Politenico di Milano

Dipartimento Elettronica, Informazione e Bioingegneria

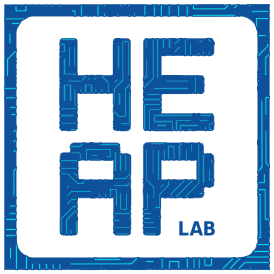
HEAPLab Project Report

# Digital Audio Effects

*Authors:*
Generoso Imparato
Yazan Matar

*Supervisor:*
Federico Terraneo

September 14, 2021




*Github repo:*
https://github.com/GenniImparato/Digital_Audio_Effect

# Contents

# 1 Introduction

The aim of this project is to create a platform to experiment with real-time DSP effects on audio, with an STM32 microprocessor, using a multi-threaded, real-time environment.
We designed an audio chain system with the following specifications:

- Input audio can be selected from two sources:
  - Audio sampled from an analog audio signal coming from a jack connector
  - Audio generated by a software audio synthesizer
- Audio stream coming from the selected source can be sent through different selectable DSP effects
- Output audio can be heard in real-time with a speaker, as well as earphones through the jack connector
- Synthesizer, as well as the effects, has parameters that can be changed in real-time.

To realize such functionalities, and to better visualize, hear, and control the system, we designed an auxiliary PCB with extra hardware which will be described in detail in the relative section.

# 2 Design and Implementation

In this section we describe the hardware components and functionalities as well as the software implementation and how those components interact with one another.

## 2.1 Hardware

In the following sections we analyze in detail the hardware components used for this project, which are the **STM32F407 Discovery Board** and the **custom PCB** designed to improve the user experience.

### 2.1.1 STM32F407 Discovery Board

The microcontroller used for the project is the STM32F407 Discovery Board, which offers (mainly for this project) the following features:

- STM32F407VGT6 microcontroller featuring 32-bit Arm® Cortex®-M4 with FPU core
- 1-Mbyte Flash memory and 192-Kbyte RAM
- Audio DAC with integrated class D speaker driver
- User and reset push-buttons
- Stereo headphone output jack
- External application power supply: 3V and 5V

For more specifications about the board, read the datasheet and the user manual or go to the manufacturer website.

### 2.1.2 Custom PCB

We designed a PCB in Eagle with connectors to insert the STM32F407 discovery board. The PCB contains extra hardware useful for the project, such as:

- Speaker with an amplifier circuit
- Audio buffer for ADC sampling
- Led matrix [10x15]
- Linear potentiometers (for effect controls)
- Logarithmic potentiometer (for amplifier volume)
- External power supply

Details on the PCB are shown in the next sections.

