THemis: a drum machine and music synthetiser for 21st century music

Alexis Poumeyrol, Bastien Fratta, Corentin Mantion, Hugo Peltzer, Loic Pineau, Lucien Manza, Mathieu Tissandier, Morgan Prioton, Nathan Tricot, Nicolas Ourmet, Roland Giraud, Ronan Martin, Thomas Mazella, Tristan Poul (2A) & Alex Niger, Corentin Colard, Diane Duthoit, Ahmed Wadie Moutawakil (3SyM)

Fall '18

Contents

Part I

Etudiants 2A

Chapter 1

Touchscreen UI and front pane simulator (by Bastien Fratta)

1.1 Hardware

To enhance user interface we add a touch screen to Themis. Thanks to it you'll be able to navigate fast and deeply in this great sound machine to create unique sounds.
It is a 7" capacitive multi-touch screen with a 800x480px resolution pretty comfortable to perform on stage, at the studio or in your bedroom.
On its back you can find a Raspberry Pi 3 which aims to drive the screen and also to communicate with the STM board and the encoders boards via SPI and I2C buses. The whole UI is based (at first)on JavaFX code to propose a fluid and modern navigation through menus. Due to compatibility issues between RPi drivers and Java versions the final UI for the year 2019 has been developed with java.swing API. The first idea in JavaFX still available in the Eclipse project for the future.

1.2 User interface

The screen have various display mode in which any basic or advanced parameters of each module can be easily read/written. The layout of those views has been designed as clean as possible in order to optimize touchscreen surface. In order to enhance and ease the user experience a real ergonomic thinking work need to be done. This will allow to give a clean stylish look to each display mode with a fluid comprehension and user experience. Let's get into it.

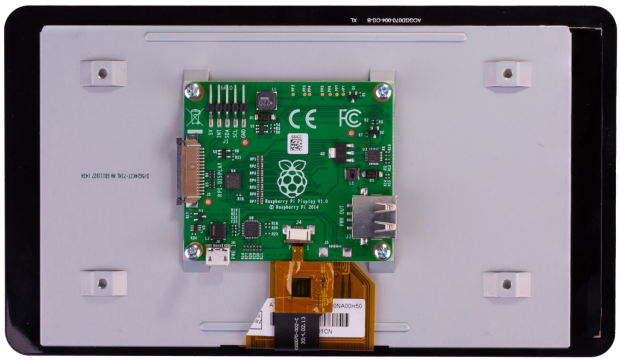


Figure 1.1: Raspberry Pi mounted on touch screen

VCO

Themis gave life to nymphs and her VCOs are among them. Generate precise and rich sound based on mythic (myhtologic ?) sound from vintage chipsets. (see Chapter 9 and 11)
Choose your Octave, adjust the pitch, select a waveform and start building a unique sound. You can also set synchronization, X modulation and ring modulation.

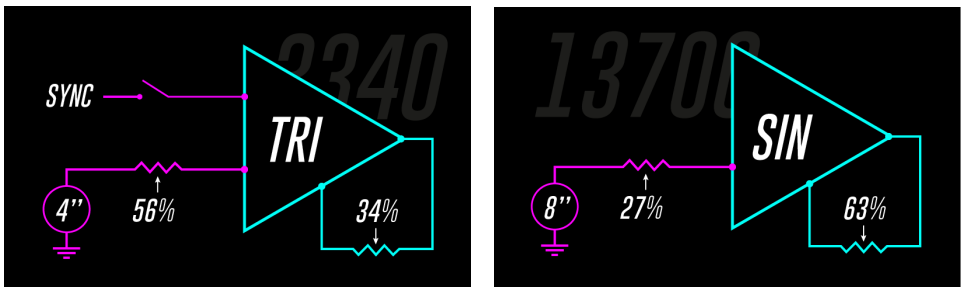


Figure 1.2: VCO geeky views for 3340 and 13700

Envelopes

To add more life to every note you can also shape multiple ADSR envelopes. The times and amplitudes of the 4 points of the each ADSR can be set.(see Chapter 7)

By holding the corresponding shift button you can switch from time to amplitude mode. By hitting shift you can tweak another envelope.

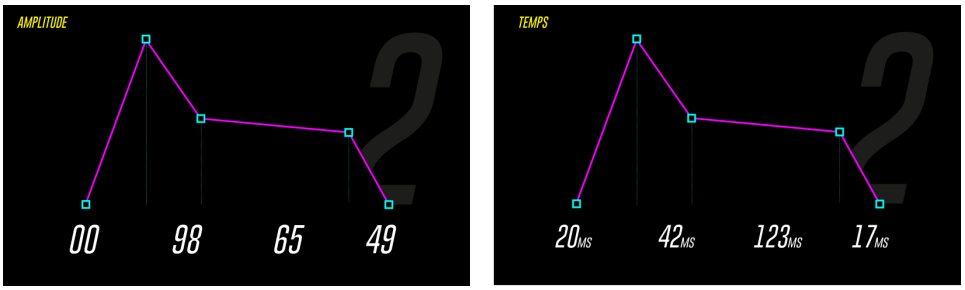


Figure 1.3: Envelope views with amp and time mode

Filter

Themis filter is another powerful musical tool. Change and modulate your filter type to precisely sculpt spectrum. Every classical parameter is here : cutoff, resonance, envelope amount, filter type and order. Like it is a highly expressive module, display of values is also improved for live performance.
Need to be more expressive ? Switch to the XY mode by holding the corresponding shift button.

