART3 interrupt Init */
    HAL_NVIC_SetPriority(USART3_IRQn, 0, 0);
    HAL_NVIC_EnableIRQ(USART3_IRQn);
    /* USER CODE BEGIN USART3_MspInit 1 */
    //Enabling interrupts
    __HAL_UART_ENABLE_IT(huart, UART_IT_RXNE);

    /* USER CODE END USART3_MspInit 1 */
}
}

```

The activities to be performed when an interrupt is triggered (due to the reception of a byte by the STM32 board) are coded in the function `USART3_IRQHandler()`. This function is defined in the file `stm32h7xx_it.c`.

```

void USART3_IRQHandler(void)
{
    /* USER CODE BEGIN USART3_IRQn 0 */

    uint8_t ch; //Byte received

    HAL_UART_Receive(&huart3, &ch, 1, 100); //For receiving a single byte.
    Byte by byte reception

    dataRx[count] = ch;
    count++;
    if (ch == SYNC_BYTE)
    {
        //Reception of 4 coordinates (as 8 bytes) ended
        count = 0;
        bbXY = get_bbox_coords(dataRx);
        //Computation of quadcopter quantities based on bounding box
        coordinates and setting
        //the command integer array (having 5 integers as 5 bytes) in
        dataTx
        compute(bbXY, dataTx);

        //Transmitting the command 5-integer array (as 5 bytes) to
        Python
        HAL_UART_Transmit(&huart3, dataTx, 5, 100);
    }
    if (count > 8)
        //This is to prevent overflow in case SYNC_BYTE is missed for
        some reason
        count = 0;

    return; //For bypassing the HAL_UART_IRQHandler() function below.
    That function causes problems.

    /* USER CODE END USART3_IRQn 0 */
    HAL_UART_IRQHandler(&huart3);
}

```



```

    /* USER CODE BEGIN USART3_IRQn 1 */

    /* USER CODE END USART3_IRQn 1 */
}

```

The STM32 board receives a byte in the variable `ch` and stores it in the array `dataRx`. The board receives one byte at a time and keeps on receiving bytes from Python until it receives the `SYNC_BYTE`. Then it reconstructs the coordinates (`bbXY = get_bbox_coords(dataRx);`) and performs the bounding box calculations (`compute(bbXY, dataTx);`) to get the command 5-integer array and stores it in `dataTx`. Then it sends `dataTx` to Python. Then it again starts receiving bytes from Python, and the cycle goes on.

5.2.4 Sending Drone Commands back to PC

Python receives the command 5-integer array from the STM32 board using the function `receive_integers(port)` as defined below

```

def receive_integers(port):
    int_array = [0]*5
    for i in range(5):
        x = port.read()
        int_array[i] = int.from_bytes(x, "big")

    return int_array

```

Python gets the coordinates in the first place by reading them from a text file. This is done by the following snippet:

```

coords_list = []
with open("output.txt", "r") as input_fs:
    for line in input_fs:
        contents = line.split(",")
        integers = [int(x) for x in contents]
        coords_list.append(integers)

```

Python stores those coordinates in the list `coords_list`. `coords_list[i]` has the $(i + 1)$ -th set of coordinates (corresponding to the $(i + 1)$ -th line of the text file). To facilitate the handling of the coordinates, we have defined a class `BBoxCoords`. An object of this class will have four variables `xmin`, `ymin`, `xmax`, `ymax`. The entire splitting of the coordinates, transmission of the bytes and reception of the command integer array by Python is handled by the following snippet:

```

for idx in range(len(coords_list)):
    bbcoords.set(coords_list[idx])
    send_coords(bbcoords, ser)
    dataRx_int = receive_integers(ser)
    commandIntList.append(dataRx_int)
    print_commands(dataRx_int)

```

Also note that for some reason, if we make SYNC_BYTE equal to 0x00, the data reception by STM32 does not work properly. So, we need to use a non-zero value of SYNC_BYTE to make the data transmission work.

6 Results

Here we will see the show the performance and overall memory utilization.

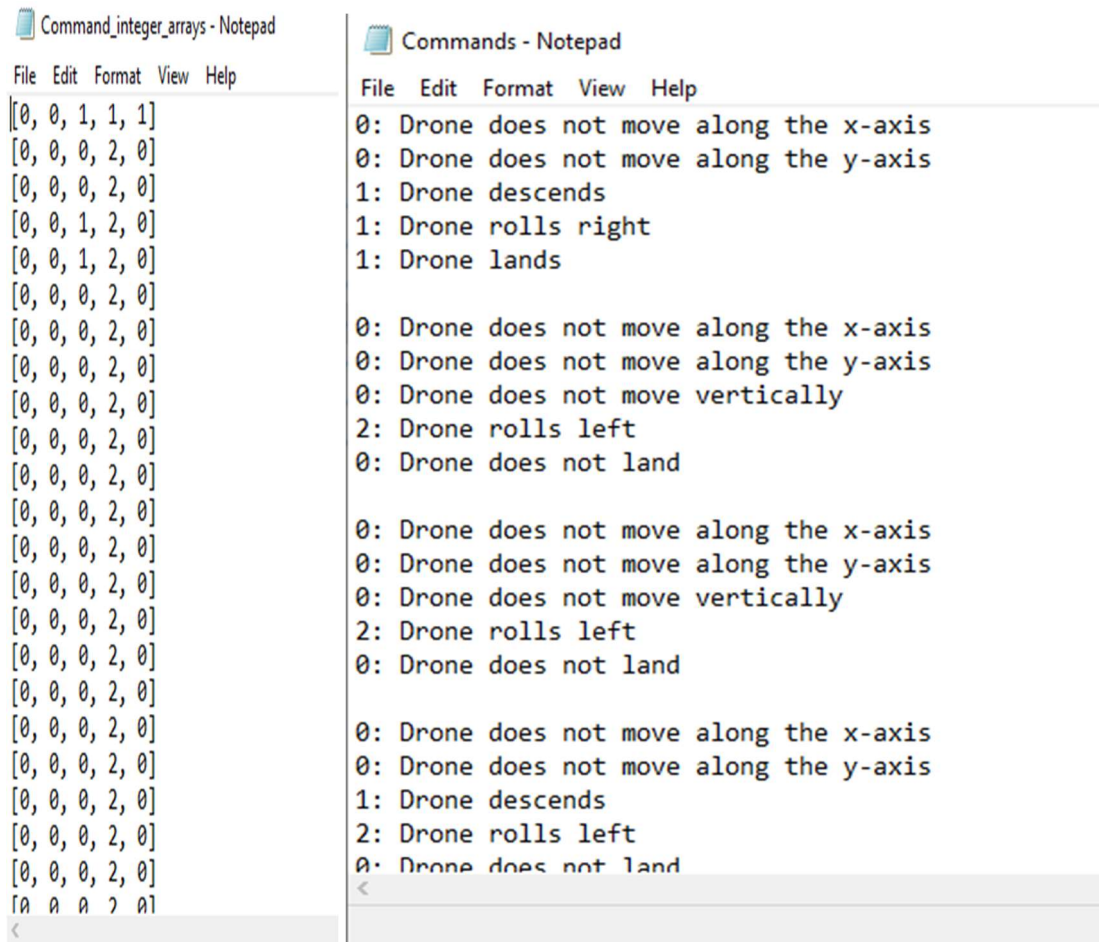
6.1 Performance

We have chosen the AirSim simulator by Microsoft in which we can see how the drone is moving corresponding to the target bounding box coordinates. This is shown in a virtual block environment as shown in Figure 7 below.









Figure 7 AirSim Environment

The corresponding drone commands and integer array values are stored in two separate log files as shown in Figure 8. For each calculated output integer array values the set drone commands are executed which can be seen in the above AirSim simulator.



6.2 STM32 Memory Utilization

Memory Regions		Memory Details				
Region	Start address	End address	Size	Free	Used	Usage (%)
 FLASH	0x08000000	0x08200000	2 MB	1.97 MB	30.06 KB	1.47%
 RAM	0x20000000	0x20020000	128 KB	124.77 KB	3.23 KB	2.53%
 RAM_D1	0x24000000	0x24080000	512 KB	512 KB	0 B	0.00%
 RAM_D2	0x30000000	0x30048000	288 KB	288 KB	0 B	0.00%
 RAM_D3	0x38000000	0x38010000	64 KB	64 KB	0 B	0.00%
 ITCMRAM	0x00000000	0x00010000	64 KB	64 KB	0 B	0.00%

7 Conclusion

This section is divided in two different parts. First, we will analyse the work done and then we introduce some improvements that can be done following the framework implemented here.

7.1 Summary and Complications

We have implemented a target-tracking controller algorithm for a quadcopter drone on an embedded STM32 Nucleo board platform. Moreover, an implementation of the software designed controller has been developed to follow the target while keeping a safe distance. The target-tracking quadcopter presented is able to track a target through the bounding box coordinates in an indoor space or an outdoor space with a set threshold for obstacle clearance.