#### 2.1.2.1 Circuits

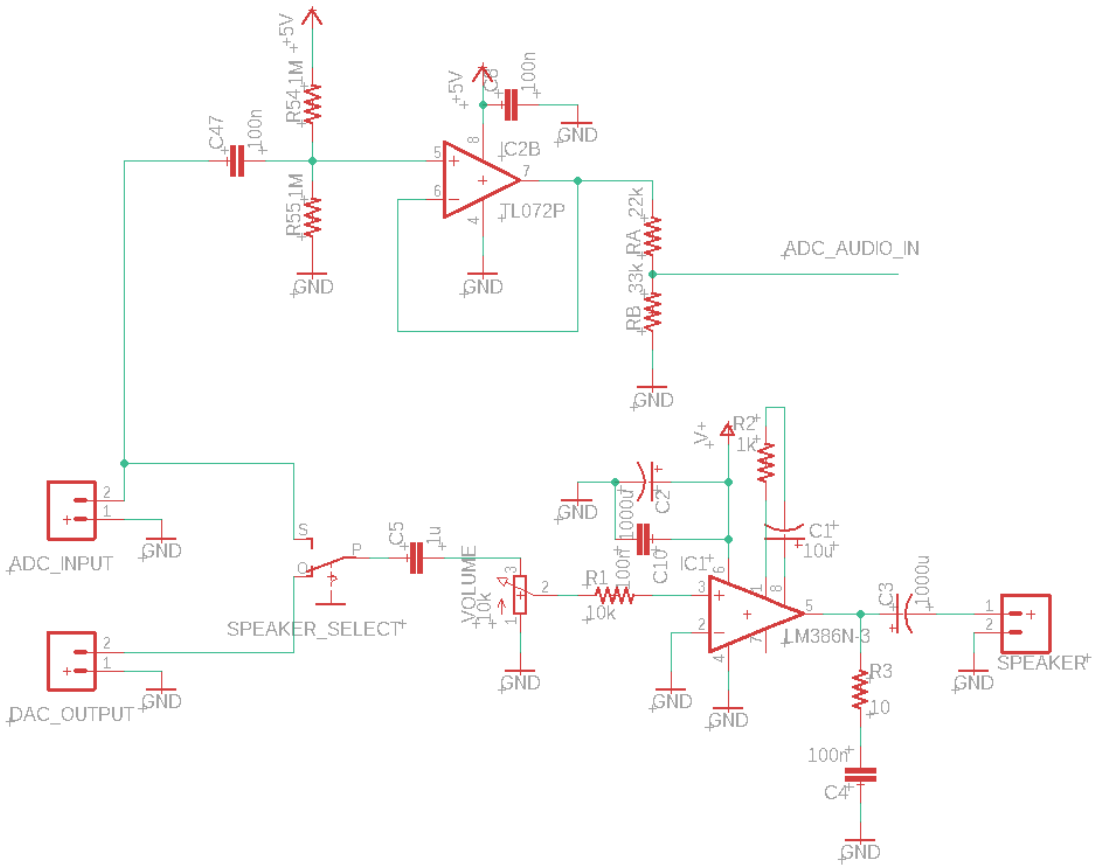



Figure 1: Audio circuit

This circuit contains a buffer between the input analog audio signal and the STM32 internal ADC input.
The signal is shifted in range from 0 to 3.3V and centered around 1.65 V to be sampled by the ADC.
It also contains the amplifier circuit that drives the 8 ohm speaker on the board, including a volume potentiometer.
The input of the speaker amplifier can be selected with a switch between the raw input audio and the DAC output of the discovery board.

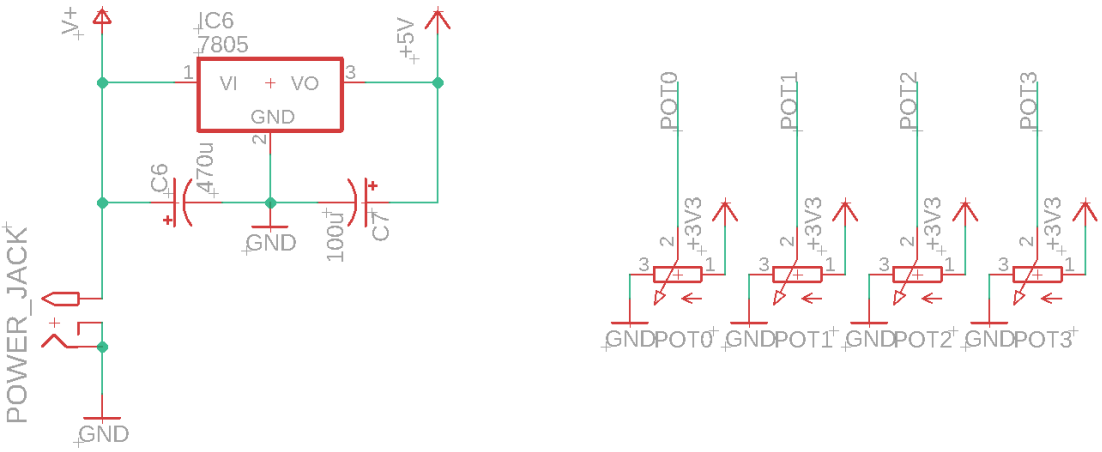



Figure 2: Power and potentiometers circuit

The input voltage from the power is applied directly to the speaker amplifier power supply, while the voltage for the discovery board is obtained by a 5V voltage regulator.