Figure 1.4: Filter views with normal and XY mode

Mixer

Set the gain of the various audio sources of the Themis. This module need some artisitc choices.

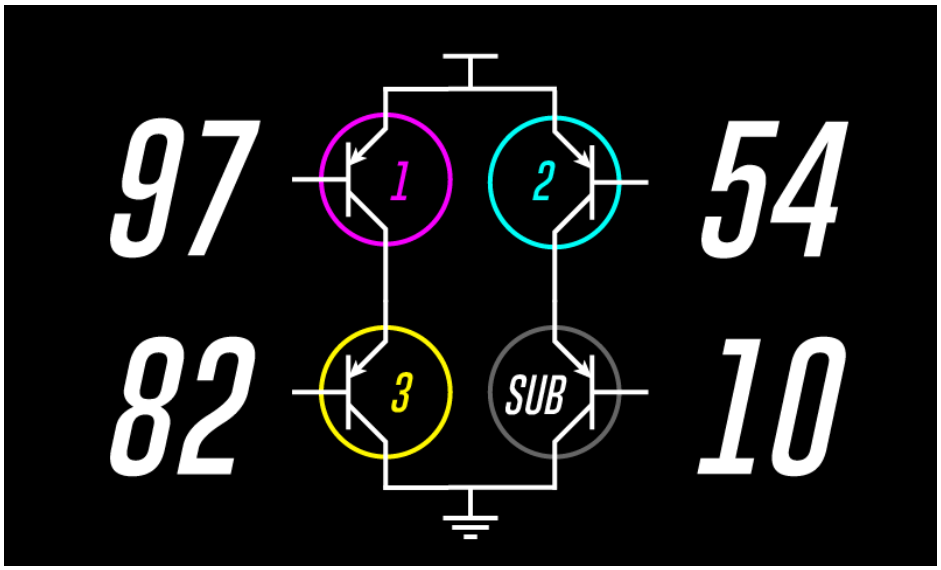


Figure 1.5: Mixer view to mix the 2 VCOs, sub, noise etc...

Matrix

(Not used)
The matrix view is a large view of Themis brain. It's here you can interconnect any modules to each other.
The matrix contains modulated on columns and modulators on lines. By clicking at the intersection between two modules, you easily connect them and start a new creative modulation. By drag and drop out of the corresponding box you can also set the intensity of this modulation. Don't hesitate to deeply explore this wonderful tool to create your own sound.

1.3 Front pane simulator

In order to discuss on technical solutions and layout organization for the front pane, I am also developing a simulator which emulates how front panes works. At the beginning the simulator was a single JavaFX windows but after multiple RPi compatibility issues it is now 3 split windows based on java.swing API. You have to set the global variables in the main depending on how and on what you want the simulator run. If SIMULATOR is false UI will start as if it was on the Themis and will display a non-decorated touchscreen frame at the (0,0) position for instance.

