Documentation: Audit Protis DMX/MIDI

ALICE RENAUX, CLARICE VELLOSO, HENRIQUE SILVESTRE,
MATHEUS BELISARIO AND ZIYAD BENKHADAJ
Institut d'Optique Graduate School

7 avril 2023

We certify that this work is original, that we cite by reference all the sources used and that it does not contain plagiarism.

1. OBJECTIVE

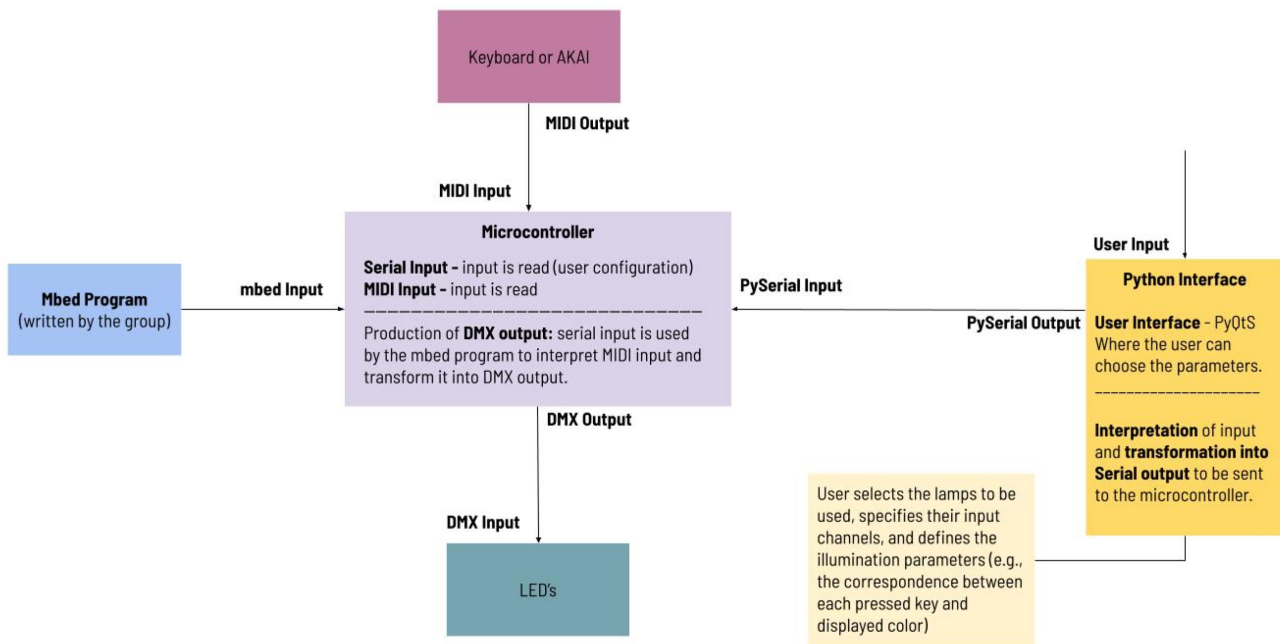

Figure 1 : *Project Schema*

We recall that the objective of this project is to develop a light control system using musical tools. In other words, our work concerns the creation of a User Interface (UI) able to manage the communication between two musical equipments, a MIDI piano keyboard and an AKAI MPD32, and two groups of four light lamps (Renkforce LV-PT12 and Eurolite PARty LED) with help of a NUCLEO-L476RG card.

To go into details, we can see the project schema in the Figure 1. The user interacts directly with the UI and one of two musical tools - our project is restrict to control just one peripheral at a time. To put it simply, the user can select two different colors in the UI and send the information to the NUCLEO card. After reading the configuration defined by the user, the card will be ready to read any data coming from the keyboard as MIDI input and convert it as DMX output to communicate with the lamps and turn on the lights in the specified colors.

A priori, we were supposed to create a User Interface (UI) with 3 preset commands : Scriabin, MIDI Controller, and Sequencer. In the first one, each MIDI data sent would represent a color in the lamps ; in the

second, each MIDI data would correspond to a specific output pattern across all the spots, be a sequence of colors that repeat with a given time, or just the lights flashing with certain frequency ; and the last one, we would preset 16 different patterns that would respond to the interaction speed of the user.

Unfortunately, we did not manage to integrate any complete mode with our UI within the time limits, although a partial Scriabin mode was possible - we managed to independently control two sets of notes from the keyboard, and not 12 (the number of notes in a piano octave) as it would be ideal. On the other side, we did manage to establish the Scriabin mode with direct connection between NUCLEO and the keyboard and also with the AKAI. We had difficulty accessing the button addresses of the AKAI, and for a while could not convert them into DMX data to turn the lights on, but we managed to find a solution before the end of the project.