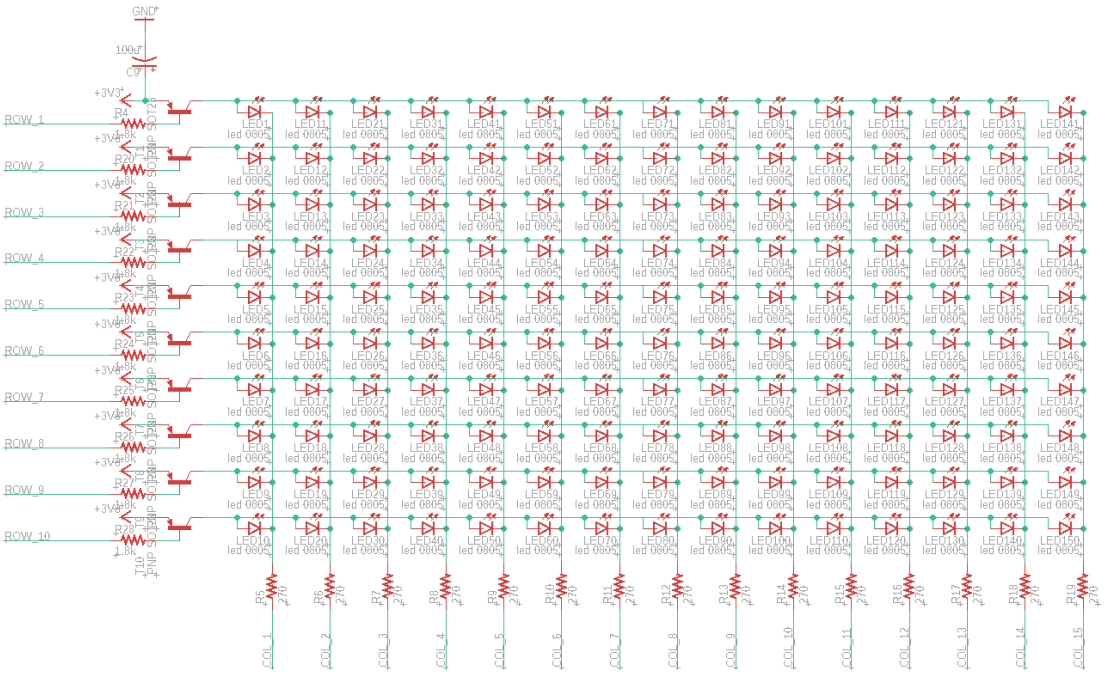



Figure 3: Led matrix circuit

The led matrix is made with 150 SMD leds, divided in 10 rows and 15 columns.
Each row is controlled by a PNP transistor with the base connected to a GPIO pin of the discovery board. Columns are connected directly to GPIO pins.

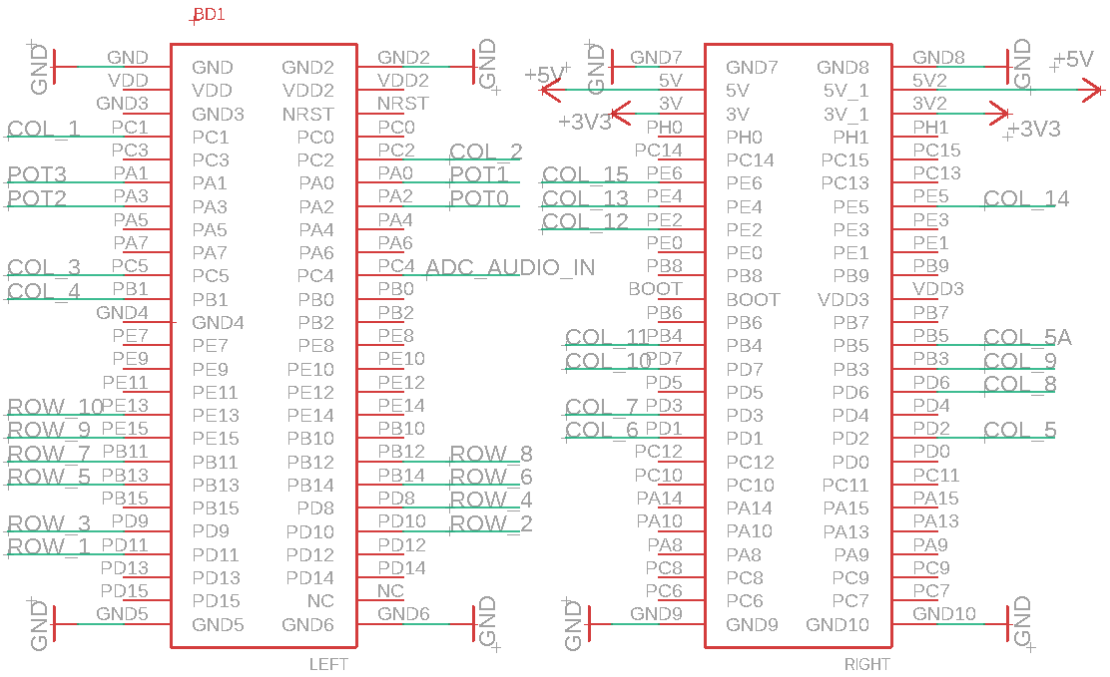



Figure 4: STM32F407 Discovery Board connectors

This connectors allow to easily insert and remove the STM32F407 Discovery Board from the PCB.