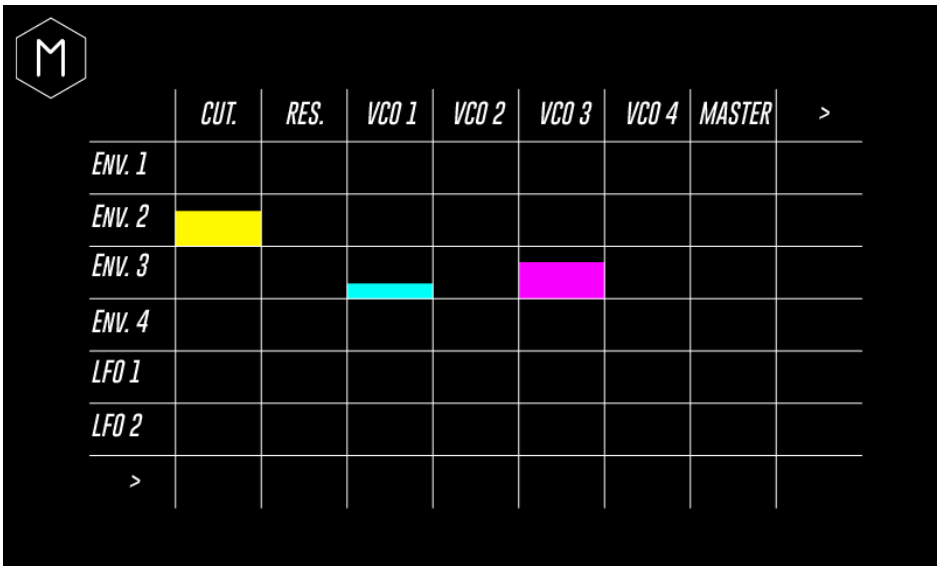


Figure 1.6: Modulation matrix view

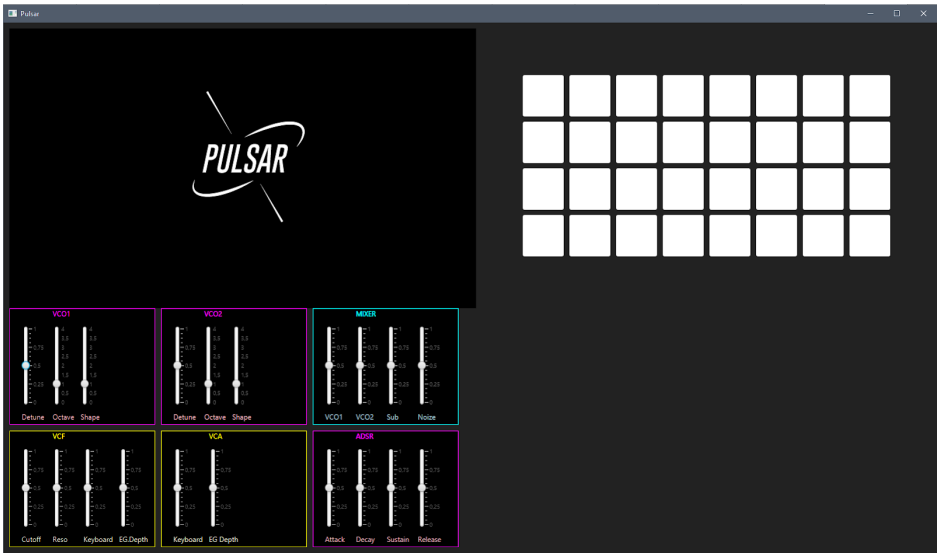


Figure 1.7: Simulator window with JavaFX

1.3.1 The architecture

Like the final UI interface, the simulator is based on the MVC (Model-View-Controller) architecture. At launch the simulator is able to instance every analog

and numerical component of the synthesizer by creating is virtual model. Each parameter with name, type, range and default value is also precised in the corresponding model. For each of them, the adequate control is created and can be displayed in a view. Encoders, touch screen or pads can act as a controller and modify models parameters values. Every changes automatically refresh the corresponding view.
This architecture is widely used as it is a very modular one. As well, it will be easy to add any additional modules in the future. At the end of the start, software architecture is ready to use and able to communicate threw SPI and I2C buses thanks to Loïc work. (see Chapter 5)

1.3.2 Simulator operation

The simulator code is based on the same architecture and classes as the UI implemented in the final product. The layout is also really closed to the front panel of the Themis with 3 sections : touch screen, performance pads and knobs(which is now split in 3 distinct windows).

Touchscreen pane

This part of the simulator is identical to the real touchscreen. You can switch between view thank to the menu bar at the top. It still need to implement touch zones over every parameters values displayed in order to be able to change them directly with a drag move.

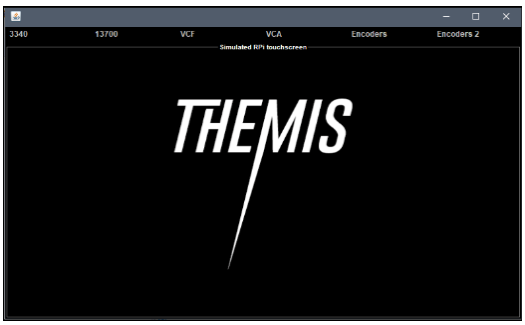


Figure 1.8: Touchscreen pane of the simulator with swing API

Expression Pads pane

Those pads can control the analog drums module (see Chapter ?) and the sequencer (see Chapter 15). Actually there are connected only to trig drum sound

but they will be a great control surface to develop in the future.

Controller pane

This pane has been designed in order to develop the UI without considering the development of hardware controls. It is also a way to check if every changes of a controller are reported to the concerned model thanks to listeners and this actualize the corresponding view. It also displays the way of bargraph should work.

Figure 1.9: Touchscreen pane of the simulator with swing API

All this means that it could be used to control an operational Themis like a configuration tool or a control VST plugin. This simulator is more a great tool to keep developing the UI and to discuss on the synthesizer work flow independently of the advance of the project.

1.3.3 Debugging

When the simulator is running, we've added terminal output in order to check if every message send is working. By doing this we can quickly verify the type of message, where is it sent, is new and old value etc... Simulator is also a good way to process unit tests of the UI. By doing those tests we will guarantee a more stable and fast running to ensure the most comfortable use of Themis. Those tests haven't been done this year but it will be an important part at the real end of the project.

