**“Light organ with musical events microcontrolled synchronization”**


**BACHELOR'S DEGREE IN COMPUTER ENGINEERING**

**Bachelor's Thesis**

# *"Light organ with musical events microcontrolled synchronization"*

ACADEMIC COURSE: 2023/2024

Juan Andrés Peña Maldonado


BACHELOR IN COMPUTER SCIENCE

# *"Light organ with musical events microcontrolled synchronization"*

AUTHOR:

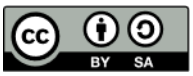
**Juan Andrés Peña Maldonado**

SUPERVISED BY:

**Prof. Andrés Roldán Aranda**

DEPARTMENT:

**Electronics and Computer Technologies**



# "Light organ with musical events microcontrolled synchronization"

**Juan Andrés Peña Maldonado**

**KEYWORDS:** *GUI, MIDI, Arduino, Qt, music, organ*

**ABSTRACT:**

This Bachelor's Thesis main objective is to develop an application capable of reproducing a song on an Arduino board while assinging each note a light on the board which will be turn on and off accordingly starting from a MIDI file. It will also have a Python interface which will allow the user to interact with it. The programm will be able to:

- Read and get all the necessary information from a MIDI file (Notes, tracks, instruments, etc...)
- Reproduce MP3 files and manage the organs lights
- Compile and execute Arduino code necessary from the interface

This Bachelor's thesis is part of one of the TFGs offered by the Aerospace Electronics group, GranaSAT.

# "Órgano de luces con sincronización microcontrolada de eventos musicales"

**Juan Andrés Peña Maldonado**

**PALABRAS CLAVE:** *GUI, MIDI, Arduino, Qt, música, órgano*

**RESUMEN:**

El objetivo de este Trabajo de Fin de Grado es desarrollar una aplicación que pueda partiendo de una canción en formato MIDI, reproducir dicha canción en una placa Arduino asignandole a cada nota una luz de la placa que se ilumine de forma correspondiente. Contara también con una interfaz en Python que permitira interactuar con el programa. El programa podrá:

- Analizar un archivo MIDI y extraer la información necesario de este (Notas, pistas, instrumentos, etc...)
- Reproducir archivos MP3 y controlar las luces del órgano
- Compilar y ejecutar el código Arduino necesario para la placa desde la propia interfaz

Este Trabajo de fin de Grado forma parte de uno de los TFGs ofertados por el grupo de Electrónica Aeroespacial,GranaSAT.

*"But this didn't feel like magic. It felt a lot older than that. It felt like music."*

**Terry Pratchett**
***Soul Music*, 1994.**

## *Acknowledgements*

To my family for always being there for everything I have ever needed and to my friends and colleges for putting up with me. Thank you everyone.

# *Agradecimientos*

A mi familia por estar siempre para todo lo que me ha hecho falta y a mis amigos y compañeros por aguantarme. Muchas gracias a todos.

# Contents

# Glossary

**A** | **G** | **M** | **P**

**A**

**Arduino** *Electronics Platform.* WIP .

**G**

**GranaSAT** *Electronics Aerospace Group.* An academic project from the UGR. This organization has an electronics laboratory where students from different degrees and education levels develop multidisciplinary projects [9] .

**M**

**MIDO** *Python Library.* The library we will use for MIDI file manipulation. We are using version 1.2.10. .

**P**

**PyQt** *Python binding for Qt.* It allows us to use tu use Python code with Qt. We are using version 5.9.2 of PyQt and version 5.9.7 of Qt .

**Python** *Programming Language.* It will be the main programming language we use for this project. We are using version 3.9.7 of Python. .

# Acronyms

**A** | **G** | **M** | **U**

**A**

**API** Aplication Programming Interface.

**G**

**GUI** Graphical User Interface.

**M**

**MIDI** Musical Instrument Digital Interface.

**U**

**UGR** University of Granada.

# Chapter 1

# Introduction

This Bachelor Thesis shows the result of knowledge and skills acquired by the student in the Bachelor's Degree in Computer Engineering which has been tested during the development process of this project.

This document is meant to reflect the development process of this application. It will include MIDI file reading and processing, Python GUIs development, Arduino board coding and compiling among other different processes, all of which will be necessary in order to create a fully functioning light organ GUI.

This Final Degree Project is carried out in collaboration with the academic project GranaSAT. This is an aerospace development group of the University of Granada (UGR), formed only by students from different fields of Engineering, such as Aerospace Engineering, Electronic Engineering, Computer Engineering or Telecommunications Engineering among others, under the supervision of Professor Dr. Andrés María Roldán Aranda.


**Figure 1.1** – *The GranaSAT logo.*

## 1.1 Motivation

When I was scrolling through the open TFG topics list I did not have anything in particular in my mind, but when I saw this one, I immediately knew this was exactly what I wanted.

It sounded like a fun topic, something that would be very visually appealing once it was finished and it allowed me to work with Arduino boards which was something I had already enjoyed in the past although it seemed like a challenge specially compared to the instances I had used Arduino before.

## 1.2 Project goals and objectives

1

When I started this project, my main objectives for its development were:

| Obj. Nº | Description |
|---|---|
| Obj. 1 | Create an Arduino application capable of reproducing a song while simultaneously turning on and off 32 lights assigned to each tone of the music. |
| Obj. 2 | Develop code that is able to process a MIDI file to get a lights sheet that tells the program which lights to turn on and off and when. |
| Obj. 3 | Coding an GUI that will help the user interact with the organ being able to add and delete songs as necessary. |
| Obj. 4 | Being able to carry on a project of this size and document it in this document so that future improvements can be carried out by other team. |

**Table 1.1** – *Top-level objectives of this Bachelor Thesis.*

## 1.3 Project structure

This document will be divided into five chapters, each of one will detail the different parts of the project.

Those chapters are:

1. **Chapter 1: Introduction.** In this chapter we will establish the structure of the TFG and a brief summary of what it consists of. It will also set up the layout of subsequent chapters
2. **Chapter 2: Background Knowledge.** This section will be dedicated to introducing all the different concepts and ideas that will be necessary for the project and the understanding of its development.
3. **Chapter 3: Project Planning.** Chapter 3 explains in detail the planification of the different stages of the project. It will contain a Gantt Diagram with the expected duration of each stage of the project and a summary of how well I adjusted to that timetable and the difference that ended up happening.
4. **Chapter 4: System description and design.** This chapter will be give an overview of the whole system, its different parts and their design and development.
5. **Chapter 5: System testing.**

   Contains different examples of how the final application works and how to navigate it.
6. **Chapter 6: Conclusion and future improvements.** Finally, the last chapter is dedicated to the conclusions and lessons learned throughout the project as well as different improvements that could be added

The addenda will contain different information such as tutorials on how to get the application to work on different computers as well as anything that didn't have a home in another chapter but was relevant to this document nonetheless.

# Chapter 2

# Project Analysis

Now that we have established the structure this document will take, we will start analysing this project, how will we approach it and the tasks that will be required.

## 2.1 Task Analysis

Here is a diagram of how we will want our system to operate

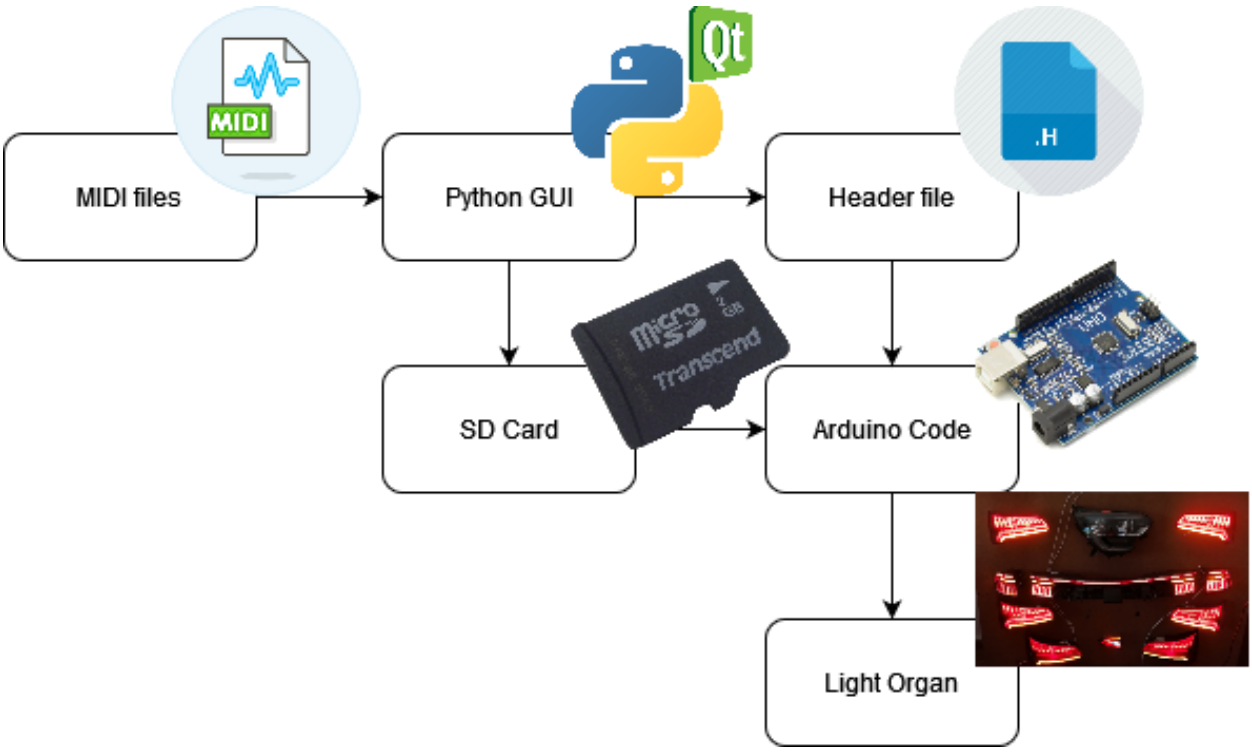



**Figure 2.1** – *System Diagram*

As we can see, our Python code will be in charge of extracting the information from the MIDI files

and converting it into both notes for the header file and an MP3 file for the SD Card. Implementing this functionality will be our first task.

After that, the Arduino board will be in charge of both reading the lights information from the header file and the MP3 file from the SD and reproduce both. Designing the code for the board so that it can do this will be our second task.

Once all the functionalities are implemented we will be designing the GUI, so that the program can be interacted with. This will be the third task

Finally, after everything is done, we will need to test everything and round up the documentation. That will be the final tasks.



## 2.2 Time Analysis

For us to visualize this information more effectively, we will use a Gantt Diagram in which we will layout the different tasks that will be required in a timetable, assigning to each of them a time period in which that task should be finished and we will also establish the time dependencies among them.