#### 2.1.2.2 Layout

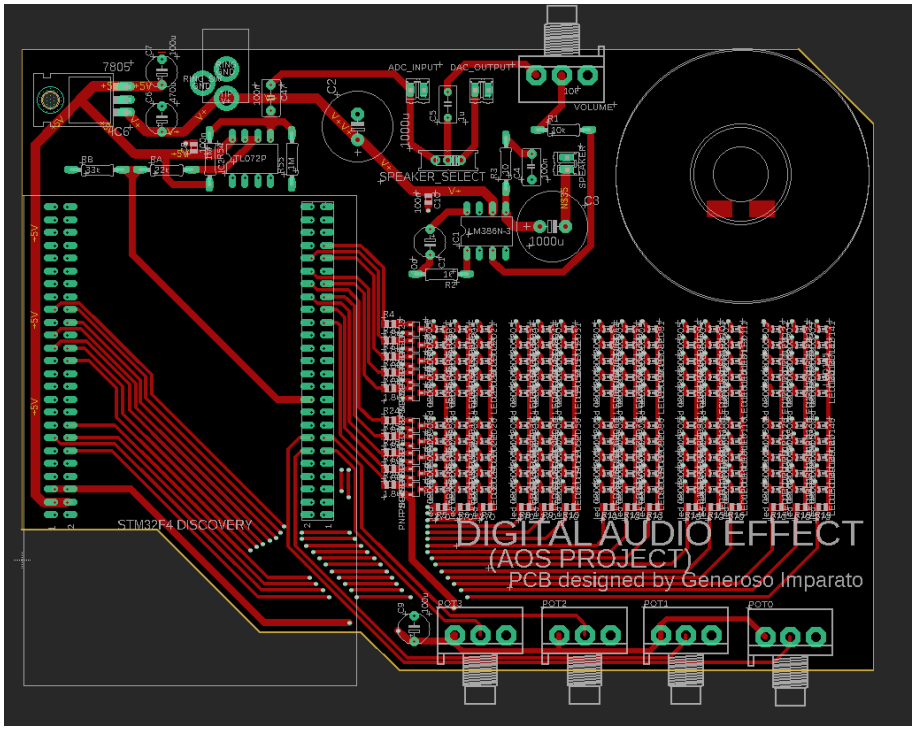



Figure 5: top layer