Chapter 2

Bargraphs and displays (by Lucien Manza-Capitao)

2.1 Presentation

2.1.1 Goal and context

Bargraphs will be use for give a visual of Themis activities, sound level, modulation used directly on the front of the synth. In addition to the touch screen, we will be able to watch without a problem important information without using the touch screen. The LED are led by the rotaries encoders and by the touch screen in function of modifications brought by modulators. As we can see, Bargraphs are controlled by two different systems, which involved a way to address the issue, in addition to the high number of LED that we will have to control in the same time.

2.1.2 Hardware

To control the LEDs, we will use the RaspberryPi 3 microprocessor which allows to implements code and use GPIO pin. In our case, we will use pin 2 and pin 5 because they will be required for use I2C protocol. The IC bus will allows us to send tasks to the driver IS31FL3731, driver which control a large amount of LED.

2.2 The I2C protocol

See the part for the I2C protocol at the section 4.2

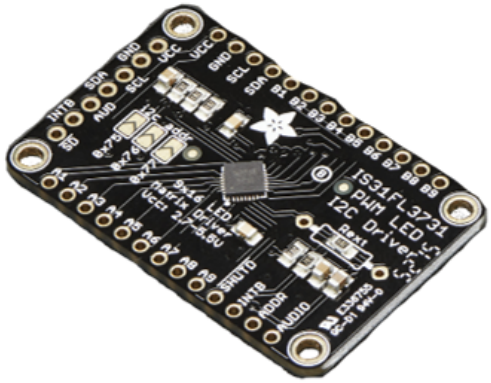


Figure 2.1: Adafruit IS31FL3731 driver

2.3 The IS31FL3731 matrix led driver

The IS31FL3731 driver have 18 current input/output ports to communicate with the LED. To manipulate LEDs, the driver assumes two LEDs matrix. Indeed, each matrix has 72 LEDs spread into 9 rows and 8 columns. Each cathodes and anodes are connected to current input/output port.

2.3.1 LED control

The IS31FL3731 driver have 18 current input/output to communicate with the LED. To manipulate LEDs, the driver assumes two LEDs matrix. Indeed, each matrix has 72 LEDs spread into 9 rows and 8 columns. Each cathode and anode are connected to the current input/output port.

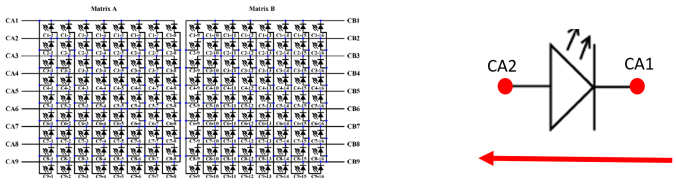


Figure 2.2: Pattern of the two Matrix LED

If we take for example the LED at the position (1,1), he is link with ports CA1 at the cathode and CA2 at the anode. To turn on the LED's light, we have to get a positive voltage, we have the tension at CA1 is lower than CA2. Nevertheless, we have to controle more than one LED: the CAx port won't act at the same time.

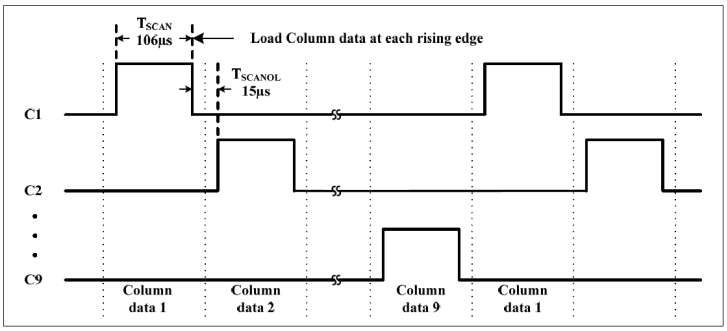


Figure 2.3: Diagram of LED timming

2.3.2 IS31FL3731 Frames Register

To manipulate the LED's behavior, the IS31FL3731 driver writes on frames which are named "Frame Register". They are the eight frames of the nine frames that the driver include. The last frame will be explained after. As explained, each component have a unique address for IC communication. Here, we can consider that the driver have two addresses instead of only one: 0x74 to write on the frames and 0x75 to read on the frames. In fact, the device has the address "0x74" and the less significant bit is 0 for write or 1 to read.

Bit	A7:A3	A2:A1	A0
Value	11101	AD	0/1

AD connected to GND, AD=00;
AD connected to VCC, AD=11;
AD connected to SCL, AD=01;
AD connected to SDA, AD=10;

Figure 2.4: Slave Address Table

Each frame registers have three particular register.

-The first register is “LED Control Register”. These registers are located at the address 0x00 to 0x11 of the frame registers. It is used to switch on or switch off a LED. As we can see, we have 18 addresses, by order, each address correspond to a row of LEDs (0x00 for A1, 0x01 for B1,... 0x10 for A9 and 0x11 for B9). To select a column, we have to write on the register. The data that we will save on a register will correspond to a column: the first column is the less significant bit and the last column is the most significant bit.

-The second register is “Blink Control Register”. These registers are located at the address 0x12 to 0x23 of the frames registers. It is used to enable or disable the blinking mode for a selected LED. The selection method is the same as the “LED Control Register”.