2. Description

In this section of this work we are interested in detail every evolution step of our project. In general, we can separate four main working blocks : Reading MIDI signal, Sending DMX signals, Python coding, and Independent Mbed program.

As detailed in Figure 1, there are 5 blocks composing our project. To tackle our objectives, we first started studying the communications between blocks, mainly the MIDI and the DMX. After that, the group started to study the construction of a graphic interface and finally, the communication between the interface and the micro-controller.

2.1. Reading MIDI signal

The MIDI signal is the one sent by the keyboard and the AKAI controller the group had available. This signal consists in an hexadecimal number that corresponds to the key that was pressed on the MIDI controller of choice. To read this, it is necessary to have the MIDI controller connected to the micro-controller MIDI connection. Besides that, it is necessary to define a Serial midi(D8, D2) ; entrance on the code.

A diagram showing the MIDI signal output from the keyboard can be seen in Figure 2.

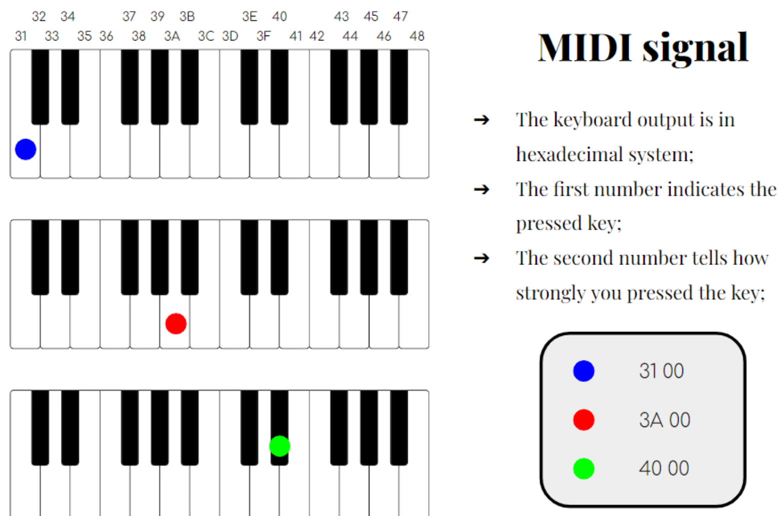

Figure 2 : *Diagram of the signal output from the keyboard.*

The code to read the MIDI note was by created Julien VILLEMEJANE, and adapted by the group to be used for the AKAI controller. The functions defined on this code read the MIDI input using Serial :

```
/* Initialization of the MIDI connection */
void initMIDI(void){
    midi.baud(31250);
```

```
    midi.format(8, SerialBase::None, 1);
    midi.attach(&ISR_midi_in, Serial::RxIrq);

/* Reading the MIDI note and the velocity that the key was pressed */
void playNoteMIDI(char note, char velocity){
    midi.putc(MIDI_NOTE_ON);
    midi.putc(note);
    midi.putc(velocity);
}

/* Prints the MIDI note on TeraTerm */
void printNoteMIDI(char note, char velocity){
    debug_pc.printf("%x %x \r\n", note, velocity);
}
```

Listing 1 : *C++ functions that initialize the MIDI connection; read the MIDI input and print it on TeraTerm*

This part was not altered by the group for reading the input from the keyboard, but changes were necessary to use the AKAI controller. To read the input, we used this part of the overall Mbed code and displayed the text in TeraTerm, which is a software used to manage several types connections and display input messages from different types of equipment. This part was used to initiate the connection between TeraTerm and the AKAI :

```
    debug_pc.baud(115200);
    debug_pc.printf("Test\r\n");
```

with *debug_pc* being defined as so :

```
    Serial          debug_pc(USBTX, USBRX);
```

And the MIDI data was display in TeraTerm wia the function *printNoteMIDI*. It is important that the connection speed the argument of *debug_pc.baud* is the same between the code and TeraTerm. Here, it is 115200*baud*, with *baud* being the unit of measurement of symbol rate. The Serial connection can be set up this way :