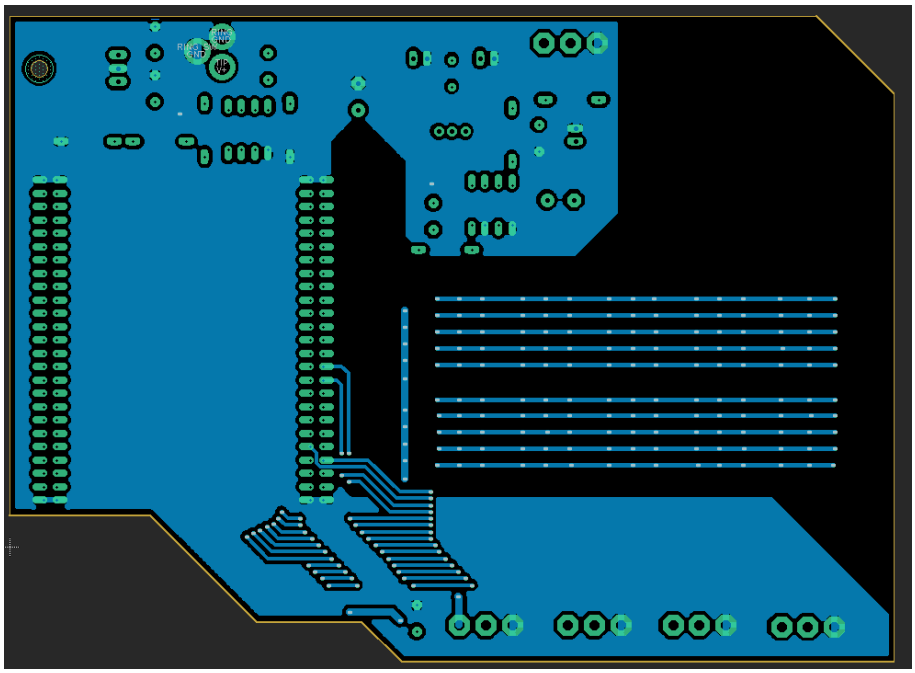
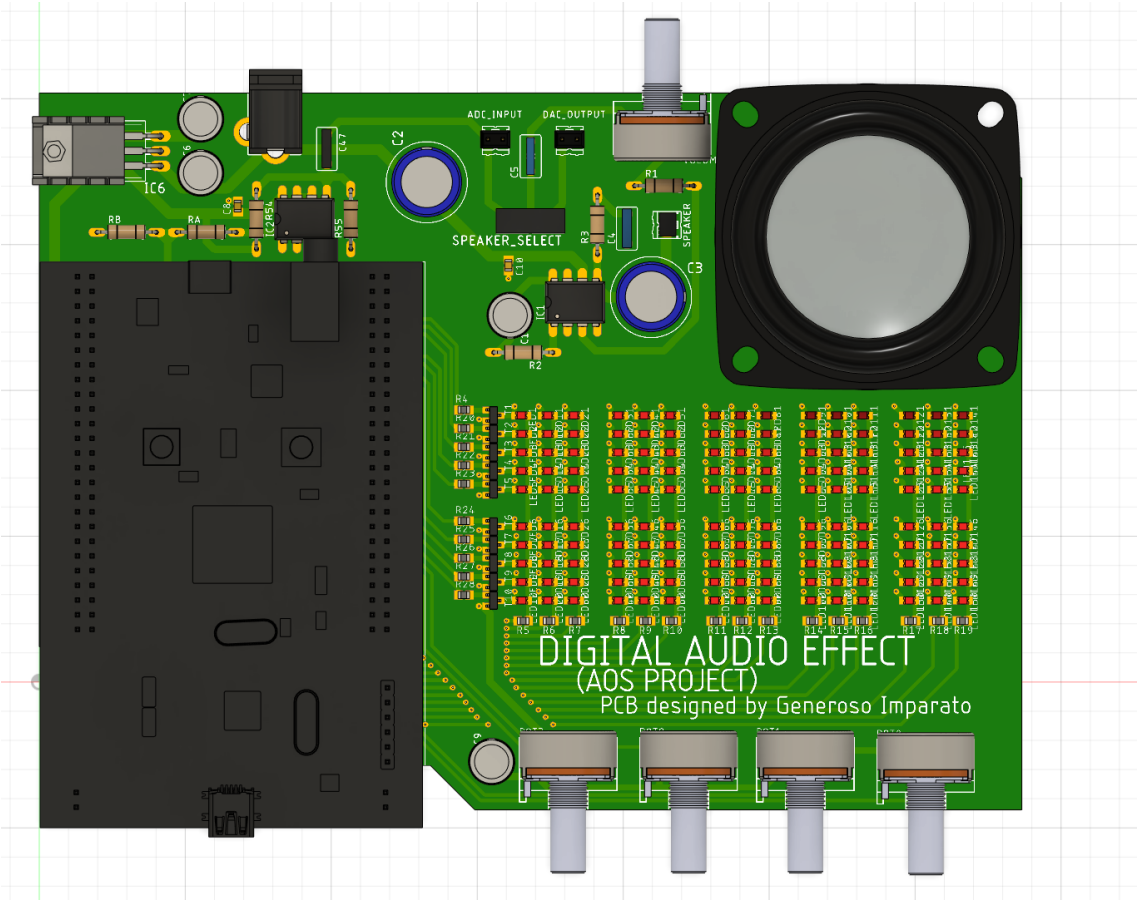
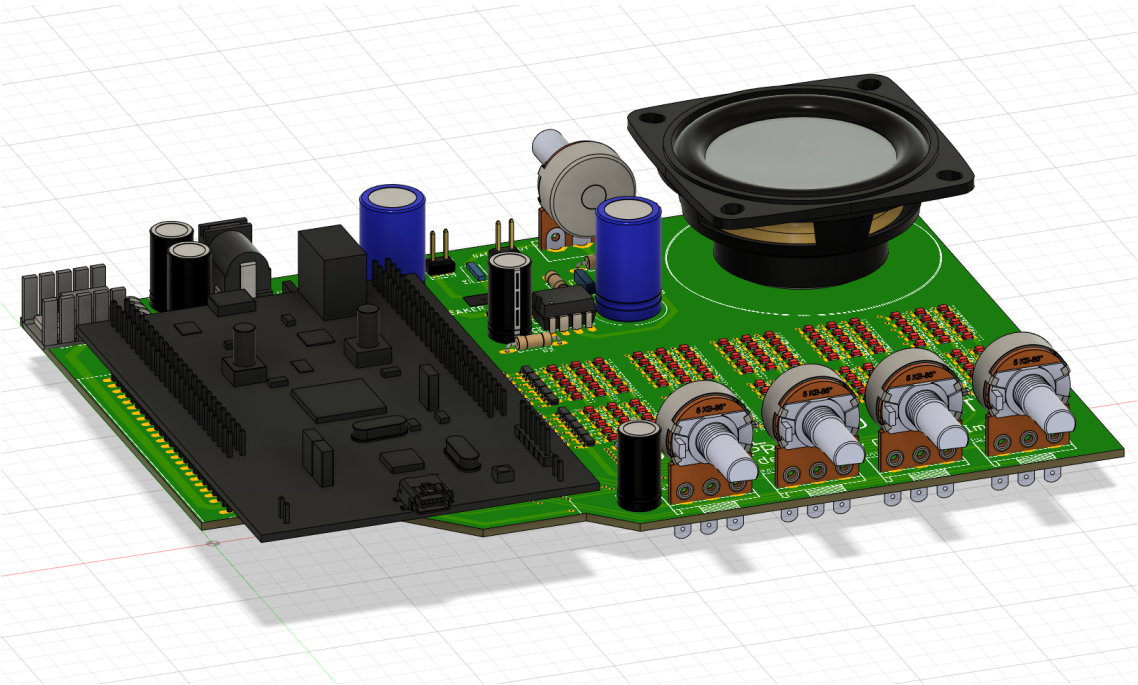