-The third register is “PWM Register”. These registers are located at the address 0x24 to 0xB3 of the frame register. It is used to regulate intensity of the LED. Now, we have 144 addresses, and each address correspond to one unique LED. On each register, we can write on a 8 bits data: then we have 256 value of intensity for one LED, 0x00 is for the lower lever, and 0xFF is for higher level.

2.3.3 IS31FL3731 Function Register

Chapter 3

Quadrature encoders and switches (by Ronan Martin)

3.1 Rotary encoder

3.1.1 General view

Quadrature encoders are the main components of the synthetizer interface. They allow the artist to manipulate a large amount of different parameters in a visual way, precisely and quickly.

Here you can see what a quadrature encoder looks like (figure 3.1).

More precisely, an encoder is an electromecanical device able to measure a motion or a position. Contrary to the potentiometer, an encoder has an axis whiches is not limited at left and at right, you can turn it from the left to the right or from the right to the left because there is no stop.

Figure 3.1: Rotary Encoder.

3.1.2 How they work

Rotary encoder has 3 pins (A signal, B signal and a ground). The first two pins are either linked or not linked to the ground thanks to two interruptors. When you turn the encoder, there is contact or not contact between the ground and one of the two pins of signal. The pin is either linked to the ground or floating. The common pin can be the ground but in our case it is a pull-up resistor. linked to the two pins. The pin is at the alimentation level (5V for example) when the encoder is not in contact and at ground level (0V) when there is contact. As usual, we can decide that the zero logic is for 0V and the one logic is for 5V.

The phase difference between signal A and B, which are phase-shifted by 90 degrees, that allows evaluation of the direction of rotation. They are in quadrature. When you turn the rotary encoder, signal "A" and signal "B" will both switch from 0V to 5V and inversely but as it exists a phase-shift, if we turn the rotary in one direction, it firstly the signal "B" that switch from 0V to 5V and if we turn in the other direction, it is the signal "A" that first switch to 0V. The electronic system interpret that like turn to the left or turn to the right in order to increase any digital parameters.

To better understand this phenomenon, this scheme resume the different signals and the digital interpretation of a encoder-right-turn or a encoder-left-turn.

If we read in the order the logic values of the signal "A" then of the signal "B", we form a binary word of 2 characters. When the encoder stays static, nothing happens in a electric point of view. If we turn to the right, the binary sequence will be successively : AB = 11, then AB = 10, then AB = 00 and finally AB = 01. We come back to the begin after. The system interprets this sequence as "increase by 1" the previous sequence and inversely for "decrease by 1".

3.1.3 Implementation of encoders in the project

The goal of this project is to code the fonctionalities of an encoder in Java. When a parameter is selected on the synthetizer, we will be able to adjust precisely this parameter via a known values list that adapts to the selected parameter (VCO,VCF,envelops...). To guarantee a sufficient number of encoder for our instrument, it has necessitated to find a method for associated them together and make them communicate. This method is the I2C communication thanks to the MCP23017 GPIO expander.

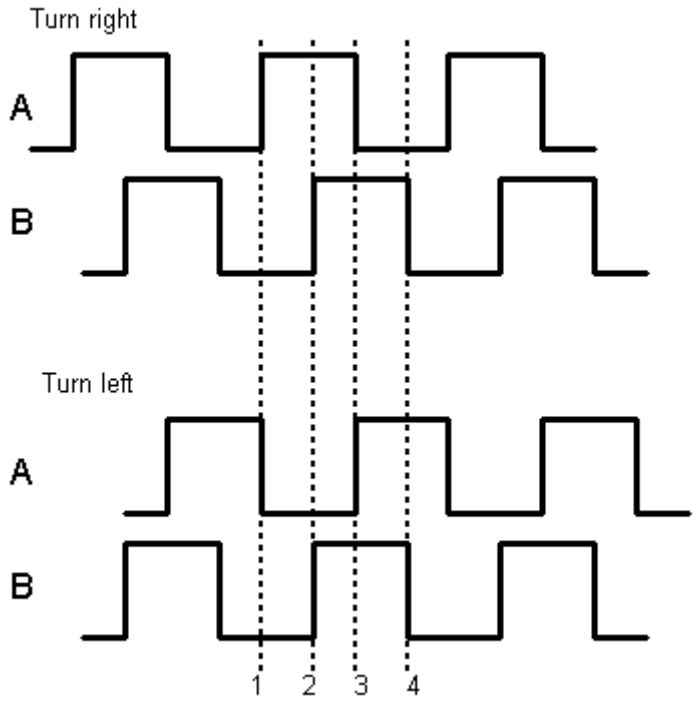


Figure 3.2: Signals.