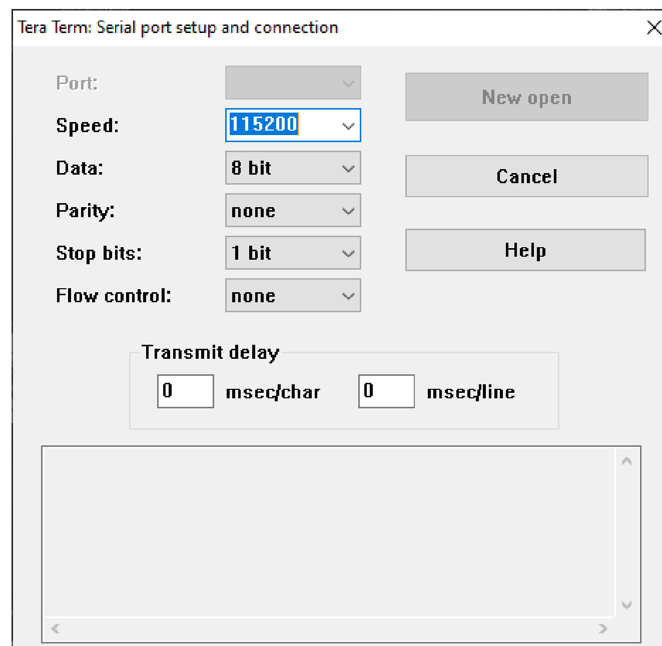

Figure 3 : *The TeraTerm Serial connection setup window*

In the beginning, when testing our code to read MIDI signals sent to the Nucleo card, we noticed that the MIDI data (the note, which is similar to the address for the DMX, and the velocity) of only 8 pads out of 16 were being read. In order to get the other MIDI addresses, we used a second software called "Beyond", which is

a laser control software edited by Pangolin Laser Systems. It is used by the LaserWave, which is the lights and lasers association of SupOptique, and allows to programm MIDI controllers to control lights, smoke machines, stroboscopes and, of course, lasers.

It appeared that the pads belonged to two channels instead of one and that the Mbed code only read the MIDI addresses of the first channel. Indeed, a button belonging to the first channel will have an address that begins with "90" when it is being pushed and "80" when it is not, whereas the addresses for a button belonging to the second channel begin with "91" or "81".

We asked Mr VILLEMEJANE if he had any idea on how to fix that, and he kindly helped us. So, after changing the code, the function that read the MIDI addresses ("notes") went from this :

```
void ISR_midi_in(void){
    debug_out = !debug_out;
    char data = midi.getc();
    if(data >= 128)
        cpt_midi = 0;
    else
        cpt_midi++;
    midi_data[cpt_midi] = data;
    if(cpt_midi == 2){
        cpt_midi = 0;
        if((midi_data[0] == MIDI_NOTE_ON) || (midi_data[0] == MIDI_NOTE_OFF)){
            new_note_midi = 1;
            note_data = midi_data[1];
            velocity_data = midi_data[2];
        }
        else{
            if(midi_data[0] == MIDI_CC){
                new_data_midi = 1;
                control_ch = midi_data[1];
                control_value = midi_data[2];
            }
        }
    }
}
```

to this :

```
void ISR_midi_in(void){
    debug_out = !debug_out;
  // R cup ration de la donn e sur la liaison s rie
    char data = midi.getc();
  // D tection du premier octet (valeur > 128 d'apr s le protocole MIDI)
    if(data >= 128)
        cpt_midi = 0;
    else
        cpt_midi++;
  // Stocktage de la donn e re ue dans une des 3 cases du tableau midi_data
    midi_data[cpt_midi] = data;

  message_type = midi_data[0] & 0xF0;
  midi_channel = midi_data[0] & 0x0F;

    if(cpt_midi == 2){
        cpt_midi = 0;
        if((message_type == MIDI_NOTE_ON) || (message_type == MIDI_NOTE_OFF)){
            new_note_midi = 1;
            note_data = midi_data[1];
            velocity_data = midi_data[2];
        }
        else{
            if(message_type == MIDI_CC){
                new_data_midi = 1;
                control_ch = midi_data[1];
                control_value = midi_data[2];
```

```
                }
            }
        }
}
```

It was a matter of modifying the mask used to read the character used to determine the channel (the "0" in "90" and "80"). Hence the first channel is coded by a "0", the second channel is coded by a "1", and so on.

A second option is to modify the addresses on the AKAI itself with the "Edit presets" mode. And we learnt it the hard way : just before out presentation, our prototype was not working, and it turned out that the addresses of the AKAI pads were not the same as in our code anymore, so we quickly fixed that.

2.2. Sending DMX signals

The second step consisted in understanding how to send a DMX signal to the spots. The DMX technology works as so : for a LED, each color and the overall brightness, as well as other features depending on the type of LED, is controlled by a DMX channel identified by a number between 1 and 512 ("DMX channel"), and each spot carries the number of its first channel ("DMX address"). Each DMX channel receives a value between 0 and 255 that gives the intensity of the light.