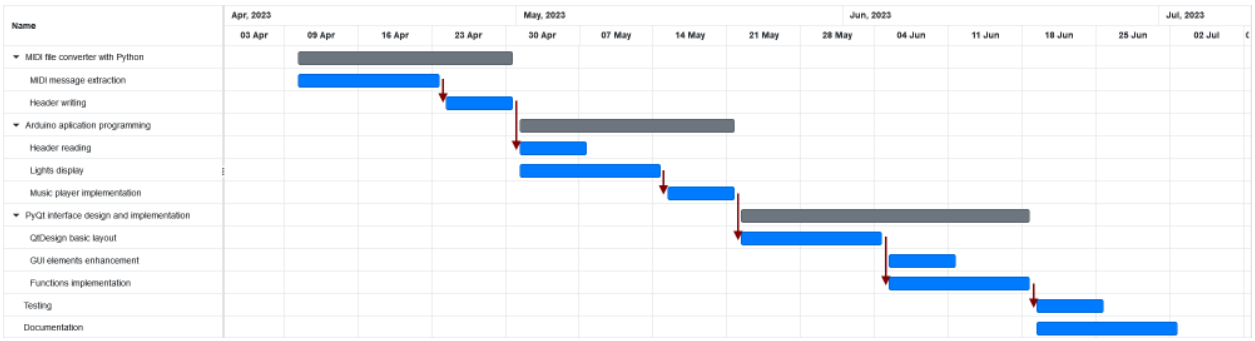



**Figure 2.2** – *Gantt Diagram*

# Chapter 3

# Background Knowledge

In this chapter, we will collect all the information necessary for this project.

## 3.1 The MIDI format[1]

**Figure 3.1** – *MIDI logo*

MIDI is a protocol which allows digital instruments to communicate with each other. A MIDI file is just a record of this conversation which allows us to play it back by reproducing each of the different messages recorded.

Because of this, by reading a MIDI file you can get information about each note that is being played, for how long is it played, at what time, etc... that would be absent from other audio files which only store sound waves.

### 3.1.1 MIDI messages[2]

Each MIDI message is composed of two types bytes: A status byte and data bytes.

A status byte informs us about the type of message and the channel used, it always comes first and its first bit is always "1".

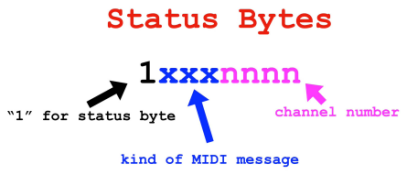



**Figure 3.2** – *Structure of a status byte*

Data bytes come after the status byte, they contain contextual information depending on the type of message described by the status byte. There can be multiple in a row and their first bit is always "0"



Channels are a way to separate messages from each other. A device can only read messages from its assigned channels, which allows us to set what instrument must play what note. Because we only have 4 bits to represent a channel there are only 16 possible channels

There are 8 types of MIDI message which will be covered in this list:

| Name | Status byte | Description |
|---|---|---|
| Note off | 1000 | Contains information about which note is released and at what velocity |
| Note on | 1001 | Contains information about which note is pressed and at what velocity |
| Polyphonic Aftertouch | 1010 | Contains information about the pressure on a held key. I more precise than Channel Aftertouch |
| Control Change | 1011 | It modifies the values of knobs, sliders and pedals. |
| Program Change | 1100 | This message is not very relevant nowadays and its used to change from one patch[1] to another |
| Channel Aftertouch | 1101 | It contains information on the pressure used on the key with the highest pressure. It requires less space than Polyphonic Aftertouch |
| Pitch Bend Change | 1110 | It controls the pitch slider. It has more resolution than the Control Change message. |
| System Messages | 1111 | They don't need to have a specified channel. They are used to send information across channels, to synchronize different clock-based MIDI components and to send manufacturer relevant information[10] |

**Table 3.1** – *List of MIDI messages*

### 3.1.2 Working with MIDIs in Python: MIDO[3]

In order to work with MIDI messages we will use the Python library MIDO.

[1]Synthesizers used to have a list of 128 presets or patches which allowed them to reproduce a particular type of instrument. These encompassed things like Violin, Recorder or Organ, but because they were also used for video games, the list contained weird patches like Gunshot or Bird

### 3.2.1 Arduino UNO

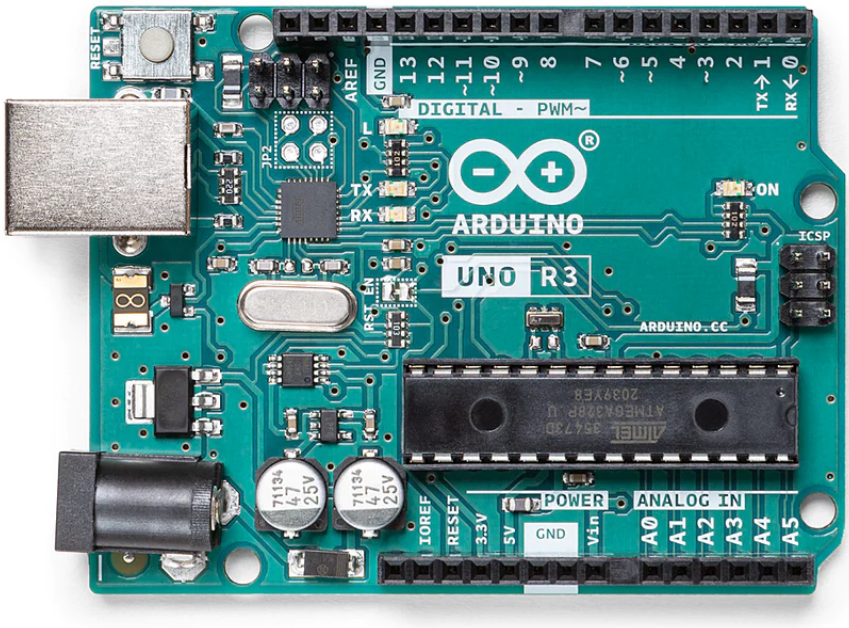



**Figure 3.7** – *Arduino UNO board*

There are lots of Arduino boards models to choose from. We won't need a lot of power for what we are aiming for, so we can choose one of the lower-end Arduino boards. Because we will need a lot of different lights to be connected, we could choose an Arduino Mega, but we will instead settle for the Arduino Uno and use libraries to circumvent this. We will do this because as we said before we already own one and because it would be cheaper if we didn't but you can realistically justify using an Arduino Mega.

As we said previously, we will reuse a board from a previous year[11], which is soldered to a PCB board with LEDs and other sockets. The board is also equipped with a clock, a speaker, an LED display and an external SD card.

### 3.2.2 Arduino Programming

Arduino utilizes its own language derived from C++. It uses most of the C++ only with a few changes. The main one is that it utilizes void setup() and void loop(). void setup() is a function executed when the board is booted, we could call it the startup function and on the other hand, void loop() is called repeatedly whenever it finishes, its the main function which we want the board to perform.

### 3.2.3 Arduino Libraries

In order to implement all the required functionalities for the board, we will use a collection of different libraries that will help us manage everything

After that, we can use it as a normal serial port, using the begin, print, available and read functions.

#### 3.2.3.3 DFRobotDFPlayerMini

This library allows us to use the DFPlayer, which we will use to reproduce the MP3 files we need from the board.

In order for us to do this, we first need to create a serial port with the AltSoftSerial library and then we will create a player by creating a player object and then assining that port to the player object.



```
// Creation of the DFPlayer object
DFRobotDFPlayerMini player;

void setup()
{
  //Audio player initialization
  Serial.begin(9600);
  softwareSerial.begin(9600);
  if (player.begin(softwareSerial)) {
    Serial.println(F("OK dfplayer"));
    player.volume(vol); // Set volume
  }
  else {
    Serial.println(F("Connecting to DFPlayer Mini failed!"));
  }
}
```

**Figure 3.10** – *DFRobotDFPlayerMini player initialization*

This will allow us to play the songs stored in the SD card by calling the player.play(number_of_song) function.

#### 3.2.3.4 TinyIRReceiver

Finally, TinyIRReceiver is a library that will help us control the board with an infrared remote. This will be handled via interruption events which will allow the rest of the code to be played at the same time. These interruptions are handled entirely by the library which calls the function handleRecievedTinyIRData() which must be declared in the code.

```
// interrupt for IR and getting the hexcode for the commands
#if defined(ESP8266) || defined(ESP32)
  void IRAM_ATTR handleReceivedTinyIRData(uint16_t aAddress, uint8_t aCommand, bool isRepeat)
#else
  void handleReceivedTinyIRData(uint16_t aAddress, uint8_t aCommand, bool isRepeat)
#endif
{
  #if defined(ARDUINO_ARCH_MBED) || defined(ESP32)
    // Copy data for main loop, this is the recommended way for handling a callback :-)
    sCallbackData.Address = aAddress;
    sCallbackData.Command = aCommand;
    sCallbackData.isRepeat = isRepeat;
    sCallbackData.justWritten = true;
  #else

    // various actions for different buttons
    switch(aCommand)
    {
      case ONE  :
      case TWO  :
      case THREE:
      case FOUR :
      case FIVE :
      case SIX  :
      case SEVEN:
      case EIGHT:
      case NINE :
      case UP   :
      case DOWN :
      case LEFT :
      case RIGHT:
      case OKAY :
      case ZERO :
      case STAR :
      case HASH :
      default:
      Serial.println(" other button : ");
      Serial.println(aCommand, HEX);
    }// End Case

  #endif
}
```

**Figure 3.11** – *IRReceiver interrupts handling code*

It is important to note that we must make two possible declarations for handleRecievedTinyIRData(), one for when ESP8266 or ESP322 are defined and one where they are not. This is because if we are using an ESP8266 or an ESP32 microcontroller, we want the code for an interruption to run on RAM instead of flash memory so that it ends as fast as possible which is indicated by the IRAM_ATTR marker at the start of the function declaration.

Once that is out of the way, the code is really simple. When the function is called, it will take the data from aCommand which contains the information on the button pressed on the remote and will execute a line of code depending on its content.

### 3.2.4 Arduino-cli[5]

Arduino-cli is an Arduino command line tool. It allows board management, library management, sketch builder, board detection and more. This will be crucial, since we will need to compile the Arduino code we have written directly from the GUI.

#### 3.2.4.1 PyDuino-cli[6]

PyDuino is a Python library that will allow us to use the Arduino-cli functionalities from Python code.

## 3.3 FluidSynth[7]

**Figure 3.12** – *FluidSynth logo*



Our Arduino board is only able to play MP3 files, so we will need FluidSynth to convert the MIDI files into MP3. Fluidsynth is a real-time software synthesizer based on the SoundFont2 specifications, and although it lacks a GUI, it has a very powerful API.

From FluidSynth we will only use the command: *fluidsynth -ni soundfont.sf2 MIDI.mid -F MP3.mp3 -r sample_rate* which will convert a MIDI file into an MP3 file, just like we wanted.

## 3.4 PyQt Interfaces[8]


**Figure 3.13** – *PyQt logo*

PyQt is one of the most popular Python bindings for the Qt cross-platform C++ framework. It will allow us to create Qt GUIs making use of the multiple tools that come with it like Qt Designer while using Python for the programming part.

### 3.4.1 QtDesigner

QtDesigner is a program utilized to as its name implies design Qt GUI in a graphical way without having to code it directly. We will be able to add different elements from its toolbox and arrange them as we see fit

to create a responsive and useful GUI. This toolbox includes buttons, menus, lists, text boxes, etc...