In the proposed framework, we have developed all the high-level calculations on the STM32 microcontroller for generating commands to fly the drone. In order to have the low level visualization of drone flight, we have tested our embedded controller performance in an AirSim simulator on three different dataset video feed collected from real drone. Our strategy, shows good agreement and can be implemented on a real drone.

7.2 Future Work

The implemented framework of this project can be further improved and extended. Due to limitation of physical team work, validation of simulated results on more datasets, have left space to fulfil more objectives. We present some points that can be used to carry on this work:

- Implementing error handling capability, to include possible failure error incident and avoid malfunction of our system and making it more robust
- An improved protocol scheme can be devised for negative coordinate values (when target is outside or the bounding box or at the edge)
- Fail safe mode for any disconnection between the STM32 board and the main drone flight controller

8 References

- [1] D. T. H. Z. X. X. ,. H. Yong Zhou, «Vision-based Online Localization and Trajectory Smoothing for Fixed-wing UAV Tracking a Moving Target,» 2019.
- [2] D. H. a. A. N. Hsiu-Min Chuang, «Autonomous Target Tracking of UAV Using High-Speed Visual Feedback,» *Applied Sciences*, 9 2019.
- [3] V. L. a. B. S. Rafik Mebarki, «Nonlinear Visual Control of Unmanned Aerial Vehicles in GPS-Denied Environments,» *IEEE TRANSACTIONS ON ROBOTICS*, τόμ. 31, αρ. 4, pp. 1004-1017, 2015.
- [4] K. B. a. C. Larbes, «Detection and Implementation Autonomous Target Tracking with a Quadrotor AR.Drone,» 2015.
- [5] [Ηλεκτρονικό]. Available: <https://www.st.com/en/evaluation-tools/nucleo-h743zi.html>.
- [6] [Ηλεκτρονικό]. Available: <https://os.mbed.com/platforms/ST-Nucleo-H743ZI/>.

9 Appendix

The following C code named pycomm_qc.c is the main implemented controller calculations for the quadcopter movement performed on the STM32 board that is passed to the python PC side.

9.1 User defined C library