The way the channels are coupled with the colors and the overall brightness varies from one type of spot to another. In our situation, the channels and the colors are coupled as explained below :

Function	Renkforce LV-PT12	Eurolite PARty LED
RGB Activation	Channel 1 (value must be 00-10)	-
Red	Channel 5	Channel 2
Green	Channel 6	Channel 3
Blue	Channel 7	Channel 4
Overall brightness	Channel 4	Channel 1

Table 1 : *Coupling between the DMX channels and the colors*

We started by using the Renkforce LEDs. In the code we used, the DMX connection was already configured, so we just had to adapt the numbers of the DMX channels to our situation. The first LED had the DMX channel "33" as its first DMX channel and we had to translate it as "32" in our code since the C++ lists indexing starts at 0 instead of 1. The second LED was at the address "41" translated as "40" in the code, the third one was at "49" trnaslated as "48" and the fourth and last one was at "57" translated as "56".

Here are two examples of C++ code, the first function makes the first Renkforce LED make a red light at full intensity and the second function makes the first Eurolite LED make a purple light at full intensity :

```
void red_renkforce(){
    dmx_data[32] = 0;
    dmx_data[35] = 255; // brightness
    dmx_data[36] = 255; // red
    dmx_data[37] = 0; // green
    dmx_data[38] = 0; // blue
}
```

```
void purple_eurolite(){
    dmx_data[64] = 255; // red
    dmx_data[65] = 0; // green
    dmx_data[66] = 255; // blue
    dmx_data[67] = 255; // brightness
}
```

The DMX (Digital MultiPlexing) technology can control other types of lighing and stage equipment such as smoke machines or stroboscopes. The norm imposes the use of 5-pin DMX ports and cables but the 3-pin DMX is more widely used, despite its prohibition. The DMX equipments (spots, smoke machines, stroboscopes...) can be

daisy-chained up to 512 channels in one "universe". For example, our Renkforce LEDs use 7 channels, so it means we could daisy-chain up to 73 Renkforce LEDs in one universe. The number of universes a controller can control depends on the controller itself.

2.3. Python coding

The last step to finish this project was to build the user interface that could send a signal to be read by the micro-controller and interpreted by an independent MBed configuration. The group decided to use Python, mainly PyQT for the UI itself and PySerial to send the message to the micro-controller.

2.3.1 User Interface with PyQT

The interface was built with the aid of QTdesigner, which gives a base for the visual interface programmed in PyQT. The visual interface is based on an RGB controller, as can be seen on Figure 4. This interface allows the user to choose an RGB value using sliders. Besides that, the user can also choose to attribute this value to either Key 1 or Key 2, which represents two keys of the keyboard or the AKAI.

This interface was built as a demonstration of 3 aspects of the project : the interface can control the color of the LED lamps, that color will only appear when an specific key of the MIDI controllers is pressed and the user can choose which key receives a color. Of course, our interface was limited to controlling the colors activated by two keys. Nevertheless, it is a matter of slightly adapting the code and the interface to allow the user to control all the keys.

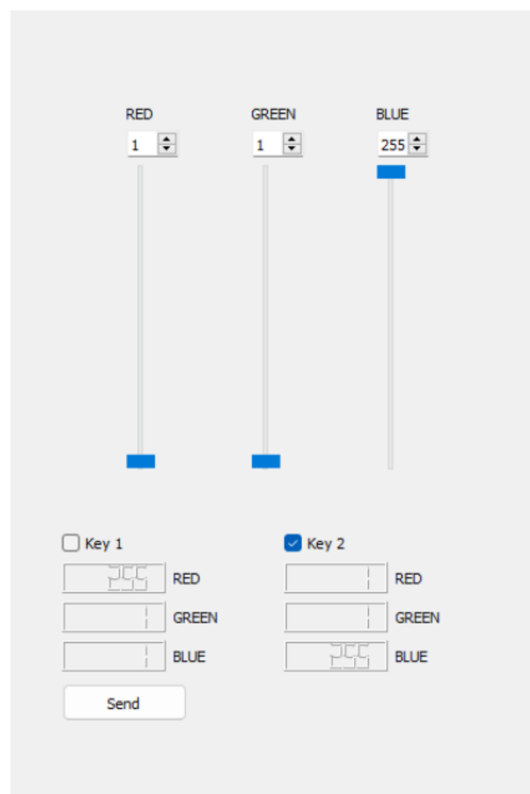

Figure 4 : *Graphic Interface*

Checkboxes

The main function of the interface that was not possible to construct with only the aid of QTdesigner, was the one that connected the checkboxes for each key with the LCD displays. This function can be seen bellow :

```
def btnstate(self,check_KEY1):
```

```
    '''
    This function connects the sliders to the LCD display only if the
    check box is checked.
    If the box is unchecked, it disconnects the LCD from the slider.
    ----------
    check_KEY1 : Check box
        Can be checked or not by the user on the interface.

    Returns
    -------
    None.

    '''
    if check_KEY1.isChecked() == True:
        self.R.valueChanged['int'].connect(self.REDLCD.display)
        self.R_2.valueChanged['int'].connect(self.GREENLCD.display)
        self.R_3.valueChanged['int'].connect(self.BLUELCD.display)
    else:
        self.R.valueChanged['int'].disconnect(self.REDLCD.display)
        self.R_2.valueChanged['int'].disconnect(self.GREENLCD.display)
        self.R_3.valueChanged['int'].disconnect(self.BLUELCD.display)
```