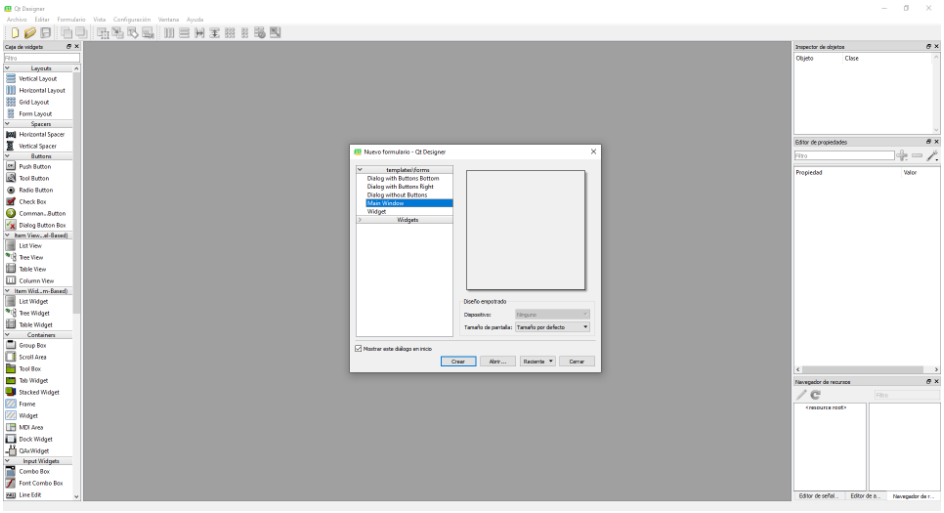


**Figure 3.14** – *QtDesigner GUI*

# Chapter 4

# System design

## 4.1 MIDI file converter with Python

The first task in our project will be to create an application that will take a MIDI file (.mid) and extract all the relevant information about it that we will need for the light organ.

We will be using Python as the programming language for this purpose because it's the one I am more acquainted with, it is very flexible which is useful when working with different types of data and it has different libraries that interact with MIDI files and as we explained in Chapter 3, we will be using the MIDO library to work with MIDI files. I have chosen this library because of its ease of use and large documentation.

### 4.1.1 MIDI message extraction

As we explained in Chapter 3, every MIDI file contains tons of different messages but we will focus only on what we will need for the organ.

Our objective will be to have a matrix in which we will store what note is being played, when it starts, its duration, and what track does it belong to.

We will need to read each note_on/off message and convert its time notation into seconds. We will also store the time where a note starts in seconds since the begining of the song, instead of ticks since last message for clarity purpose and because it will help us later when we are programming the organ.

First, we will need an auxiliary class which we will call note_obj. This class will store a notes track, its tone, its starting time and its finishing time. This will allow us to create note objects from the information we get by reading the messages.

```python
#La clase note_obj es una clase auxiliar que guarda la información necesaria de cada nota
class note_obj():

    def __init__(self,track,note,start):
        #En que pista se encuentra esta nota
        self.track = track
        #El tono de dicha nota
        self.note = note
        #Cuando empieza a sonar (En decimas de segundo)
        self.start = start
        #Cuando termina de sonar (En decimas de segundo)
        self.end = None

    def __str__(self):
        return "{} {} {} {}".format(self.track,self.note,self.start,self.end-self.start)
```

**Figure 4.1** – *Code for the note_ obj class*

We will create two lists: One will contain the notes that have been started by a *note_ on* message but have no finishing time, we will call this list *unfinished_ notes*. The second one will be *notes*, which will store notes which have a finishing time. We will also create a tempo and beat variable in which we will store the fixed tempo of the file and the current ticks_per_beat of the current track. We will update this variable each time we read a tempo change message.

Finally, we will read each message on each track. When we read a *note_ on* message, if it doesn't have velocity 0, we will create a *note_ obj* with the note, its starting time in seconds (Calculating it from the time in ticks, the beat and the tempo), and the channel it belongs to. Whenever we read a *note_ off* message or a *note_ on* with velocity 0, we will look for that note in *unfinished_ notes* and add the finishing time calculated just as the starting time and we will move that note_obj into textitnotes.

```python
#Unfinished notes es un diccionario con las notas que sabemos cuando empiezan pero no cuando acaban.
#Es un paso intermedio antes de ser guardadas en notes
unfinished_notes = {}
#Notes guarda las notas que tienen un inicio y un final definido.
notes = []

#Tempo determina el tempo de la pista y se usa para pasar de unidades de tiempo de MIDI a segundos
tempo = 0
#Track determina la pista actual que estamos leyendo
track = 0
#Leemos cada pista del archivo
for i in mid.tracks:
    #Contamos el tiempo transcurrido desde el inicio de la canción
    #Se actualiza con cada mensaje leido y se mide en decimas de segundo
    absolute_timer = 0
    #Leemos cada mensaje de la pista
    for j in i:
        #Si es un mensaje meta, comprobamos si cambia el tempo de la pista
        if (j.is_meta):
            #En caso de que lo haga, actualizamos el tempo
            if (j.type == "set_tempo"):
                tempo = j.tempo
        #En caso de que no sea un mensaje meta
        else:
            #Actualizamos el timer
            absolute_timer += mido.tick2second(j.time,mid.ticks_per_beat,tempo)*100
            #Comprobamos si el mensaje inicia una nota o la termina
            #Si es un mensaje note_on, inicia una nota y almacenamos la nota en unfinished
            if (j.type == "note_on"):
                if (j.velocity != 0):
                    unfinished_notes[j.note] = note_obj(track,j.note,absolute_timer)
            #En caso de que sea un mensaje note_off o que la velocidad del mensaje note_on sea 0,
            #termina una nota, la sacamos de unfinished y la almacenamos en note
                else:
                    new_note = unfinished_notes[j.note]
                    new_note.end = absolute_timer
                    notes.append(new_note)
                    unfinished_notes.pop(j.note)
            elif (j.type == "note_off"):
                new_note = unfinished_notes[j.note]
                new_note.end = absolute_timer
                notes.append(new_note)
                unfinished_notes.pop(j.note)
    track += 1
```

**Figure 4.2** – *Code for the reading of messages*

Now that we have everything organized in note_obj inside a list, we will convert it to a Numpy array. Numpy arrays are much easier to work with and faster to operate with. This Numpy array will have 4 columns: Note, Start, Duration and Track. And with that out of the way, we have successfully converted a MIDI file into a data structure with which we can work and operate.

```
#Ordenamos la lista de notas en función de cuando empiezan
notes.sort(key= lambda x: x.start)

#Pasamos la información almacenada en una lista de objetos a una matriz numpy
matrix = np.zeros( (len(notes),4),dtype=int )
line = 0
for i in notes:
    matrix[line,0] = int(i.note)
    matrix[line,1] = int(i.start)
    matrix[line,2] = int(i.end - i.start)
    matrix[line,3] = int(i.track)
    line = line + 1

return matrix
```

**Figure 4.3** – *Numpy array creation*

### 4.1.2 Header writing

The next part of the task will be to convert this Numpy array into a form of data our Arduino board will be able to read and we will also need to convert each tone of the song into a corresponding light. For this purpose we will be creating the file *musiclight.sh* which will be a header file separate from the main Arduino file and will store the information of all songs.

The first part of this process will be converting the tones of the different notes into lights. Because we want different instruments to be differentiated in the organ, we will convert each track individually, assigning them a set of lights. For example, if we only have one track, any of the 32 different lights could be turned on. Contrary to that, if we have 4 tracks in the song, the notes of each track must be represented by only 8 lights each. Then we will use a mapping function to convert the original tone range to the new lights range.

The mapping function in question works like this:

1. We take the original notes range and we subtract the lowest note from all elements
2. The result should be new range from 0 to the new highest note. We divide each element by that new highest number
3. Now our range goes from 0 to 1 so we multiply it by the size of the new range (Size is equal to the highest number minus the lowest number)
4. Finally, we add to each element what we want to be the new smallest element (The lowest number from the light range)

This function follows the following properties:

1. It respects the proportions of the original range (The separation between notes is kept the same)
2. It makes sure that the highest and lowest light always get used.
3. Its easy to implement

However, it also has the following downsides:

1. It doesn't force each light to be used meaning depending on the song some lights might be used for a lot of notes while others don't get used at all
2. It doesn't reflect the spread of the original song. It doesn't matter how high were the highest notes or how low were the low ones, everything is put into the same range.

Nevertheless, although this is something that I think must be discussed, we are willing to accept this trade off. In the future we may implement different more mapping functions but for now this is more than enough.

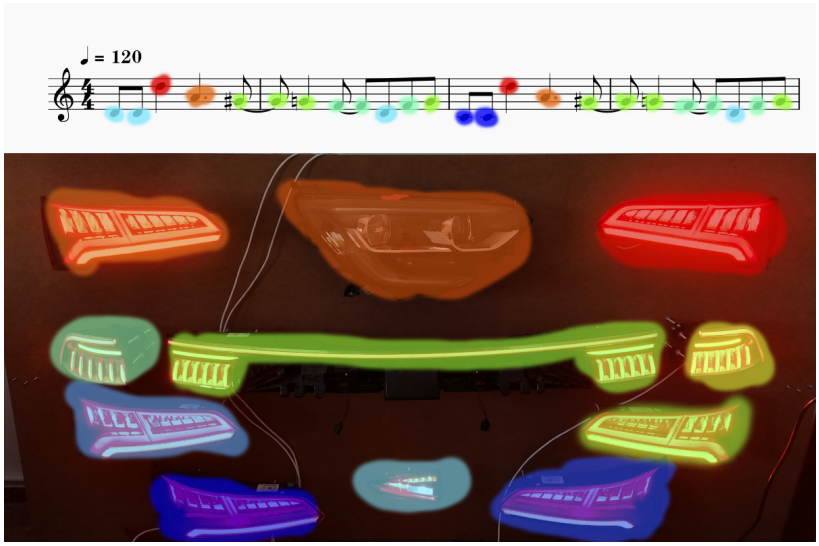



**Figure 4.4** – *Visual representation of the mapping process*

Finally, we will take all this processed information and we will add it to the *musiclights.h* file. Each song consists of:

- The tag: Its a comment before the song that indicates its number (Its main use is for file writing management)
- #define note: Its a keyword that will signal that the following variables are defined. This will be useful when we build the Arduino code
- notes: The number of notes in the song
- light: An array containing the different lights that must be turned on. They are listed in chronological order
- start: An array where each element corresponds to same numbered element of light and determines when that light has to be turned on. It's measured in tenths of second.
- delay: The final component of a song. Its another array whose elements correspond to the same numbered elements of light and determines for how long must that light be kept shining. Its also measured in tenths of second.

We will need a function that adds a new song to the file (add_song_to_header()) and a function that deletes and existing song from the file (delete_song_from_header()). We must respect the order of the songs, avoiding having a song number two and a song number four without the existence of a song number three.

- add_song_to_header: It reads the *musiclights.h* file, stores each line in a list and then looks for where the song tag for the correct place is and saves the song's data right after that

- delete_song_from_header: This function also reads the same file and stores each line on an list. When it reads each line, it looks for the song tag of the song we want to remove and then write each line after substituting each songs number for the previous one. So if we remove song number four from the header file, the next songs will become song number 4, song number 5, etc...

We must also make sure that if a file doesn't exist, it is created and populated with the necessary text that will be required (Like #ifndef statements to check that the file has not been compiled multiple times and the aforementioned song tag)

## 4.2 Arduino Application Programming

Now that our header file is ready and we can add as many songs as we want, we will start developing the code that reads this header files. This code must be able to read all the notes in the header file and turn on the lights accordingly as time passes. It will also have to allow us to select the different songs that exist in the board and will also have to reproduce the MP3 files which will be associated to each song and stored in the SD card.