Figure 6: bottom layer

#### 2.1.2.3 Renderings



## 2.2 Software

In this section we describe the real-time OS utilized, the structure of our code in terms of optimizations and scalability, as well as our design choices regarding the management of the various threads and the implementation of different drivers.
Finally, we talk about the digital audio effects and their implementation in detail.

### 2.2.1 Miosix

Miosix is an embedded OS kernel developed by Federico Terraneo designed to run on 32bit microcontrollers, especially on STM32, EFM32 and LPC2000 microcontroller families.

It is used for real-time operating applications and provides operative system abstraction layers such as single process, multi-threading and multi-process (experimental).

There is also support for the standard POSIX thread API (threads, mutexes and condition variables) other than full support for C and C++ standard libraries. The Miosix kernel is licensed under the GPL license.

#### 2.2.1.1 OS Choice

This project heavily relies on multi-threading and real-time management, so this OS was a good fit, since it supports multi-threading natively, as well as synchronization mechanisms.

Another advantage is the small code size, fundamental for limited flash memory as in the case of most microcontrollers.

### 2.2.2 Multi-threading Design


Figure 7: Diagram Legenda

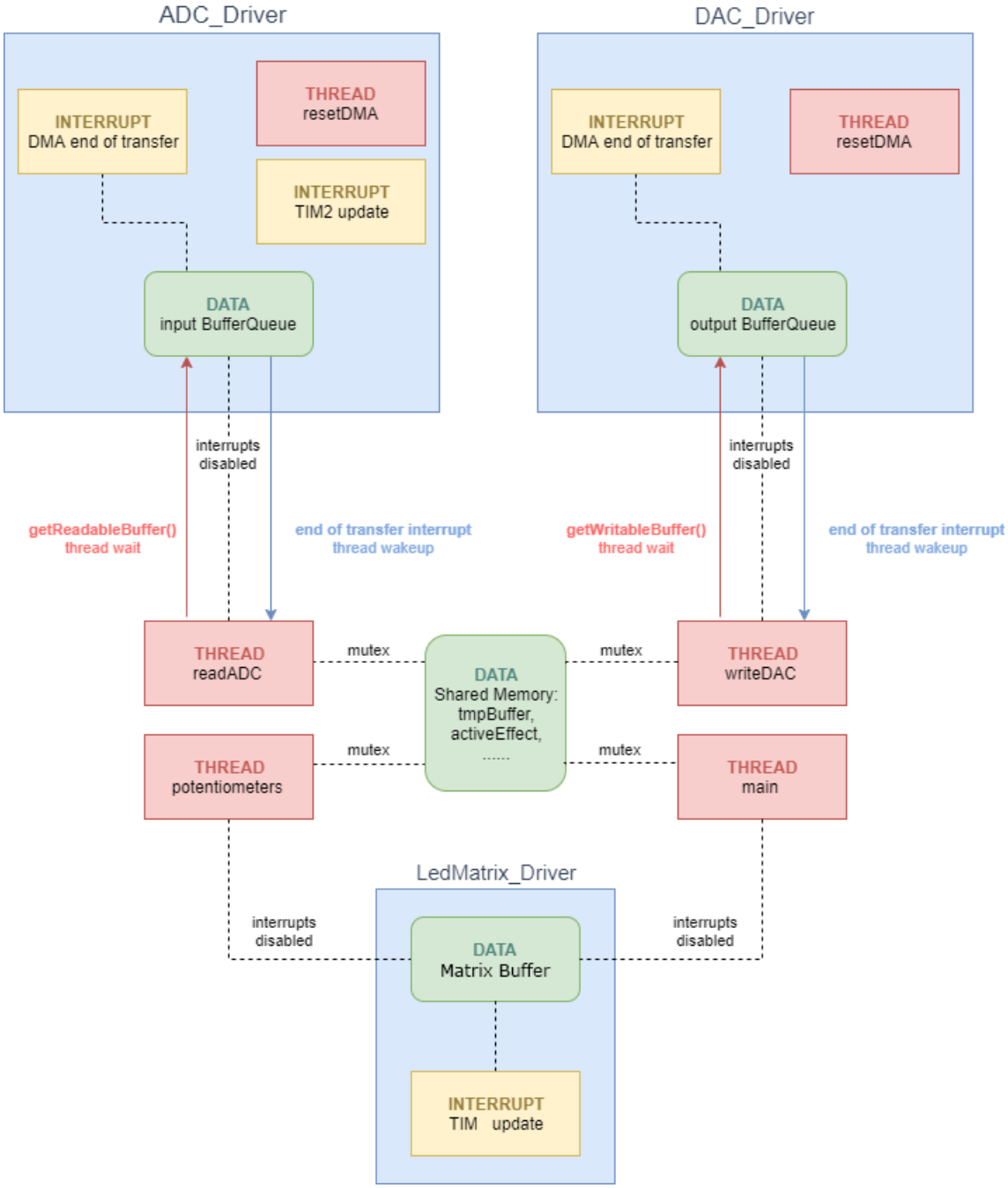



Figure 8: Threads, interrupts and data access relationships

The application is realized with 6 threads. The two most important threads `readADC` and `writeDAC` have the highest priority because they should be responsive to allow the audio stream without lags or skipping buffers.

`readADC` reads from a buffer queue in the `ADC_Driver`, which is filled with audio sampled from ADC at 44,1 kHZ, in a similar way `writeDAC` writes in the buffer queue of the `DAC_Driver`. When `getReadableBuffer()` is called the `readADC` thread is put in a wait state until a buffer is available to be read, when the DMA has finished the transfer.
In a similar way, when `getWritableBuffer()` is called, the `writeDAC` thread is put in wait until a buffer is available to be wrote.

In order for the threads to be synchronized, it is required that the samples in input, and the ones in output are read/wrote at the same frequency, which in this case is 44,1 Khz, because it is a standard sampling frequency for audio.
The input/output buffer queues are accessed by interrupts (DMA) and threads ( `readADC` / `writeDAC` ), so to avoid concurrency problems, interrupts are disabled when accessing from the thread.

The potentiometer and main thread spin at fixed rates and are responsible for reading potentiometers values from the ADC and for checking the button to switch effect/audio source.