```
/*
 * pycomm_qc.c
 *
 * Created on: Jul 25, 2021
 * Author: afaroo01
 */

#include "pycomm_qc.h"
#include <time.h>
#include <math.h>

#define FOCAL_LENGTH 1002.907
#define WIDTH 0.22
#define SCALE_PERCENT 100
#define DIST_THRESH_FAST 1.5
#define DIST_THRESH_SLOW 0.5
#define LAND_THRESH 100
#define FAST_FACTOR 2
#define SLOW_FACTOR 0.5
#define MIN_VEL 0.00001

const int imgWidth = 960*SCALE_PERCENT/100;
const int imgHeight = 720*SCALE_PERCENT/100;

coords get_bbox_coords(uint8_t *Rx)
{
    coords XY;
    XY.x_min = Rx[1]*256 + Rx[0];
    XY.y_min = Rx[3]*256 + Rx[2];
    XY.x_max = Rx[5]*256 + Rx[4];
    XY.y_max = Rx[7]*256 + Rx[6];
    return XY;
}

float calculateDistance(unsigned int boxWidth)
{
    if(boxWidth < 1)
        boxWidth = 1;
    //Can modify
    float distance = WIDTH*FOCAL_LENGTH/(boxWidth*1.0);
    return distance;
}
```



```

int calculateRotationAngle(unsigned int x_min, unsigned int boxWidth,
                          unsigned int imageWidth)
{
    int theta;
    //Can modify
    //theta = (int)(((82.6*x_min + 41.3*boxWidth)/(imageWidth*1.0)) -
41.3);
    theta = (int)(((82.6*x_min + 41.3*boxWidth)/(imageWidth*1.0)) - 4.3);
    return theta;
}

float calculateVelocityOfPlatform(unsigned int distance, unsigned int
lastKnownDistance,
                                float velocityTimer)
{
    //Can modify
    float velocity = (distance-
lastKnownDistance)/(clock()*1.0/CLOCKS_PER_SEC-velocityTimer);
    return velocity;
}

int rotationDrone(unsigned int x_min, unsigned int boxWidth,
                  unsigned int imageWidth, uint8_t* commandIntArray)
{
    int theta = calculateRotationAngle(x_min, boxWidth, imageWidth);
    if (theta > 2)
    {
        //commandIntArray[1] = 1; // Roll right
        commandIntArray[3] = 1;      // Roll right
    }

    else if (theta < -2)
    {
        //commandIntArray[1] = 2; // Roll left
        commandIntArray[3] = 2;      // Roll left
    }
    else
    {
        //commandIntArray[1] = 0; // No roll
        commandIntArray[3] = 0;      // No roll
    }
    return theta;
}

float horizontalMovementDrone(float distance, float platformVelocity,
                              unsigned int firstCalculation, float droneVelocity, int theta,
                              uint8_t* commandIntArray)
{
    if (!firstCalculation)
    {
        //Can modify
        if(distance > DIST_THRESH_FAST)
            droneVelocity = droneVelocity +
platformVelocity*FAST_FACTOR;
        else if(distance <= DIST_THRESH_FAST && distance >
DIST_THRESH_SLOW)
            droneVelocity = droneVelocity + platformVelocity;
        else if (distance > 0)

```

```

        droneVelocity = droneVelocity +
platformVelocity*SLOW_FACTOR;
    else
        droneVelocity = 0;
}
else
    droneVelocity = 0;

float droneVelocityX = droneVelocity*cos(theta*3.1416/180);
float droneVelocityY = droneVelocity*sin(theta*3.1416/180);

if (droneVelocityX >= MIN_VEL)
    commandIntArray[0] = 1; //Forward motion
else if (droneVelocityX <= -MIN_VEL)
    commandIntArray[0] = 2; //Backward motion
else
    commandIntArray[0] = 0; //No motion

if (droneVelocityY >= MIN_VEL)
    commandIntArray[1] = 1; //Right motion
else if (droneVelocityY <= -MIN_VEL)
    commandIntArray[1] = 2; //Right motion
else
    commandIntArray[1] = 0; //No motion

return droneVelocity;
}

void verticalMovementDrone(unsigned int y_max, unsigned int imageHeight,
float distance,
uint8_t* commandIntArray)
{
    // Can modify
    if (distance > DIST_THRESH_FAST)
    {
        commandIntArray[2] = 0; //No vertical motion
        commandIntArray[4] = 0; //No landing
        if (y_max == 0)
            commandIntArray[2] = 0; //No vertical motion
    }
    else if (distance >= DIST_THRESH_SLOW)
    {
        if (y_max > 0)
        {
            commandIntArray[2] = 1; //Move downwards
        }
        else
            commandIntArray[2] = 0; //No vertical motion

        commandIntArray[4] = 0; //No landing
    }
    else
    {
        //Decreasing LAND_THRESH will make the drone approach the
platform closer (vertically) before trying to land
        if(y_max > imageHeight - LAND_THRESH)
        {
            commandIntArray[2] = 1; //Move downwards
            commandIntArray[4] = 1; //Land

```