First, we will import the necessary libraries and *musiclights.h*. We will need to setup some variables which are required for the different libraries we will be using like we saw in Chapter 3.



```
#include <Arduino.h>
#include <Wire.h>

// Include Shift Register Library
#include "ShiftRegister74HC595/src/ShiftRegister74HC595.h"
#include "ShiftRegister74HC595/src/ShiftRegister74HC595.cpp"

// Include Clock
#include "ArduinoRTClibrary-master/virtuabotixRTC.h"
#include "ArduinoRTClibrary-master/virtuabotixRTC.cpp"

// Include MP3 Pin Library
#include "AltSoftSerial/AltSoftSerial.h"
#include "AltSoftSerial/AltSoftSerial.cpp"

// Include MP3 Player Library
#include "DFRobotDFPlayerMini-master/DFRobotDFPlayerMini.h"
#include "DFRobotDFPlayerMini-master/DFRobotDFPlayerMini.cpp"

// Include Display Graphic Library
#include "Adafruit_GFX_Library/Adafruit_GFX.h"
#include "Adafruit_GFX_Library/Adafruit_GFX.cpp"
#include "Adafruit_SSD1306/Adafruit_SSD1306.h"
#include "Adafruit_SSD1306/Adafruit_SSD1306.cpp"



#define IR_INPUT_PIN    2 //IR receiver output
// Include for InfraRed Signal Library (Remote)
#include "IRremote-3.7.0/src/TinyIRReceiver.hpp"

#include "musiclights.h"  // Header file contating the notes and delay of the music
```

**Figure 4.5** – *Libraries import*

We will also need to declare some global variables for our program:

- starting_time: It stores the time given by the function millis() in which the current song started

- current_time: It is used later to store the current time also given by the function millis(). It is updated in each loop

- playing_song: If its -1, it means no song is being played. 0 means keep playing whatever song is currently playing and any value above 0 sets the variables to play the song with that number. For example, if playing song is 7, the required variables will be setup so the seventh song is played and then it will become 0 again until the song ends.

- current_note: When a song is playing, it stores the position of the last note played



- delay_on_lights: Its an array which helps us keep track of how long lights should stay on. Each position represents a light and stores the amount of time it must remain lit which is updated accordingly as time passes.

- pinValues: Its a 4 bytes array, each bit representing a light. It is used for the function setAll() to turn on or off all the lights.

- total_notes: The total amount of notes in the currently played song

- start_array: A pointer to the start array of the currently played song

- light_array: A pointer to the light array of the currently played song

- delay_array: A pointer to the delay array of the currently played song

```
//Definition of max and min volume
int vol = 25;
int minvol = 0;
int maxvol = 40;

static const uint8_t PIN_MP3_TX = 9; // Connects to module's RX
static const uint8_t PIN_MP3_RX = 8; // Connects to module's TX
AltSoftSerial softwareSerial(PIN_MP3_RX, PIN_MP3_TX);

// Creation of the DFPlayer object
DFRobotDFPlayerMini player;

#if !defined(STR_HELPER)
#define STR_HELPER(x) #x
#define STR(x) STR_HELPER(x)
#endif

volatile struct TinyIRReceiverCallbackDataStruct sCallbackData;

// Inicialization clock DS1302
#define CLK 14
#define DAT 15
#define RST 16
virtuabotixRTC myRTC(CLK, DAT, RST);

int starting_time;
int current_time;
int playing_song;
int current_note;
int delay_on_lights [32];
uint8_t pinValues [4];
int total_notes;
const int * start_array;
const int * light_array;
const int * delay_array;
```

**Figure 4.6** – *Global Variables Declaration*



Now that those variables are declared, we will start construction the turnOnLights() function. It will take the delay_on_lights array and the pinValues array as an input. It will set all the positions in pinValues to 0 to make sure that initially all lights are turned off. Then, we will in order read each value of delay_on_lights and if that position is greater than 0, we turn on the bit representing that light in the pinValues array. Doing this requires a bit of knowledge on bit operators, masks and on how the lights are arranged in the board. For this purpose, I have arranged a diagram which illustrates what bit of what byte turns on each light in the board:

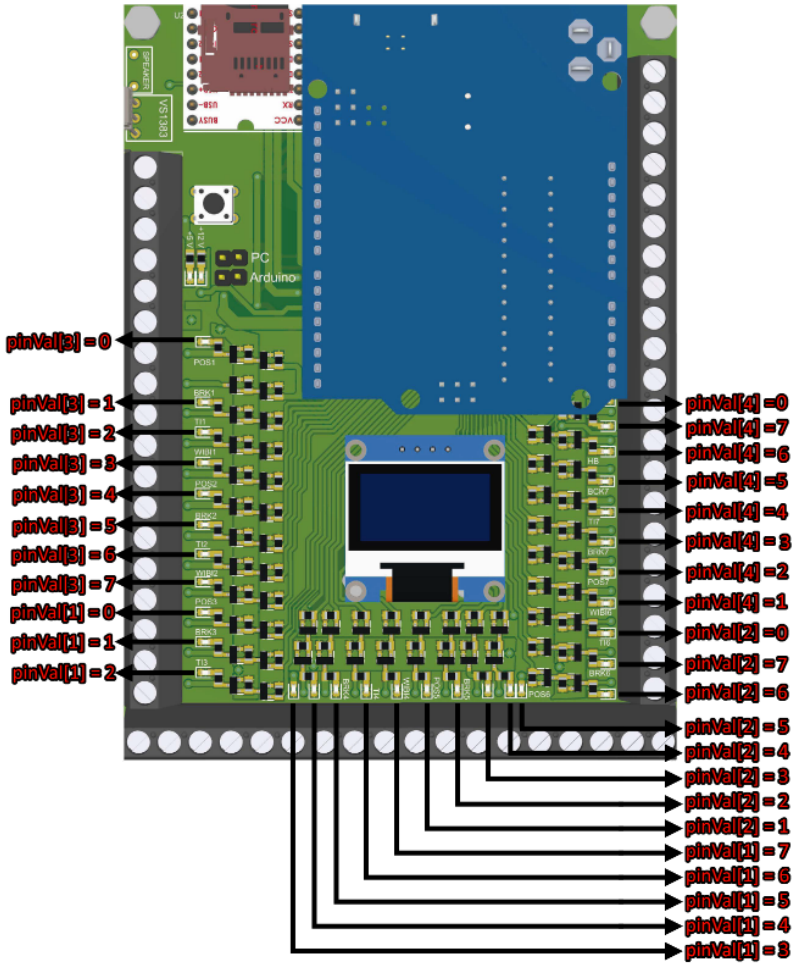



**Figure 4.7** – *Lights layout on the Arduino board*

As we can see, the order of the bytes is not the same order in which the lights are arange, so we take first 8 positions correspond to the range of pinValues[3], next comes pinValues[1], then pinValues[2] and finally pinValues[4]. It is also important to point out that pinValues[2] and pinValues[4] have their zero bit light right at the end instead of the beginning like the other bytes, so we will have to account for that too.

Once all of that is accounted for, we will use pinValues as an input for the setAll() function which will turn on all the corresponding lights.

```
//Turn on all lights given in the lights array and turn off all others.
void turnOnLights(int * lights,uint8_t * pinValues){
  //Set pinValues to blank
  pinValues[0] = { 0b00000000 };
  pinValues[1] = { 0b00000000 };
  pinValues[2] = { 0b00000000 };
  pinValues[3] = { 0b00000000 };

  //For each light position
  for (int i = 0; i < 32;i++){
    int pos = i;
    //If light is greater than 0, turn on the bit representing that light in pinValues
    if (lights[pos] > 0){
      uint8_t mask = 0b00000001;

      if (pos < 8){
        mask = mask << pos;
        pinValues[3] = ( pinValues[3] | mask);
      }
      else if(pos >=8 && pos < 16) {
        mask = mask << (pos % 8);
        pinValues[1] = ( pinValues[1] | mask);
      }
      else if (pos >= 16 && pos < 24){
        mask = mask << ((pos+1)%8);
        pinValues[2] = ( pinValues[2] | mask);
      }
      else if (pos >= 24 && pos < 32){
        mask = mask << ((pos+1)%8);
        pinValues[0] = ( pinValues[0] | mask);
      }
    }
  }
  //Turn on all the lights
  sr.setAll(pinValues);

}
```

**Figure 4.8** – *Lights_ on function*

The next step is to feed to this function we just created the information provided by *musiclights.h*. This is not a trivial task, since we need to make it in such a way that the board can be interacted with while a song is playing so we can modify the volume, change songs or even stop the program.

One way to solve this problem is with multithreading, but since our Arduino board doesn't have enough processing power to do that, we will use interruptions. An interruption is a signal to the processor that disrupts whatever the processor is doing to execute some code designed to react to whatever external stimulus is being fed to the Arduino. In this case, our interruption will be provoked by our IR Remote using the TinyIRReceiver library as explained in Chapter 3. We will get into the code of the interruption later but for now, we will focus on the code that we will use for the void loop() function.

First of all we have to check the playing_song variable to check if a song should be playing, in case its -1, nothing happens and we keep checking. The variable will be changed in the interruption so we don't have to worry about that for now either. If playing_song says that a specific song should playing (playing_song greater than 0) we will assign the pointer variables and the total_notes value to the correspondent arrays and value in *musiclights.h*. We will also set the starting time to the current time, the current_note to 0 and the playing_song variable to 0 so that we know a song is playing. When we were saving the information of each song in musiclights.h, we discussed the addition of a #define NOTESX line: this line will help us check if the song we are locking for exists in the file or not. Unlike a normal if statement, #ifdef is checked at compilation time and if its clause isn't fulfilled, the code it precedes isn't compiled at all, so if we have #ifndef NOTES1, the code will not be compiled unless there is a #define NOTES1 line at musiclights.h and if there isn't, we will just set playing_song to -1 indicating that no song is playing now.

```
//Set the arrays acording to the song you want to play
switch(playing_song){
  case 1:
    //If the song is declared in the header file, we select it
    #ifdef NOTES1
    start_array = start1;
    delay_array = delay1;
    light_array = light1;
    total_notes = notes1;
    playing_song = 0;
    starting_time = millis()/10;
    current_note = 0;
    player.play(1);
    break;
    //If the song is not declared, we do nothing and restart the playing song variable
    #else
    playing_song = -1;
    #endif
  case 2:
    #ifdef NOTES2
    start_array = start2;
    delay_array = delay2;
    light_array = light2;
    total_notes = notes2;
    playing_song = 0;
    starting_time = millis()/10;
    current_note = 0;
    #else
    playing_song = -1;
    #endif
  case 3:
    #ifdef NOTES3
```

**Figure 4.9** – *Music Check Function*

In the case that we exit that verification and there is a song playing, we will execute the next chunk of code:

1. We start by storing the current time as current_time. We will also need to know: What note must be turned on next, when we will have do it and how long will it stay on. We can obtain this information from current_light, current_start and current_delay respectively more specifically from their item in the current_note position. We will store that information in current_light, current_start and current_delay.