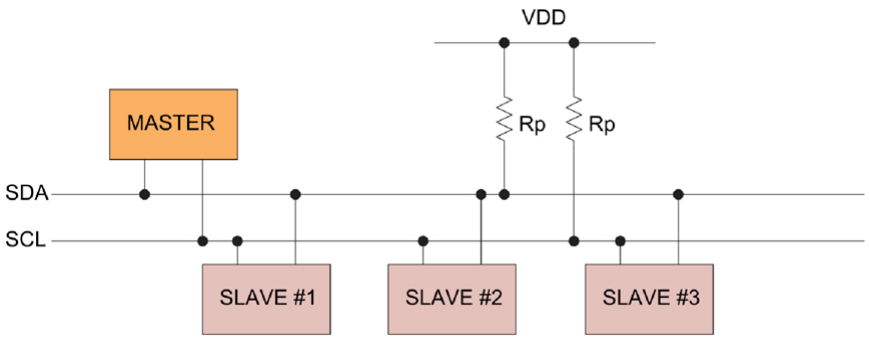


Figure 3.3: I2C.

3.2 I2C communication

On a I2C bus, all the complex information run on the serial bus by 8 bits group. I2C communication only needs three wires, a ground, a SCL wire (Serial clock line) that rythms the signal and a bidirectionnal line of data SDA (Serial data line). Each time the SCL line is on the high state, one bit is available on SDA. When a component controls the I2C bus, he is the master for example in our case it's the raspberry Pi. It observes, after the sending of each octet, the presence of a acquittment signal. This signal is produced by a slave that confirms it has received the whole octet but a slave never takes initiatives, it only answers master's claims.

Each I2C component has a unique address. When a master send a message, it begins by trasmit the component's address which it wants to call then the transmission direction (writing or reading). In the reading case, a slave stops to send data when the master has not send the acquittment signal. A trame begins by a starting condition and ends with a stop condition. A master can repeats many trames by sending again and again starting conditions one by one after the first trame. It is also possible to connect on the same I2C bus others I2C master components in addition with I2C slaves components. You can see the scheme below :

3.3 MCP23017

3.3.1 Presentation of the bus expander

MCP23017 (figure 3.4) uses 2 I2C pins and, in exchange, it give us the possibility to use 16 more pins of general utilization. The integrated circuits on a I2C bus have one address each. It allows to connect many circuits on the same bus and to communicate to a particular circuit with the use of its address. It is impossible to have two different circuits with the same address on the bus. If we want to put the

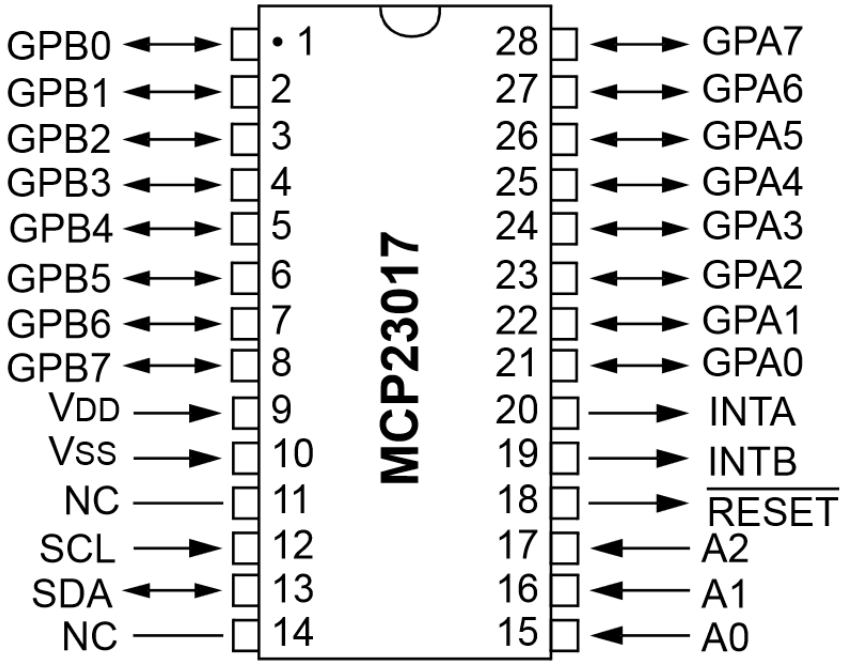


Figure 3.4: MCP23017.

same circuit many times on a I2C bus we need to be able to change the address thanks to the three pins A0, A1 and A2. They allow us to modify the default address of the component on the I2C bus. There are 8 different adresses so we can link 8 MCP23017 that gives us 128 additional pins in total. If we go back to what we are looking for, we could link 64 rotary encoders.

3.3.2 Connexion between encoders on the bus expander

In order to properly connect the encoder to the bus expander, we need to add a debounce circuit between the encoder and the MCP23017. It is important to debounce our circuit because when you turn the encoder, the transition between a low state and a high state is never perfect, there are some bounces, like if you turn very quickly your button, that could interfere the measure.

On each MCP23017, we can add two types of encoders: four encoders with push button and two encoders without push button.