Listing 2 : *Checkbox control function*

2.3.2 Sending the message with PySerial

In this part we describe how we used the PySerial library from Python to establish the communication between the UI and the NUCLEO card, and also how we encoded and sent the information to the microprocessor. Firstly, we establish the connection with the microprocessor right at the start of the graphical interface, as demonstrated in the Listing 3.

```
class Ui_MainWindow(object):
    def __init__(self):
        # To open the serial communication at a specific baudrate
        self.Ui_MainWindow_serNuc = Serial(
            serial.tools.list_ports.comports()[0].name, 115200
        )
```

Listing 3 : *Python function to establish connection between the UI and the NUCLEO card.*

The second step concerns the information encoding. To do so, we compiled the color we wanted to send in a sequence of 8 characters : the first character is the symbol "$", which characterizes the beginning of the color code; the second character means the key of the keyboard that is selected, it can be the values 1 or 2 (we made a demonstration with our UI controlling just two keys, or two groups of keys, of the keyboard because we did not have time enough to create a more complex UI); the last six characters are the HTML color code. The HTML format is basically "`RRGGBB`" where "`RR`" is the hexadecimal value between 0 and 255 representing the red color value as well as "`GG`" is for green and "`BB`" is for blue. Finally, since the NUCLEO does not read strings, we converted these 8 characters' strings into a sequence of bytes with "ASCII" encoding before sending the message. The port communication is closed at the same time as the software. The whole Python code used to send the color to the NUCLEO can be seen in the Listing 4.

```
def sendRGB(self, Pushbutton):
    '''
    This function sends the information to the microcontroller with
    PySerial when the Send button is pressed.

    Parameters
    ----------
        None.

    Returns
    -------
```

```python
        None.
    '''

    if self.check_KEY1.isChecked() == True:
        key = '1'
        # To convert the RGB values into HEX '&RRGGBB' format
        # Integer values
        REDvalue = self.REDLCD.intValue()
        GREENvalue = self.GREENLCD.intValue()
        BLUEvalue = self.BLUELCD.intValue()

    elif self.check_KEY1_2.isChecked() == True:
        key = '2'
        # To convert the RGB values into HEX '&RRGGBB' format
        # Integer values
        REDvalue = self.REDLCD_2.intValue()
        GREENvalue = self.GREENLCD_2.intValue()
        BLUEvalue = self.BLUELCD_2.intValue()

    # Hexadecimal values
    REDhex = "{:02x}".format(REDvalue)
    GREENhex = "{:02x}".format(GREENvalue)
    BLUEhex = "{:02x}".format(BLUEvalue)

    color_code = '&' + key + REDhex + GREENhex + BLUEhex

    # Sending the information '&KRRGGBB'
    self.Ui_MainWindow_serNuc.write(bytes(color_code, 'ascii'))
```

Listing 4 : *Python function to send information from the UI to the NUCLEO card.*

2.4. Independent Mbed program

In this section, we will explain the complete Mbed code that is utilized to read the encoded information coming from both the UI and the keyboard, and then send it to the lamps in DMX format. First, let's delve into the two functions that allowed us to perform this task, begging with *ISR_debug_pc_reception*(). This function enables us to read the sequence of characters sent by the User Interface and to store it in global variables. It was based on an example code from Julien VILLEMEJANE and substantially modified to address our needs, being presented in Listing 5.

```cpp
void ISR_debug_pc_reception(void){
    /*
     * Function to read the data coming from the Python interface:
     */

    data = debug_pc.getc();         // get the received byte
    if(data == '&'){
        /* if this symbol is read, the next 7 bytes will correspond to 1 char and 3 hex numbers
         * the char indicates which note we want to address, while the 3 numbers encode,
         * respectively, the red, green and blue coordinates of the color we want to set
         */

        // get the byte associated to the selected note:
        note = debug_pc.getc();

        // if this byte is the char '1', store the RGB coordinates in the global variables
        // corresponding to the note (or set of notes) 1:
        if(note=='1'){
            // get the two bytes associated to the color red:
            color1 = debug_pc.getc();       // read each one as a char
            color2 = debug_pc.getc();

            std::string st1(1,color1);      // transform each one in a string
```

```
            std::string st2(1,color2);

            // concatenate the two strings, forming a 2-digit hex number in string format
            std::string st = st1 + st2;

            // transform the hex string into an integer
            r1 = std::stoi (st,nullptr,16);

            // get the two bytes associated to the color green:
            color1 = debug_pc.getc();
            color2 = debug_pc.getc();

            std::string st3(1,color1);
            std::string st4(1,color2);
            std::string st_ = st3 + st4;

            g1 = std::stoi (st_,nullptr,16);

            // get the two bytes associated to the color blue:
            color1 = debug_pc.getc();
            color2 = debug_pc.getc();

            std::string st5(1,color1);
            std::string st6(1,color2);
            std::string st__ = st5 + st6;

            b1 = std::stoi (st__,nullptr,16);
        }

        // Analogous procedure if the char '2' is read:
        else if(note=='2'){
            ...
        }
    }
}
```