2. Next, we will check if the time has come to start a new note, we will need to know if there are notes that haven't been played (total_notes greater than current_note) and if it's time for the next light to be turned on (current_start less than current_time). If that is the case, we proceed as normal, in any other case we skip next step

3. If a note should be played, we add its delay plus the current time to the corresponding position (current_light) in the delay_on_lights array and once we have done that, we increment current_note and check if we have notes left to play. If we don't the song is over otherwise we update current_start, current_delay and current_light

4. Finally, we update delay_on_lights by checking if the time at which they will turn off has already passed, if it has, we convert that to 0, else we do nothing. Once that is ready we pass delay_on_lights to the turnOnLights() function and repeat from step1

We have also added a small delay between iterations of the cycle so that there is capability of meaningful change between each iteration. Also, because of a technical problem with the board, the LED at the

$24^{th}$

position must always be on so that the other lights work otherwise, nothing even starts. This is just a temporary solution but a small line has been added to keep that light on in the meantime.

With that out of the way, it is now time to talk about the code in the interruption. The interruption will take control of the CPU whenever an IR signal is received by the board. This allows us to run the main program all the time while being able to attend to the instructions given by the user via the remote. This interruption is defined in Chapter 3 so we will go to the important part.

In this code we will read the information received from the remote, and depending on what number from 1 to 9 it will try to play that song which means calling the MP3 player function (player.play()) and setting playing_song to the correct number. The Up and Down messages control the volume and okay pauses the song.

```
// various actions for different buttons
switch(aCommand)
{
  case ONE  : display.clearDisplay();display.setCursor(0,0);Serial.println(F("1 is pressed"));di
  case TWO  : display.clearDisplay();display.setCursor(0,0);Serial.println(F("2 is pressed"));di
  case THREE: display.clearDisplay();display.setCursor(0,0);Serial.println(F("3 is pressed"));di
  case FOUR : display.clearDisplay();display.setCursor(0,0);Serial.println(F("4 is pressed"));di
  case FIVE : display.clearDisplay();display.setCursor(0,0);Serial.println(F("5 is pressed"));di
  case SIX  : display.clearDisplay();display.setCursor(0,0);Serial.println(F("6 is pressed"));di
  case SEVEN: display.clearDisplay();display.setCursor(0,0);Serial.println(F("7 is pressed"));di
  case EIGHT: display.clearDisplay();display.setCursor(0,0);Serial.println(F("8 is pressed"));di
  case NINE : display.clearDisplay();display.setCursor(0,0);Serial.println(F("9 is pressed"));di
  case UP   : vol = vol +1; if(vol>40){vol = maxvol;} player.volume(vol);display.clearDisplay();
  case DOWN : vol = vol -1; if(vol<0 ){vol = minvol;} player.volume(vol);display.clearDisplay();
  case LEFT : Serial.println(F("LEFT is pressed")) ;break;
  case RIGHT: Serial.println(F("RIGHT is pressed"));break;
  case OKAY : Serial.println(F("OK is pressed"))   ;break;
  case ZERO : Serial.println(F("0 is pressed"))    ;break;
  case STAR : Serial.println(F("STAR is pressed")) ;break;
  case HASH : Serial.println(F("HASH is pressed")) ;break;
  default:
  Serial.println(" other button : ");
  Serial.println(aCommand, HEX);
}// End Case
```

**Figure 4.10** – *Interrupt Code Function*

## 4.3 PyQt GUI design and implementation

Because the first part of this assignment is developed in Python I decided that the GUI of the application should also be made in Python, so after a rough first sketch with the Tkinter library, I decided to use PyQt to create a more sophisticated and much better looking GUI.

Our GUI must have the following functionalities:

- It should allow you to load any song into the board. This should load the MP3 file and add the song to the header file as seen in the first section of this chapter
- It must be able to remove songs from the board by deleting the MP3 file and the song from the header while renaming all files accordingly.
- It has to be able to compile the code in the board without any external functionality. This includes selecting the correct board from a list and giving proper errors when compiling is not possible

We will create the main layout of the GUI in QtDesigner as described in Chapter 3: PyQt. It will look like this and will be saved as Organ.ui:

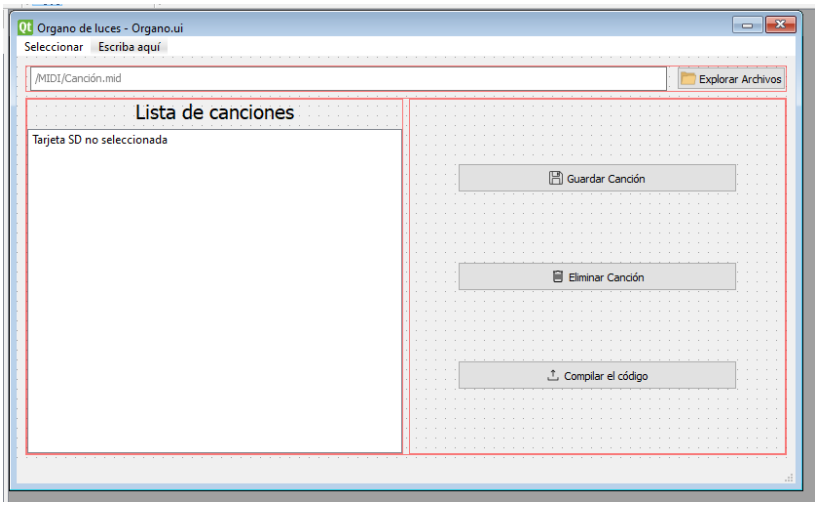



**Figure 4.11** – *QtDesigner view of the GUI*

As we can see, because of the use of layouts, the GUI will be responsive and will adjust the size of its elements to correspond with the current resolution of the main window.



Once the design of the GUI is created, we need to export it to Python, which will create the Organ.py file. This file contains all the necessary code for the GUI, but its buttons are disconected and don't really do anything. We will need to create some functions to connect them to the buttons.

### 4.3.1 Add songs to the board

Before we can save a song on the board we must have first found the path to the SD card and also selected the path to the song we want to add. Without those requirements, the save a song function shouldn't work.

We will create the searchSD() function to the GUI. It will be located on the top left drop-down menu *Seleccionar* on the *Buscar SD* button. It will open a file explorer in which you will be able to select a folder which will be stored in the SDpath variable and its contents will be displayed in the songs list at the left part of the screen.

Now, we have to select the MIDI file from which we will extract the information for the song. This functionality will be added to the top part of the GUI, on the *Explorar archivos* button. It will also open a file explorer only in this case it will only allow you to select .mid files. Once a file is selected the path to said file will apear in the text box on the left. You can also write the path there if you want.

Finally, we will add the saveSong() function. This function will be conected to the Guardar Canción button on the right side of the GUI. It will first check if both the file path and SD path are valid and it will throw an error message in case any of them is not. Once that is checked, we will save the song to the device.

We first convert the file to MP3 and for that we will have to use the program Fluidsynth as described in Chapter 3: Fluidsynth. By calling the fluidsynth command using subprocess, it will convert it to MP3 and store it in the proper folder provided by SDpath. We will also add a tag before the name of each song so that they maintain an order within the SD card. This is useful because the Arduino player stores the songs in this order and are referred internally as numbers.

Now that the song is stored as an MP3 on the SD, we will store it in the header file of the board.

First, we cheek if said file exists and if not, we create it and populate it with the necessary elements

(#ifndef declarations and first song tag), then we call the function add_midi() which calls to the functions we wrote in the first section of this chapter to create a header and then to add_song_to_header(). Once this is all done we update the song's list.

### 4.3.2 Remove songs from the board

Once the SD is selected, all the songs in the board will appear on the left side of the GUI, from where they can be selected. We will add a function deleteSong() which will delete the selected song. It will first check if there is an SD route and then it will check if a song is selected, if its not the case, it will throw an error message, otherwise it will start the deletion process.

First, we get the number of the song we need to remove. This is important because unlike added songs which are added at the end of the list, songs can be deleted from the middle or the beginning and the other file must be renamed accordingly. After we have the number of the song we need to delete, we will remove the file from the SD card and then we will update the tag of each song after that by updating its number on the list accordingly to its new position.

Finally, we call the delete_song_from_header() function and update the songs list

### 4.3.3 Compile code

The last task our GUI must be able to handle is the compilation of Arduino code in the board, and for that we will use pyduinocli, a library created for this intent and purpose which we discussed in .



To compile Arduino code into a board we will need three things:

- The Arduino .ino file which contains all the Arduino code. This file is called project_granasat and already exists
- The Arduino port address which we will need to find out
- And finally the fqbn, which is a descriptor of the type of board we are using and will require an explanation later.

The Arduino port address is obtained by using the board.list() function from pyduinocli: It returns a dictionary with some elements, but we only care about "result", which contains a list of information about the different boards attached to the computer. From each board, we will want its port address which is accessed via "port" then "address". The final information we'll want is obtained by looking for Arduino.board.list()["result"][number_of_board]["port"]["address"]. Now that we know how to access the information, we will add it to the GUI. We will add a new menu in the "Seleccionar" QMenu which will be called Seleccionar Arduino. Whenever it opens, it will call the board.list() function and create a button for each result it gets, said button will be named after the port is assigned to and we will add a function to it which will save said port as a global variable which we will call savedPort whenever we click the button. This allows us to select what board we will want to compile our code.

For the fgbn, the code is pretty similar, using board.list() too and looking at ["result"][number_of_board]["matching_boards][0]["fqbn"] instead but for some reason our board is not recognised so we cannot access the information this way. I've been looking for ways to get the information out of the board but I couldn't find any, so to fix this I've decided that I'll add a default fqbn which will be the one for Arduino Uno boards: "arduino:avr:uno" which will be used whenever the board can't provide one. This is by no means a perfect solution but it works for the board we are using.

Now that we have all the things we need, we can start working on the compiling function. This function will be called by the "Compilar el código" button and should do the following:

- It will check if a port has been selected, if not, it will give an error message
- Then, it checks if the port that has been selected is still conected, if not, it gives an error message too
- If the port is connected, it will try to check if we can get its fqbn. If it can, it will store it and if it can't, it just uses the default one.
- Finally, if nothing else failed, we run the compile and upload functions.
- When it finishes, it shows a complete compilation message

And with that finished, the application is completed

# Chapter 5

# System Testing

This chapter will be dedicated to the use and understanding of the different parts of the system.