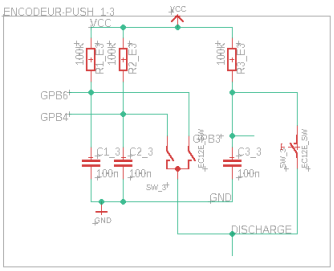


Figure 3.5: Encoder and debounce circuit

TABLE 1-6: CONTROL REGISTER SUMMARY (IOCON.BANK = 0)

Register Name	Address (hex)	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	POR/RST value
IODIRA	00	IO7	IO6	IO5	IO4	IO3	IO2	IO1	IO0	1111 1111
IODIRB	01	IO7	IO6	IO5	IO4	IO3	IO2	IO1	IO0	1111 1111
IPOLA	02	IP7	IP6	IP5	IP4	IP3	IP2	IP1	IP0	0000 0000
IPOLB	03	IP7	IP6	IP5	IP4	IP3	IP2	IP1	IP0	0000 0000
GPINTENA	04	GPINT7	GPINT6	GPINT5	GPINT4	GPINT3	GPINT2	GPINT1	GPINT0	0000 0000
GPINTENB	05	GPINT7	GPINT6	GPINT5	GPINT4	GPINT3	GPINT2	GPINT1	GPINT0	0000 0000
DEFVALA	06	DEF7	DEF6	DEF5	DEF4	DEF3	DEF2	DEF1	DEF0	0000 0000
DEFVALB	07	DEF7	DEF6	DEF5	DEF4	DEF3	DEF2	DEF1	DEF0	0000 0000
INTCONA	08	IOC7	IOC6	IOC5	IOC4	IOC3	IOC2	IOC1	IOC0	0000 0000
INTCONB	09	IOC7	IOC6	IOC5	IOC4	IOC3	IOC2	IOC1	IOC0	0000 0000
IOCON	0A	BANK	MIRROR	SEQOP	DISSLW	HAEN	ODR	INTPOL	—	0000 0000
IOCON	0B	BANK	MIRROR	SEQOP	DISSLW	HAEN	ODR	INTPOL	—	0000 0000
GPPUA	0C	PU7	PU6	PU5	PU4	PU3	PU2	PU1	PU0	0000 0000
GPPUB	0D	PU7	PU6	PU5	PU4	PU3	PU2	PU1	PU0	0000 0000
INTFA	0E	INT7	INT6	INT5	INT4	INT3	INT2	INT1	INT0	0000 0000
INTFB	0F	INT7	INT6	INT5	INT4	INT3	INT2	INT1	INT0	0000 0000
INTCAPA	10	ICP7	ICP6	ICP5	ICP4	ICP3	ICP2	ICP1	ICP0	0000 0000
INTCAPB	11	ICP7	ICP6	ICP5	ICP4	ICP3	ICP2	ICP1	ICP0	0000 0000
GPIOA	12	GP7	GP6	GP5	GP4	GP3	GP2	GP1	GP0	0000 0000
GPIOB	13	GP7	GP6	GP5	GP4	GP3	GP2	GP1	GP0	0000 0000
OLATA	14	OL7	OL6	OL5	OL4	OL3	OL2	OL1	OL0	0000 0000
OLATB	15	OL7	OL6	OL5	OL4	OL3	OL2	OL1	OL0	0000 0000

Figure 3.6: Registers of the MCP23017

3.3.3 How to pilot an encoder with the MCP23017 registers

After the connexion of the three pins of the encoder and the addition of the debounce circuit, you need to pilot your encoder. First, you need to set the different registers of the MCP23017 at precise values. Here are the registers:

To activate the interruptions that will pilot our encoders, each register except IOPOLA, IOPOLB, the two IOCON have to be set to the value 0xFF. This will activate in particular the interruption register (0x0E) that is the main important register. In fact, when you set this register at 11111111, it sets the default value of the signal A and B of your encoder at ”HIGH”. Thus, when you turn your encoder, the falling-edge front while activate the interruption and you can use this to know if you turned your encoder. The flag register will have a value of 64 (01000000) or 128 (10000000) it depends if you turned the encoder on the right or on the left. To clear the interruption, the program just have to call the flag register to erase the interruption.

Chapter 4

SPI bus communication between the Raspberry and the STM32 boards (by Loïc Pineau and Hugo Peltzer)

4.1 General view and context

The digital part of the THEMIS work with a Raspberry Pi 3 and a STM32 board. The Raspberry manages the man machine interface and the STM32 controls the analog board
The user control the synthesizer with the different buttons and the touch screen. The inputs of the user are handled on the Rapberry Pi but the digital signals controlling the VCO are generated by the STM32. We must therefore create a communication between the two boards.

The Serial Peripheral Interface is a synchronous serial communication interface specification. The devices can communicate in full duplex mode, using a master-slave architecture with a single master. The master device originates the frames for reading and writing. Multiple slave devices can be supported with individual Slave Select lines.