Listing 5 : *Auxiliar function to read and store the data coming from the Python User Interface*

The idea of this function is to read one byte at a time of the char sequence. Each char is read through the command $.getc()$ applied to the Serial port *debug_pc*. In the original code of Mr. Villemejane, the port was declared as $UnbufferedSerial$, which allowed for the application of the commands $.read()$ and $.write()$ to *debug_pc* in order to receive and send back commands to the PySerial interface. Nonetheless, our MIDI/DMX Mbed code utilized a declaration of the port as simply *Serial*, and this unable the use of the two mentioned commands (because of how these declarations were defined in the $mbed - os$ documentation of each project), which needed to be substituted by $.getc()$ and $.printf()$.

That being said, the first char that the function receives indicates which type of command is being sent by the UI. In our case, we only utilized the symbol &, which was associated with a command to define the colors linked to each key. Nonetheless, given more time we would send other types of commands (and thus employ other symbols), such as informing the channels of the LEDS, and choosing which of them we wanted to use.

The next char assigns the keyboard note (or set of notes) that will be programmed - in our case it could only assume the values $'1'$ or $'2'$, but the idea would be to have 12 options to fully address the notes of each octave - defining which set of global variables we would modify subsequently. Then, the next 6 chars represented 3 hexadecimal numbers corresponding to the red, green, and blue coordinates, respectively, of the color to be set in the LEDs. After reading these chars and converting them into integers, the values are stored in global variables that can then be used to send the desired DMX signal to the LEDs when the key/keys of interest are pressed.

The second function that we used is $setcolor()$, presented in Listing 6. It is a very simple function that aims to simplify and quicken the task of sending the DMX signal to the Renkforce LEDs after both the UI and MIDI data are gathered (for the MIDI, the data consists of which note is pressed and with what intensity). The function

takes 4 integers as arguments, 3 concerning the RGB coordinates of the color that will be set in the LEDs and 1 addressing the intensity of the displayed light.

```
void setcolor(int r, int g, int b, int v){
    /*
     * Function to turn on all lamps with a color of RGB
     * coordinates (r,g,b) and intensity v:
     */

    // Lamp 1:
    dmx_data[32] = 0;
    dmx_data[35] = v; // Total Int.
    dmx_data[36] = r; // Red
    dmx_data[37] = g; // Green
    dmx_data[38] = b; // Blue

    // Lamp 2:
    dmx_data[40] = 0;
    ...
    dmx_data[46] = b;

    // Lamp 3:
    dmx_data[48] = 0;
    ...
    dmx_data[54] = b;

    // Lamp 4:
    dmx_data[56] = 0;
    ...
    dmx_data[62] = b;
}
```

Listing 6 : *Auxiliar function to define the lamps' displayed colors.*

Until the stage we reached here, this function simply sent the DMX signal (corresponding to the input arguments) to all 4 lamps of the Renkforce LED set. Nonetheless, given more time, the idea was to set more input arguments to the function in order to define which lamps would be lightened up, as well as the addresses of their first channels. This addition, coupled with the proper additions in the PyQT/PySerial code, would then allow users to have a big degree of freedom in the customization of the setup.

To finish, we can comment on the main code of this part to explain how the aforementioned functions were applied. The code is shown in Listing 7, and starts by setting the appropriate baud rate of the Nucleo Serial port (initialized as *debug_pc*), i.e., the rate at which information is transferred through that communication channel. Subsequently, the function $ISR_debug_pc_reception()$ is called through the command *.attach* in *debug_pc*, and then we initialize both the DMX and MIDI links. A first loop is called in order to maintain the Nucleo continuously capable of receiving Serial commands from the UI, and then another loop is implemented just for the reception of MIDI data.

```
// Main
int main() {
    debug_pc.baud(115200);
    debug_pc.attach(&ISR_debug_pc_reception, Serial::RxIrq);
    initDMX();
    initMIDI();
    while(1){
        while(1){
            if(isNoteMIDIdetected()){
                playNoteMIDI2(note_data, velocity_data);
                if(note_data == 0x2f || ... || note_data == 0x40){
                    setcolor(r1, g1, b1, velocity_data);
                }
                else{
                    setcolor(r2, g2, b2, velocity_data);
```

```
16                  }
17                  resetNoteMIDI();
18              }
19          updateDMX();
20          wait_us(1000);
21          }
22      }
23  }
```