## 5.1 GUI usage

In this section, we will test each function of the interface and how each of them works perfectly. We will start by saving a song:

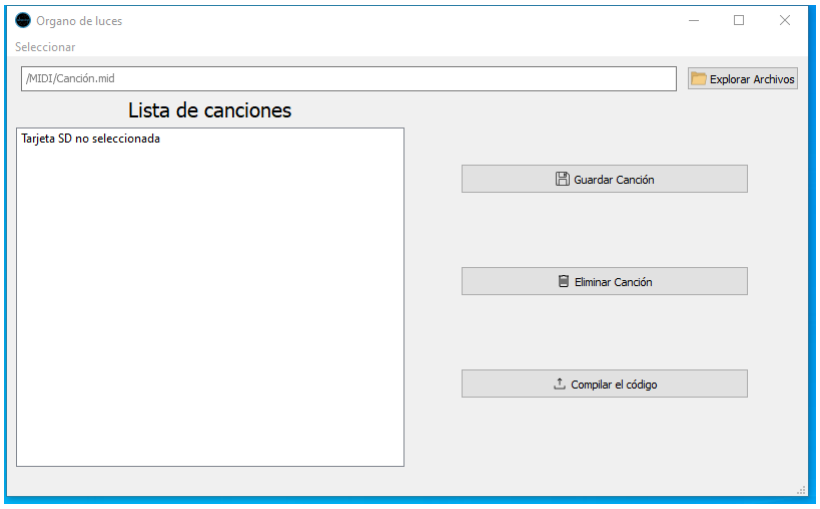



**Figure 5.1** – *Main screen of the interface*

As we can see, the interface starts with no SD selected and no MIDI selected. If we try to save a song as is, we will get an error warning us that no SD card has been selected:

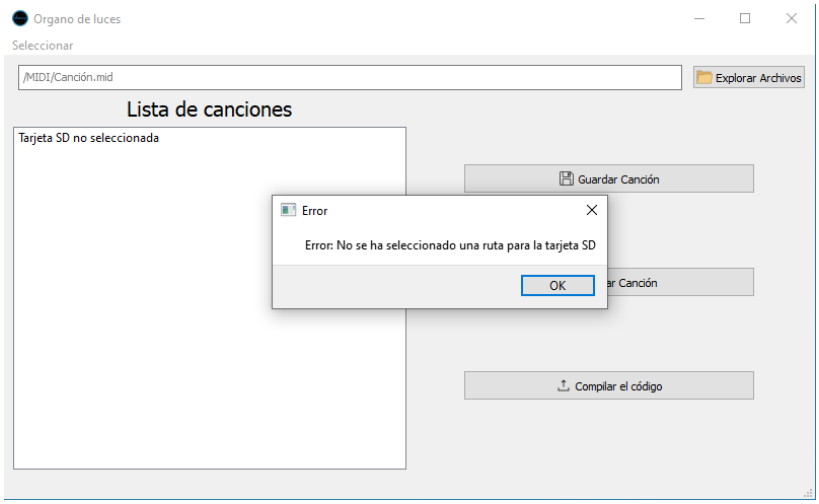



**Figure 5.2** – *No SD selected error*

This error is solved by selecting an SD card path in "Seleccionar" menu on the top left. If we click there, the "Buscar SD" action will appear and we will be able to select the folder in which our SD card is located from there once the SD card is selected, all the songs will appear in the songs list. If the SD is empty, so will the songs list:

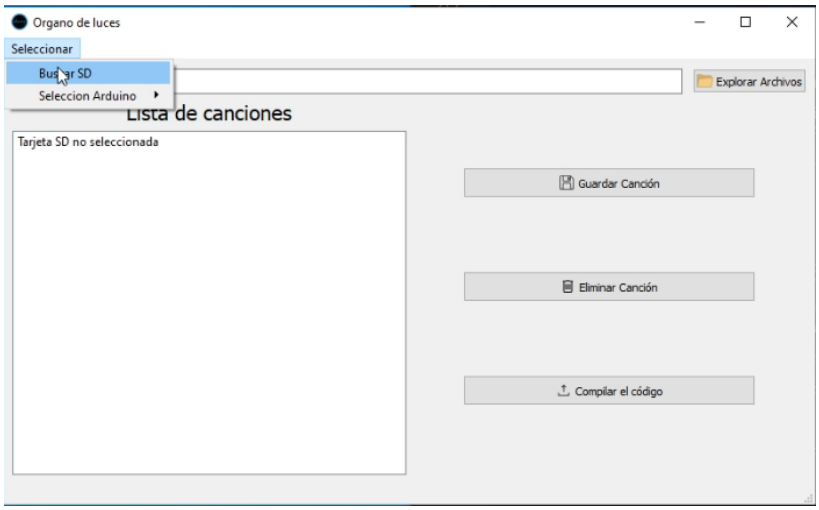



**Figure 5.3** – *Select SD button*

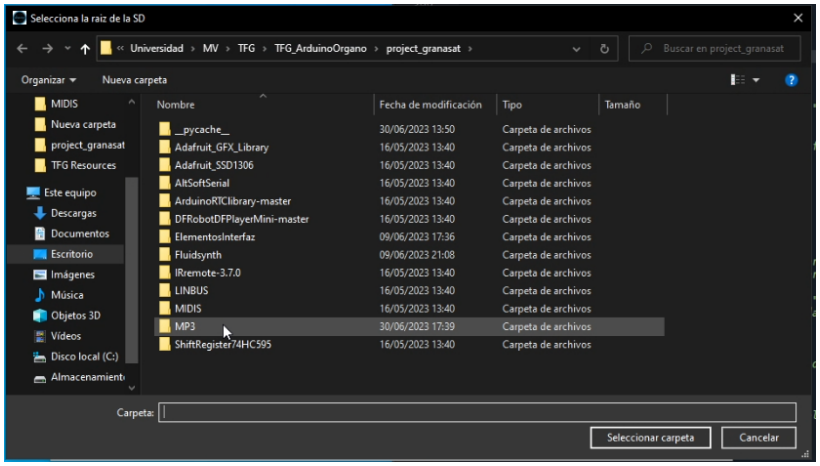



**Figure 5.4** – *Find SD path*

If we try to save a song now, the GUI will inform us that no MIDI file has been selected, so we will do that next by either typing its path in the upper box or by clicking the "Explorar Archivos" button.



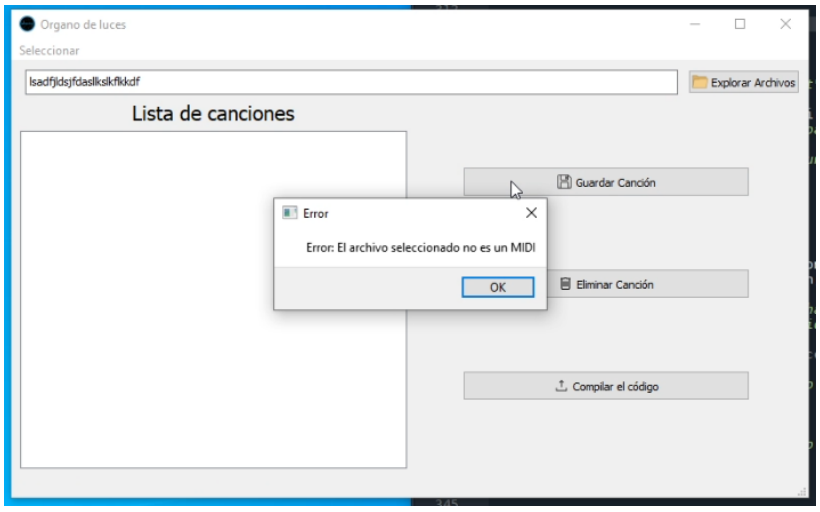



**Figure 5.5** – *No MIDI was selected error*

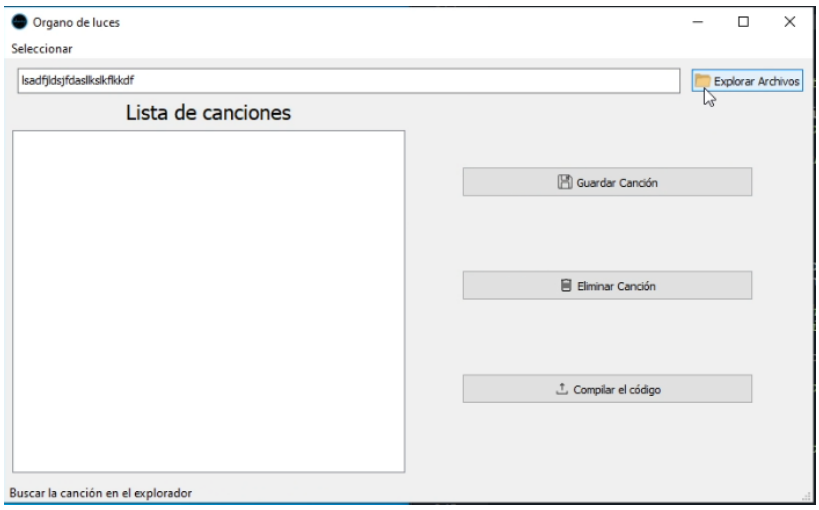



**Figure 5.6** – *Find file button*

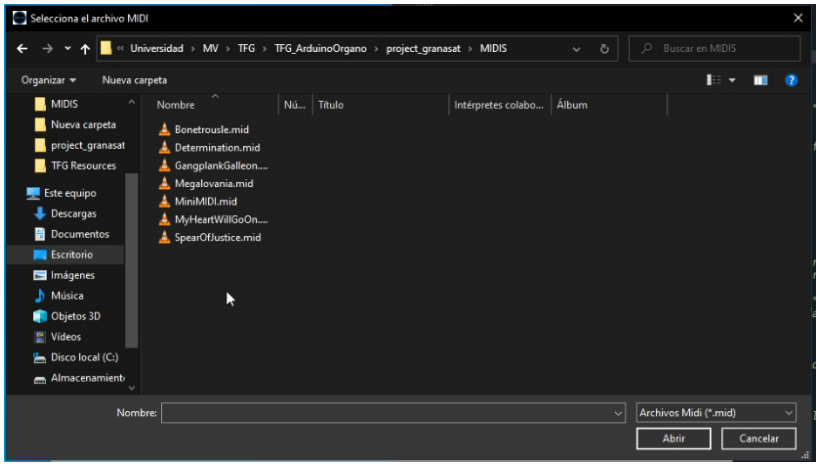



**Figure 5.7** – *Find MIDI path*

Once the file is located, if we click the save song button, it will properly save the song in the SDCard, in the songs list and in the musiclights.h file.