The two `resetDMA` threads are responsible to restart the DMAs in case the process stops because no buffer was available to be read/wrote.

### 2.2.3 Drivers

In order to hide the low level handling of peripheral registers and the logic required, we wrote some classes that act as drivers, providing higher level methods to use the ADC, DAC and Led Matrix. The drivers are implemented as static classes.

#### 2.2.3.1 TIMx Configuration

Since drivers use hardware timer interrupts, we will show how the right frequency of the interrupts is obtained.

Starting from the internal clock frequency ($168MHz$), we pass through the AHB Prescaler and the AHB1 Prescaler, so the initial timer frequency is:

$$f_{TIMx} = \frac{f_{clk,int}}{AHB\_PSC \cdot APB1\_PSC} = \frac{168MHz}{4} = 42Mhz$$

Then, the frequency can be further prescaled setting the 16 bit prescaler (TIMx_PSC) and the auto-reload register (TIMx_ARR), according to this formula:

$$f_{event} = \frac{f_{TIMx}}{(TIMx\_PSC + 1) \cdot (TIMx\_ARR + 1)}$$

#### 2.2.3.2 ADC Driver

The peripherals used by the driver are ADC1, ADC2, TIM2, DMA2.

ADC2 is configured in DMA mode, single conversion triggered from the TIM2 update event and the channel is connected to the audio buffer input circuit.

TIM2's update frequency is set according to this formula, which was explained in the *TIMx Configuration* section:

$$f_{update} = \frac{f_{TIM2}}{(TIM2\_PSC + 1) \cdot (TIM2\_ARR + 1)} = \frac{42MHz}{(16 + 1) \cdot (55 + 1)} = 44.1kHz$$

DMA2 is set in peripherals to memory mode to fill `AUDIO_BUFFERS_SIZE` number of samples of the input buffer queue.

The ADC resolution is 12 bits, so the values of the samples are in range from 0 to $2^{12} - 1 = 4095$

The driver acts like a *producer* of audio buffers in the input queue. If the queue is full and a new buffer cannot be added, the DMA stops, so there is a thread that spins in the background to restart the DMA as soon as the queue is free again.

`ADC_Driver` provides access to the reader thread through the following method:

```
const unsigned short* ADC_Driver::getReadableBuffer(miosix::Thread* readerThread)
```

The reader thread is set in wait state if a buffer is not available to be read, it will be woken up later, when the DMA end of transfer interrupt occurs and the buffer is filled again.

The driver has also methods to perform single ADC conversions or multiple conversions with a low pass filter. For these functionalities ADC1 is used:

```
unsigned short ADC_Driver::singleConversion(unsigned short channel);
unsigned short ADC_Driver::filterConversions(unsigned short channel);
```

#### 2.2.3.3 DAC Driver

The DAC used is the CS43L22, which is already mounted on the discovery board.
It operates in I2S mode, for which the SPI3 peripheral is used with the internal audio PLL set to 44,1 kHz.
DMA1 is set in memory to peripheral mode to send `AUDIO_BUFFERS_SIZE` number of samples from the the output buffer queue to the DAC through I2S.
I2S protocol uses 16 bit unsigned PCM samples, so the values are in range from 0 to $2^{16} - 1 = 65535$

The driver acts like a *consumer* of audio buffers in the output queue. If there is no buffer available in the queue, the DMA stops, so there is a thread that spins in the background to restart it as soon as the queue is filled again.

`DAC_Driver` provides access to the writer thread through the following method:

```
unsigned short * DAC_Driver::getWritableBuffer(miosix::Thread* writerThread)
```

The writer thread is set in wait state if a buffer is not available to be wrote, it will be woken up later, when the DMA end of transfer interrupt occurs and the queue is free again.

The DAC output is connected to audio jack on the discovery board, which is then connected to the speaker amplifier on the main PCB with a jack cable.

#### 2.2.3.4 Led Matrix Driver

The Led Matrix serves as the output viewer of this project. To better handle this component, `Led_Matrix_Driver` was implemented.

`Led_Matrix_Driver` provides access to the writer thread through the following method:

```
void LedMatrix_Driver::setString(const std::string &str);
```

This method writes the led values in a buffer according to the string given as input.
To write in the buffer, a `FastInterruptDisableLock` is needed to avoid concurrency problems with respect to the other threads which access this method.

Then, the method

```
void LedMatrix_Driver::writeLeds();
```

lights up the leds based on the buffer's values, *one row at a time*.

As we can see in the *Circuits* section of the PCB, there are PNP transistors only for the rows, so we can light up at most one entire row to avoid damaging the board, since the maximum current output allowed by the board is $100mA$.
Running this on a separate thread was not a good choice, since the other threads affected its behaviour in a visible manner, so we chose to implement an *Interrupt Routine* to achieve a constant refresh rate of the led matrix.
So we used an internal timer (TIM5) to trigger the IRQ. First of all, we configured TIM5's frequency according to this formula, which was explained in the *TIMx Configuration* section:

$$f_{update} = \frac{f_{TIM5}}{(TIM5\_PSC + 1) \cdot (TIM5\_ARR + 1)} = \frac{42MHz}{(349 + 1) \cdot (239 + 1)} = 500Hz$$

This means that each row of leds is refreshed in such a way that we can see all of the rows active at the same time.
Note that, since an Interrupt call refreshes a single row of the led matrix, lowering the frequency value makes the view unstable.

