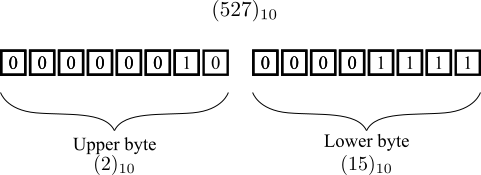
Bounding box coordinates have values that can exceed (for example, in a HD video, , and ). Hence, for transmission and reception, a coordinate is split into two bytes – the least significant 8 bits of the binary representation of the coordinate form the lower (or least significant) byte, and the next 8 bits form the upper byte. For example, if , then . The lower byte is then and the upper byte is .



Hence, for any coordinate ,

Lower byte = remainder of upon division by (256)  
Upper byte = quotient of upon division by

And (upper byte)256 + (lower byte)

These lower and upper bytes are 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 ) 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.

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);

}

Dd

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.

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.