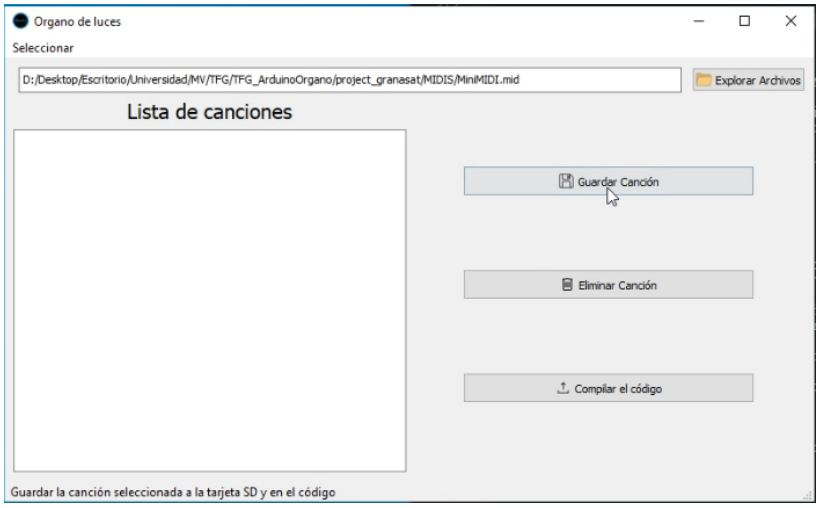



**Figure 5.8** – *Ready to save a song*

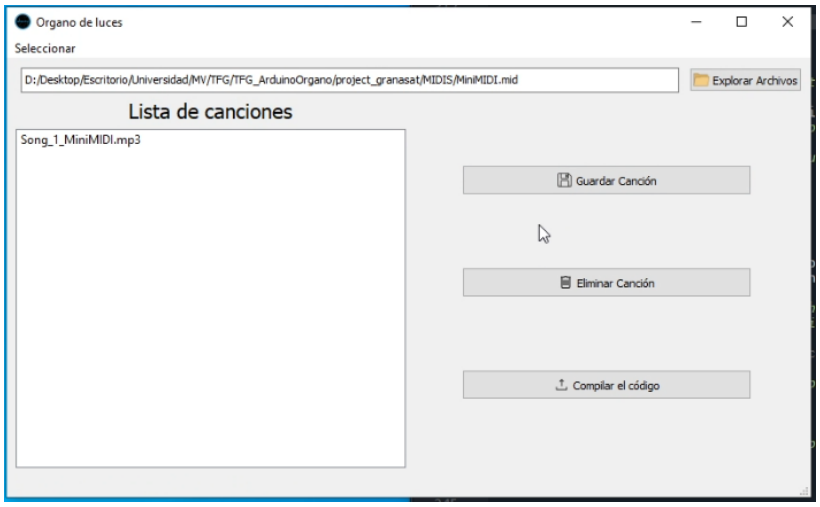



**Figure 5.9** – *Song saved*

This is the resulting files of this operation

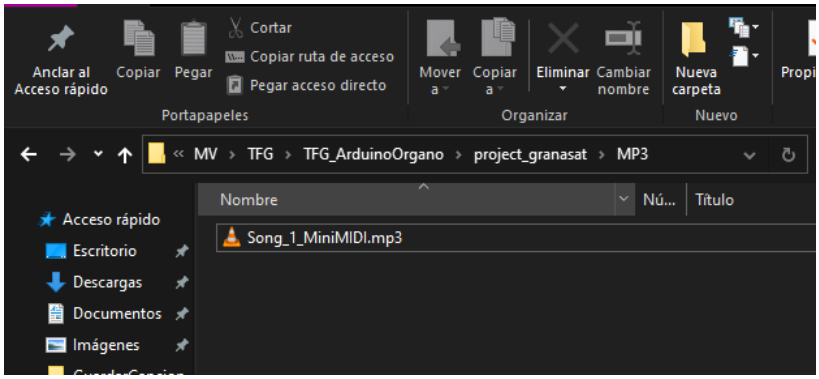



**Figure 5.10** – *Resulting SD files*

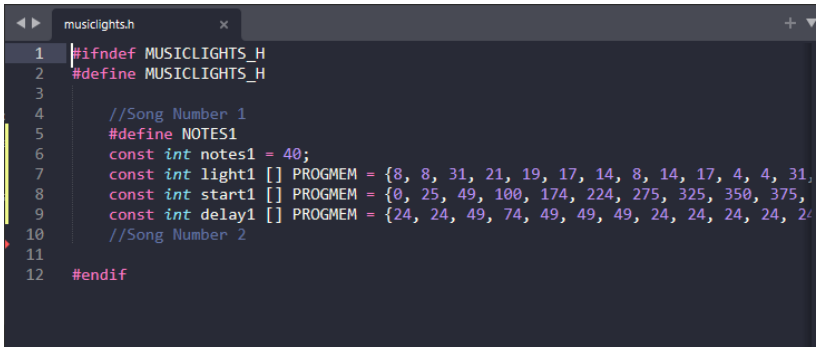



**Figure 5.11** – *Resulting header file*

5

Next, we are going to delete a song. If we try to just hit the delete button, we will get an error message because no song has been selected from the list.

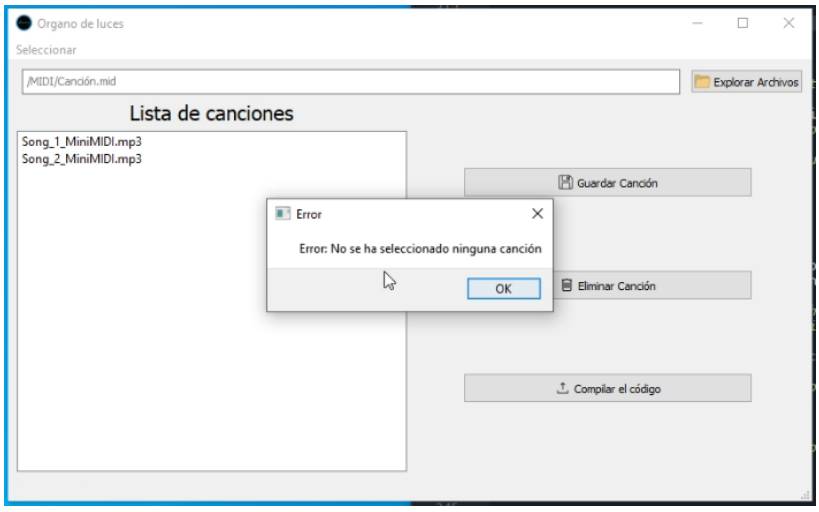



**Figure 5.12** – *No song was selected*

So, first we will select a song from the songs list and then we will hit the delete button. As we can see, all the other songs are renamed properly when we delete a file making sure that we don't have song number 3 without a song number 2:

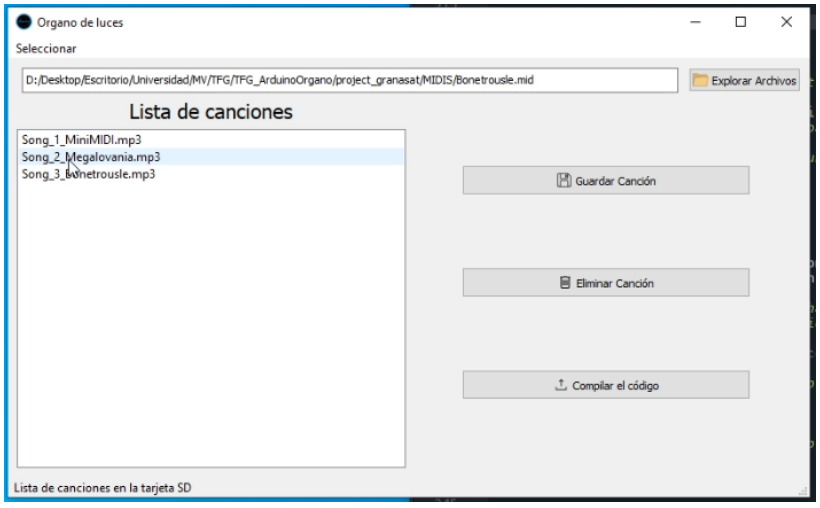



**Figure 5.13** – *We select a song*

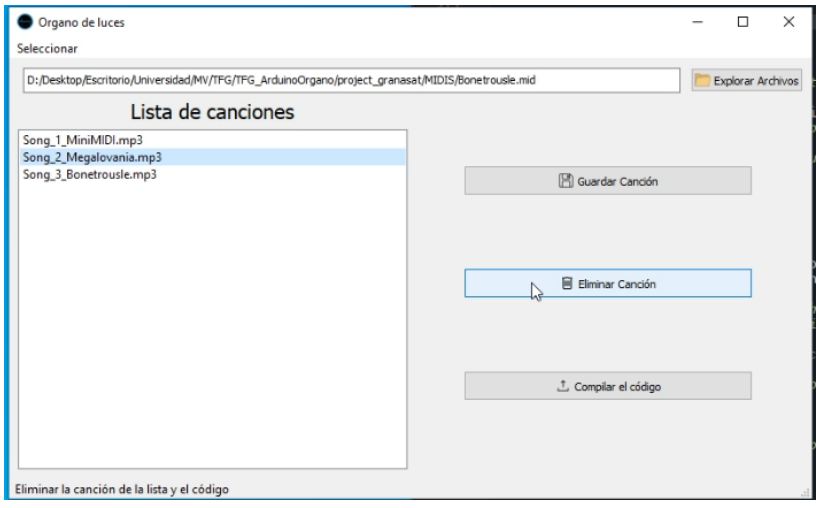



**Figure 5.14** – *We hit delete*

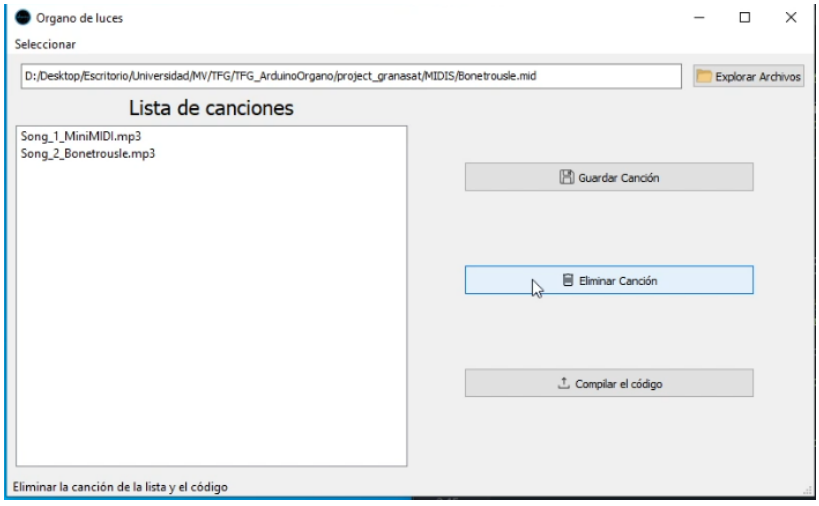



**Figure 5.15** – *The song is deleted*

Finally, its time to compile our code. For that, we will use the "Compilar el código" button:

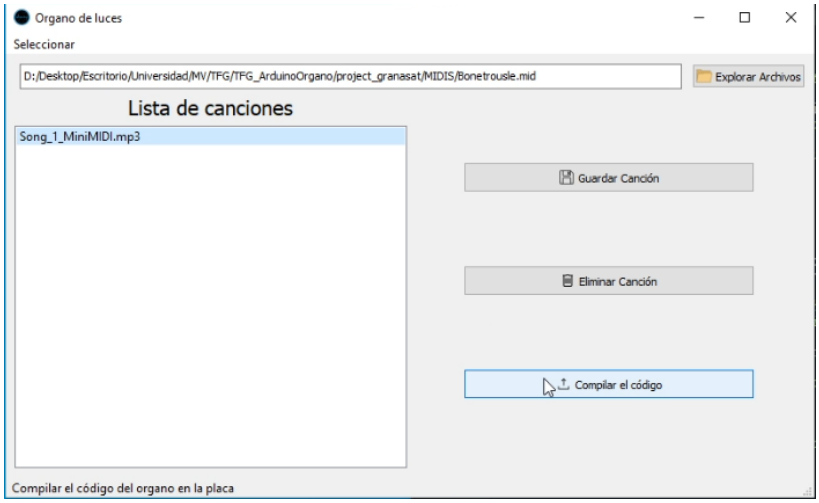



**Figure 5.16** – *Compile Button*

If we try to compile it now, we will get an error message because we have not yet selected the port where the board in which we want to compile the code into is connected. So to do that we will use the "Seleccionar Arduino" menu on the top left.