Listing 7 : *Mbed main function for reading MIDI and UI inputs and sendind the proper DMX output to the LEDs.*

This construction of two integrated infinite loops was necessary because, after the UI commands are read and the configurations of the system are set, the second loop is what allows the Nucleo to keep receiving MIDI commands from the keyboard, even if the UI does not send any other commands. Furthermore, the Nucleo is still able of receiving commands from the UI at any moment, meaning that the user can change the setup configurations (such as the link between keys and colors) at any given time, and they will immediately result in an update of the system.

The function $isNoteMIDIdetected()$ enables us to know whether a note is played or not (returning respectively the booleans *True* or *False*), and in the affirmative case, the function $playnoteMIDI2(note_data, velocity_data)$ is employed to access the information of that note. This information becomes stored in the variables *note_data*, that informs which note is played, and *velocity_data*, which informs the velocity/force that the note is played.

After this, a simple if-elif-else conditional, along with our function *setcolor*, can be implemented to define which color will be displayed. For example, in our code we associated half of the keyboard notes to the encoding $'1'$ of the UI command, and the other half to $'2'$. In that way, we were able to independently control (through the UI) the LED colors displayed for each half of the keyboard. Before any user commands, the colors associated with each key will depend on the initialization values granted to each global color variable (r1, r2, ...). In our case, we initialized all of them as zeros, so that pressing any key would not result in any response of the LEDs (this is justified because the RGB values sent as DMX also control the intensity of the light - the specific intensity argument informed by *velocity_data* controls only the total intensity).

To finish, as the name proposes, the function $resetNoteMIDI()$ resets the read note, allowing for a new reading, and then $updateDMX()$ sends all the DMX data (that we came to define) to the lamps. Also, a minimal time between each iteration can be set through $wait_us()$, whose input is given in microseconds.

3. Task distribution and Organization

In order to achieve the initial goals of the project, the distribution of tasks, the definition of smaller missions to be validated, as well as the general organization of the group were fundamental. In the beginning, we created both a Trello and a Google Drive to properly organize all information that could be useful for the project. This included not only various documentation of the available equipment, example programs, and links from LEnsE, but also our own codes, organization documents, and internet research. Having all this information organized and easily accessible was essential to keep all group members updated and integrated around the project.

Nonetheless, although we had these organization spaces, in the first two sections we did not give enough focus to building a solid organizational base and fully understanding the project. On the contrary, we tried to attack the project directly, quickly dividing the immediate challenges that we had at hand among the group members. As a result, we ended up being unconscious of the project as a whole and got stuck in some simple tasks that demanded a better degree of organization (since at that phase, everything was dependent on finding the right addresses of the equipment and understanding how they worked).

As a consequence, the communication inside the group got worse and we had big difficulty getting back on track with the project at the beginning of the sessions. After this happened for the second time, we finally recognized our mistake and started working to solve it. This was done not only to improve our chances of success on a technical level, but also to make the sessions more pleasant and fruitful in a general sense. That being said, the first thing we did was gather the whole group to clarify the goals and challenges of the project as a whole,

	06/02	27/02	06/03	13/03	20/03	27/03
Programming with mbed directly to send the information to the NUCLEO	Edit the code to make the LEDs work					
	Display MIDI notes played on the keyboard					
	Interpret the MIDI notes and how the microcontroller reads it					
	Do a schema of the project					
		Understand Hexadecimal				
		Learn how to transform each MIDI input into one color.				
		Programme the Scrabin mode for the keyboard (each key is a color)				
		Set up pattern for the lights, undestanding how to control them				
		Understand the output the AKAI (its hexadecimal, but different from the keyboard)				
		Read MIDI signal from AKAI and print on the computer				
			Write a program to respond to the AKAI PAD's			
			Do more complex programs with the different possible buttons and outputs of the AKAI			

Figure 5 : *First part of our Gantt diagram with the tasks concerning the codes in Mbed.*

which was followed by a restructuration of our plannings and a directed effort into documenting and organizing the project.

In this context, one thing that was essential to keep the group motivated was dividing the project into smaller steps and objectives that we could validate at each session. In that way, we could track our progress, adapt our weekly goals, and thus stay centered even in the sessions where nothing was working as predicted. This segmentation was performed through a Gantt Chart, which is basically a bar chart that allows for the illustration of a project schedule.