```

        }
        else if(y_max > 0)
        {
            commandIntArray[2] = 1;    //Move downwards
            commandIntArray[4] = 0;    //No Land
        }
//        else
//        {
//            commandIntArray[2] = 2;    //Move upwards
//            commandIntArray[4] = 0;    //No Land
//        }
    }
}

static float platformVelocity = 0;
static float droneVelocity = 0;
static float lastKnownDistance = 0;
static unsigned int firstVelocityCalculation = 1;
static float velocityTimer = -1;
float distance;
int th;

void compute(coords bbXY, uint8_t* commandIntArray)
{
    //    static float platformVelocity = 0;
    //    static float droneVelocity = 0;
    //    static float lastKnownDistance = 0;
    //    static unsigned int firstVelocityCalculation = 1;
    //    static float velocityTimer = -1;

    unsigned int boxWidth = bbXY.x_max - bbXY.x_min;
    unsigned int boxHeight = bbXY.y_max - bbXY.y_min;
    //float distance = calculateDistance(boxWidth);
    distance = calculateDistance(boxWidth);

    if (clock()*1.0/CLOCKS_PER_SEC - velocityTimer > 1 || velocityTimer
== -1)
    {
        platformVelocity = calculateVelocityOfPlatform(distance,
lastKnownDistance, velocityTimer);
        velocityTimer = clock()*1.0/CLOCKS_PER_SEC;
        lastKnownDistance = distance;
    }