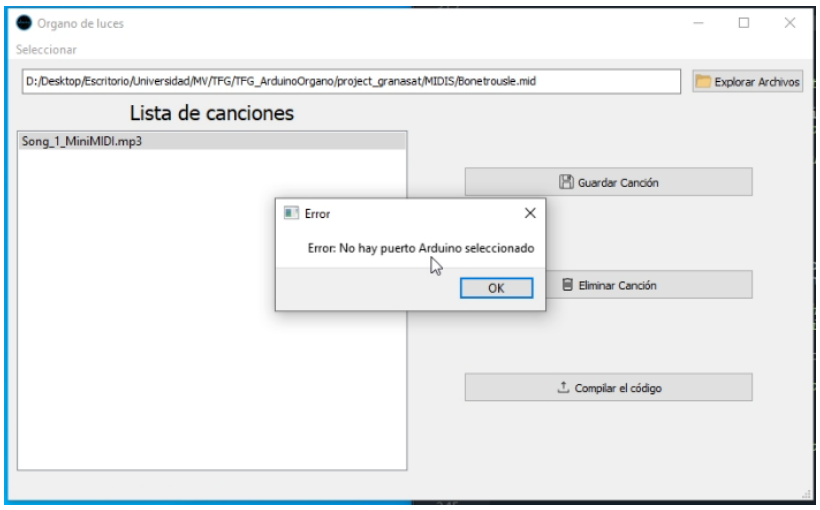



**Figure 5.17** – *No Arduino port was selected*

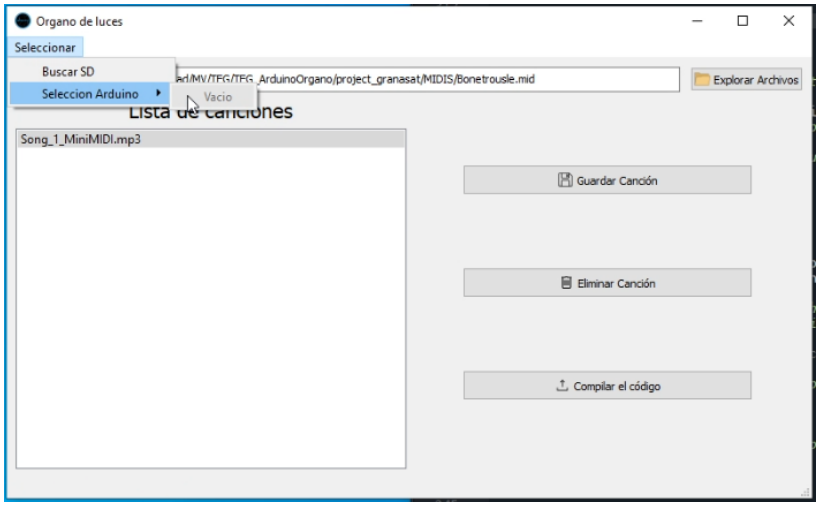



**Figure 5.18** – *Port menu (Empty)*

As we can see, when there are no boards connected to the board the menu is empty, but as soon as we connect a board, it appears in the menu and we can select it. This will give a succesfully selected port message:

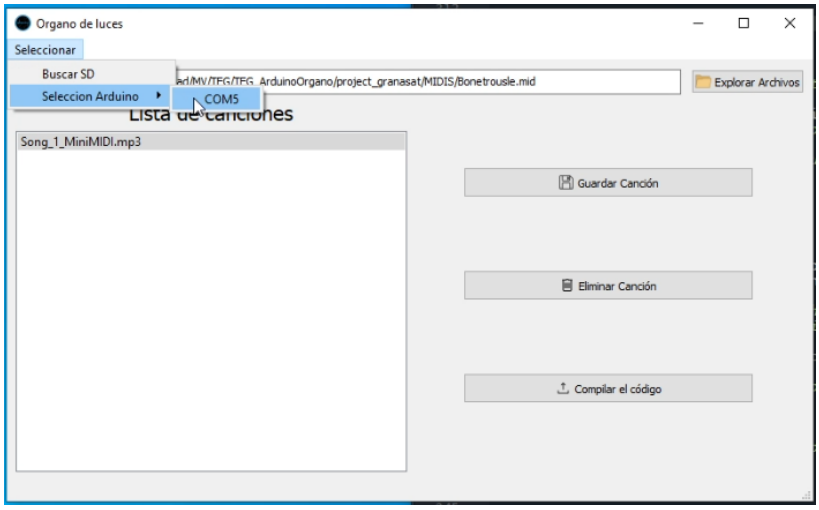



**Figure 5.19** – *Port menu*

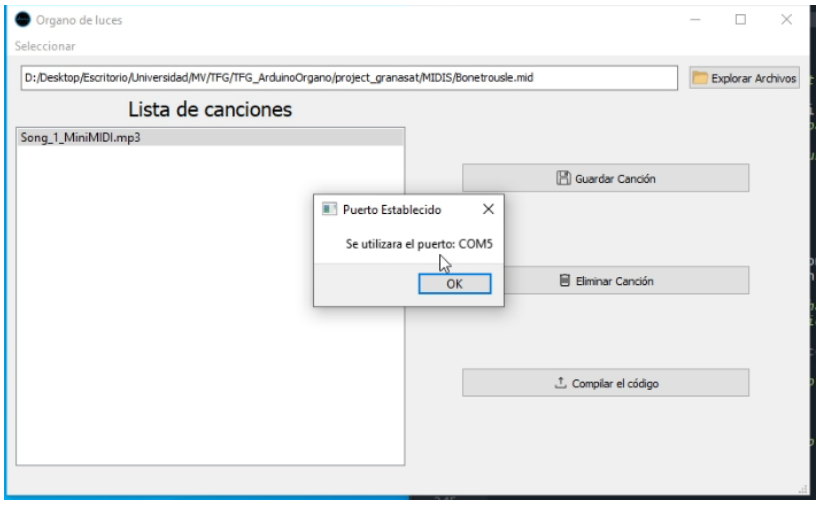



**Figure 5.20** – *Port selected successfully message*

Finally, we can click the compile button to compile our code. If the board has been disconected, it will give us a warning that the board is unavailable and it will not compile, but if the board is connected, it will compile the code like we wanted:

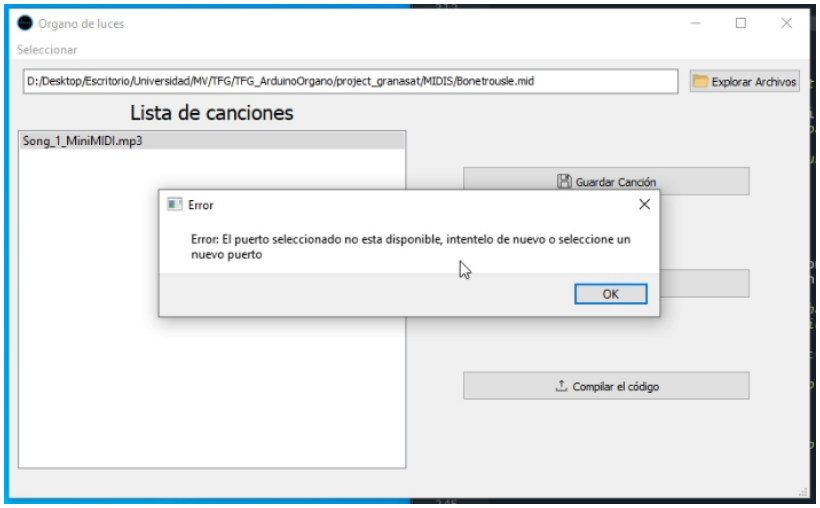



**Figure 5.21** – *The board has been disconnected*

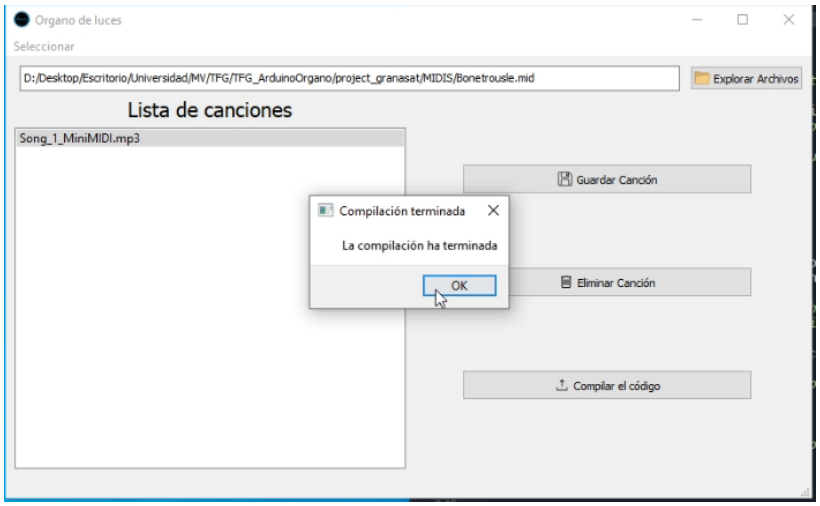



**Figure 5.22** – *the code has been compiled*

I have created a video explaining all the functionalities of the board, which can be seen in the following link: https://drive.google.com/file/d/1AofBIRvY2QK2PxBxPGs7y97nTvQoF27T/view?usp=sharing

## 5.2 Board testing and demonstration

The next section, is a demonstration of the organ on action. Because this is very difficult to test in the document, I will provide a video of the board working in the following link: https://drive.google.com/file/d/1qhaujn55S9aTiM17rAAKWflDxjjs0jOK/view?usp=sharing

As we can see in the video, the board lights up acordingly to the music which is exactly what we wanted.

# Chapter 6

# Conclusion and future improvements

Now that we have finalized the testing and demonstrated that both the board and the interface work, we can check that all the objectives expressed at the beginning of this document have been fulfilled. The final product is capable of taking MIDI files of any category and convert them to MP3 while creating a sequence of lights that accompany the different notes of the song and then reproduce said files and lights.

With that being said, with more time and resources put into it, there are a lot of places in which it could be improved:

- Due to the fact that the lights sequences of the songs are hardcoded, we cannot store them in the SD card and must be stored in memory and because of this we cannot really store very long songs
- Because we cannot interact with the SD card inside the board, we need to remove it from it and insert it into the PC each time we want to change songs.
- The interface could be improved with additional features like translation, an MP3 player, etc...
- We could also change so that the board directly reproduces MIDI files without the need to convert them to MP3

Finally, I would like to talk about my experience working on this thesis. Since it's been the first time I've worked in a project of this magnitude in which I had to organize everything myself without deadlines this has been a very important learning experience. I've had the opportunity of learning about a lot of diverse themes and acquire a lot of obscure knowledge (especially about MIDI files) which is always a lot of fun.

I think this is also a very interesting project because unlike any of my other projects, its something that will be seen and used by other people which really gives me a sense of accomplishment.

# Bibliography


[1] A. Swift, *An introduction to MIDI*. https://web.archive.org/web/20120830211425/http://www.doc.ic.ac.uk/~nd/surprise_97/journal/vol1/aps2/.

[2] S. Hutchinson, "The midi protocol: Midi messages." https://www.youtube.com/watch?v=2BccxWkUgaU.

[3] *MIDO*. https://mido.readthedocs.io/en/latest/.

[4] "What is arduino," 2023.

[5] *Arduino-Cli Docs*. https://arduino.github.io/arduino-cli/0.33/.

[6] "Compiling arduino code in python." https://www.tinkerassist.com/blog/compile-upload-arduino-code-with-python.

[7] *Fluidsynth*. https://www.fluidsynth.org/.

[8] *PyQt5 Reference Guide*. https://www.riverbankcomputing.com/static/Docs/PyQt5/.

[9] GranaSat webpage. https://granasat.ugr.es Accessed: October 2021.

[10] *MIDI control messages*. https://gigperformer.com/program-change-management/.

[11] E. D. Águila, "Design of demosntrator device for automotive pilots of high performance vehicles," bachelor's thesis, University of Granada, 2022. https://digibug.ugr.es/handle/10481/80480.