As we advanced with the project, discovering more about its potential and limitations, there was a constant need to keep the diagram updated, which was done every week. That being said, the final version of our Gantt Chart can be seen in Figures 5 and 6. The grey boxes represented the objectives that have been finished, while the colored boxes corresponded to the ones that were still open. We also divided the schedule according to three main categories, with a specific one concerning the organization.

Another organization tool that we implemented was using a set of slides to clearly distribute the weekly tasks of each member. At every beginning of a session, we would recapitulate what had been done in the previous week and then define the missions that each member would attack in the given afternoon. An illustration of these slides is shown in Figure 7.

	06/02	27/02	06/03	13/03	20/03	27/03
Graphic Interface			Understand how to create a graphic interface with Python to automatize the process of controlling the Mbed and sending the program to the microcontroller			
			Start learning about graphic interface (we were doing it wrong!)			
			Understand how to connect the C++ and Pyhton interface!			
				Plan how our interface is going to work according to what's possible		
				Read a PySerial input with the microcontroller		
				Start the part "design" of the interface		
				Change the code for it to accept a PySerial input that affects the resulting LED programming		
					Connect the Graphic Interface, the Pyserial commands and the Mbed program	
					Be able to select the color of all LEDs using PySerial commands (**without the MIDI link**)	
					Be able to select the color of all LEDs using the Graphic Interface (**without the MIDI link**)	
					Be able to select the color of all LEDs using PySerial commands (**with the MIDI link**)	
					Be able to select the color of all LEDs using the Graphic Interface (**with the MIDI link**)	
						Define the color associated with each key **individually** using PySerial commands
						Define the color associated with each key **individually** using the Graphic Interface
Organization			Reorganize the project. We now understand what to do and have to reorganize our tasks.			
			Prepare the documentation for the Audit of next week. Demonstration, comments on the code and powerpoint presentation.			
					Refine the documentation for the Audit. Final	

Figure 6 : *Second and last part of our Gantt diagram with the tasks concerning the UI development and our organization.*

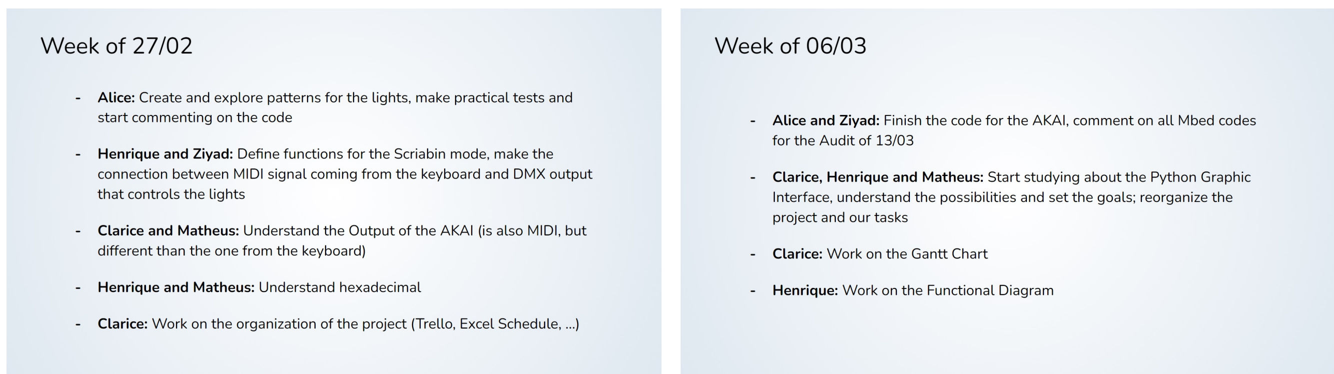

Figure 7 : *Example of the Weekly Task Distribution Slides*

4. Conclusion

After 6 sessions of work to appropriate the MIDI/DMX tools, we were able to control the provided lights using a keyboard and an AKAI pad, following what was proposed in the project objectives. Furthermore, we managed to create a User Interface using Python, as well as to integrate it to an Mbed code, allowing the user to directly customize the procedures of conversion between MIDI input and DMX output in the Nucleo card (as an example, to define the LED color associated with a certain group of keys) it with the Mbed code o have a proper communication between the instruments and the lights. We were not able to control the lights according to all the pattern modes we wanted, but it was a matter of time to solve the problem and to make the UI too capable of doing so.

Overall, we were in good working conditions in the team, especially after we realized our initial mistake of putting organization on a second plan. That being said, better communication and more precise task division at the beginning could have definitely saved us some time and energy. We also managed to tackle the technical obstacles without great difficulty even if, for most of us, it was the first time manipulating these types of electronic systems. In fact, we believe that we could have achieved the lacking work if we had one or two additional sessions.