### 2.2.4 Audio Chain

The audio chain is mainly implemented in the `readADC` / `writeDAC` threads.
`readADC` thread copies buffers from the `ADC_Driver` into a shared buffer, which is accessed with mutex protection because it is also used by `writeDAC` thread.
It also converts values from unsigned 12 bit to floating point.
`writeDAC` is the thread that performs the actual DSP on the buffer, before writing it to the `DAC_Driver`.

If the audio source is set to `ADC`, the shared buffer is used as input for the active effect DSP algorithm.
If the audio source is set to `Synthesizer` the shared buffer is not used, because input samples are generated by the thread and then fed into the active DSP algorithm.
Then samples are converted from floating point to unsigned 16 bit PCM and sent to the `DAC_Driver`.
The number of samples in each buffer can be changed by defining `AUDIO_BUFFERS_SIZE`, which is currently set to 512.

### 2.2.5 DSP Effects

The synth and the DSP effects are all implemented by extending a base class `AudioEffect`, that provides virtual methods that can be overridden to create different effects.
In particular the DSP algorithm is executed in the method:

```
virtual void AudioEffect::writeNextBuffer(float* inBuff, float* outBuff);
```

The default implementation just copies `inBuff` into `outBuff`, applying no effect.

This design allows to easily implement different effects, focusing only on the DSP part and avoiding to handle all the audio chain, as well as the controls updating from the potentiometers.

The effects included in this demonstration are a *Delay*, a *Filter* and a *Tremolo*.

#### 2.2.5.1 Delay

The delay effect is obtained thanks to a circular buffer, which is continuously refilled from the beginning once is full.
This buffer has a size that is a multiple of `AUDIO_BUFFERS_SIZE`, and it is used to keep in memory samples from past buffers.
When producing the next buffer, samples from the delay buffer are added to the input, multiplied by a feedback parameter.
The maximum delay time depends on the size of the circular buffer, and can be computed as:

$$T_{max} = \frac{buffer\_size}{f_{sampling}}$$

In this case, the circular buffer size is set to:

$$T_{max} = \frac{k \cdot \texttt{AUDIO\_BUFFERS\_SIZE}}{44.1kHz} = \frac{45 \cdot 512}{44.1kHz} = 522ms$$

The actual delay time, as well as the feedback are left as parameters controllable with potentiometers.

#### 2.2.5.2 Filter

The filter effect implements a simple low-pass, band-pass and high pass filter.
Controllable parameters are cutoff frequency and resonance.
The DSP algorithm implemented is from Paul Kellett and can be found at:
https://www.musicdsp.org/en/latest/Filters/29-resonant-filter.html

#### 2.2.5.3 Tremolo

The tremolo effect applies a modulation on the signal amplitude, which is controlled by a software LFO (low frequency oscillator).
The signal samples are multiplied by a factor which is generated by the LFO and the speed can be controlled as parameter.

#### 2.2.5.4 Synthesizer

We added a wavetable synthesizer to be used as an audio source.
it is implemented by extending the `AudioEffect` class in a way similar to the DSP effects, but since it generates the samples from scratch it does not read from `inBuffer`, but writes to `outBuffer`.

Wavetables are pre-computed to make the real-time computations faster, so when the samples are generated, values from the wavetables are used, with proper stretching according to the desired sound frequency.

The synthesizer has 3 independent oscillators, each one using a different wavetable:

- Sine
- Triangle
- Square Wave

Those waveforms are added together and multiplied by the respective amplitude, in order to generate the output samples.
Each individual oscillator and its parameters can be controlled with potentiometers.

Controllable parameters are:

- Frequency
- Amplitude
- Speed
- Mode

*Speed* parameter controls the speed at which the selected pattern is played.
*Mode* allows to select a pattern for the oscillator, such as continuous note, scale, random notes, octaves and so on.

# 3 Conclusions

This project covered the development of a system in which multiple audio sources could go through different digital audio effects, all in a real-time environment and with an enhanced user experience brought by a custom PCB.

We experienced some noise in the audio sampled from ADC, and we couldn't remove it completely, but overall, we can say that our initial goal was reached and we built a solid base for future improvements.

The `AudioEffect` class is very easy to extend, so adding a new digital audio effect and viewing it in the led matrix, is a very simple task, and this applies also to adding other audio sources.

## 3.1 Future Improvements

We could implement audio effects which require the usage of a FFT to expand the pool of options.

Another possible improvement would be adding an anti-aliasing filter on the audio signal before the ADC sampling, which could maybe solve the noise problem mentioned before.

Also, we could use a cleaner power supply and a better analog/digital power separation on the PCB.

Finally, an improvement to the user experience would be a better handling of controls shown in the led matrix.