    th = rotationDrone(bbXY.x_min, boxWidth, imgWidth, commandIntArray);
    droneVelocity = horizontalMovementDrone(distance, platformVelocity,
firstVelocityCalculation, droneVelocity, th,
        commandIntArray);
    firstVelocityCalculation = 0;
    verticalMovementDrone(bbXY.y_max, imgHeight, distance,
commandIntArray);
}

```

9.2 Initial Experiments and Complications

In the first steps, we try to learn more about the board. We downloaded tutorial manuals and started to understand the pins and the ports on the STM32 Nucleo board as shown below.

LEDs

User LD1: a green user LED is connected to the STM32H7 I/O PB0 (SB39 ON and SB47 OFF) or PA5 (SB47 ON and SB39 OFF) corresponding to the ST Zio D13.

User LD2: a yellow user LED is connected to PE1.

User LD3: a red user LED is connected to PB14.

These user LEDs are on when the I/O is HIGH value, and are off when the I/O is LOW.

LD4 COM: the tricolor LED LD4 (green, orange, red) provides information about ST-LINK communication status. LD4 default color is red. LD4 turns to green to indicate that communication is in progress between the PC and the STLINK-V3E, with the following setup:

- Slow blinking red/OFF at power-on before USB initialization
- Fast blinking red/OFF after the first correct communication between PC and STLINK-V3E (enumeration)
- Red LED ON when the initialization between the PC and STLINK-V3E is complete
- Green LED ON after a successful target communication initialization
- Blinking red/green during communication with the target
- Green ON communication finished and successful
- Orange ON communication failure

LD5 PWR: the green LED indicates that the STM32H7 part is powered and +5 V power is available on CN8 pin 9 and CN11 pin 18.

LD6 USB power fault: LD5 indicates that the board power consumption on USB exceeds 500 mA, consequently the user must power the board using an external power supply.

LD7 and LD8 USB FS: Refer to [Section 6.6.6: USB OTG FS](#).

Figure 10 LED and Board Connection

The ST website supports two software, the Cube MX and the Cube IDE. We downloaded the ST-Link software tool for establishing the board and serial port communication. Then with the Cube MX we found the board that we going to use and then we started to “play” with the board.

We can configure the clock speed and other parameters and then export our results into cube ide. In the cube ide you can find all the code that export from your selections in the cube mx. This code has all the coding that we need to set up all the inputs and outputs. After that we were able to flash the led in the board as shown in Figure 11.

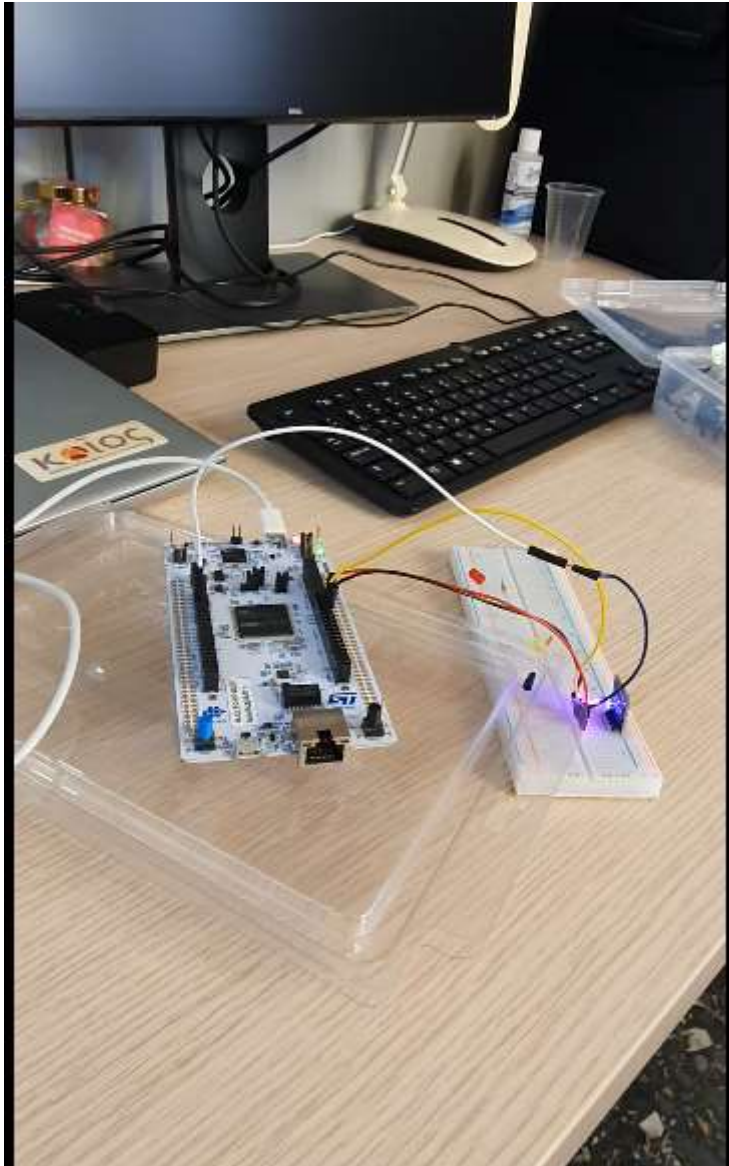


Figure 11 External RGB LED Changing Colors

After this we tried to send data from the computer into the board via Anaconda which is an application that create an environment in the computer able to run python scripts. We try different script and after we found how to send data from computer into the board. After trying with different ways, we were finally able to establish the duplex serial communication shown in Figure 12 connected with the acronymous "UART".

Pin name	Function	Virtual COM port (default configuration)	ST morpho connection
PD8	USART3 TX	SB5 ON and SB7 OFF	SB5 OFF and SB7 ON
PD9	USART3 RX	SB6 ON and SB4 OFF	SB6 OFF and SB4 ON

Table 11. LPUART1 connection

Pin name	Function	Virtual COM port	ARDUINO [®] D0 and D1	ST morpho connection
PB6	LPUART1 TX	SB9 ON SB8 and SB18 OFF	SB8 ON SB9 and SB18 OFF	SB9 OFF and SB18 OFF
PB7	LPUART1 RX	SB34 ON SB12 and SB68 OFF	SB68 ON SB34 and SB68 OFF	SB12 OFF and SB34 OFF

Hardware connection required for USART bootloader:

The STM32H7x3 embeds a USART bootloader. To use the USART bootloader (USART1), hardware modifications are required on the NUCLEO board. Flying wires have to be connected between PD8/PD9 (USART3 available on SB19/SB12) and PB10/PB11 (USART1 available on CN15).

Figure 12 Nucleo STM327Z1 UART Connection