The SPI bus specifies four logic signals :

- SCLK, Serial Clock (output from master)
- MOSI, Master Output Slave Input
- MISO, Master Input Slave Output
- CS, Chip Select

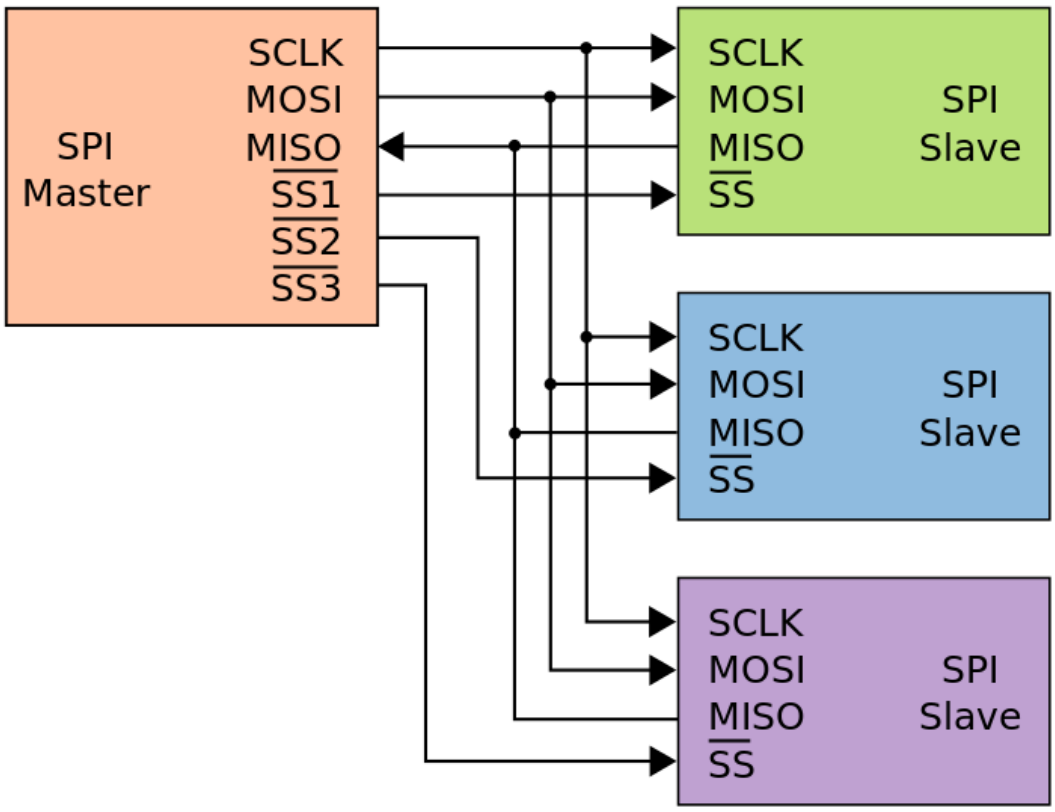


Figure 4.1: SPI with multiple slaves

During each SPI clock cycle, a full duplex data transmission occurs. The master sends a bit on the MOSI line and the slave reads it, while the slave sends a bit on the MISO line and the master reads it.

4.2 Implementation in our project

We don't need to have a feedback from the STM32 so we need to use only three signals : SCLK, MOSI and SS. Using a slave select with only one slave might seem inappropriate but in our way we use this signal in order to know when the master is speaking and not to whom.

Since we use three signals we have to use four pins of the Raspberry.This pins are the following :

Pin 19 : MOSI

Pin 23 : SCLK

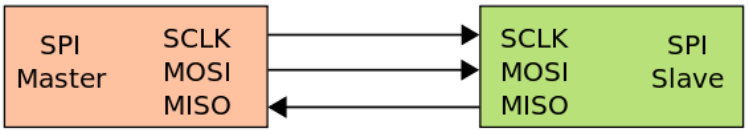


Figure 4.2: SPI with a single slave

Pin 24 : CS

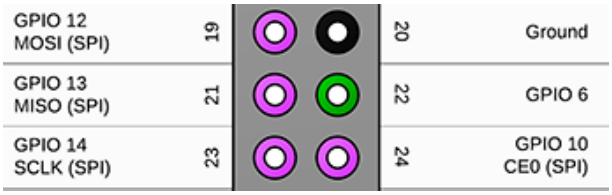


Figure 4.3: Pin numbering

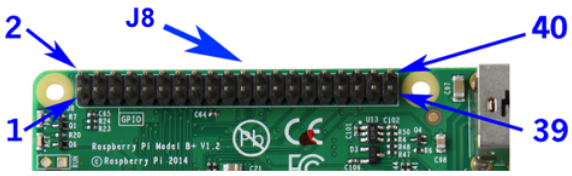


Figure 4.4: Pin numbering

You can find this schematic at `https://pi4j.com/1.2/`. The installation of PI4J is needed on the Raspberry Pi to use the pins with the Java code. You have two options to install PI4J on your Raspberry :

If your Raspberry is connected to the Internet : execute the following command directly on your Raspberry Pi curl -sSL https://pi4j.com/install — sudo bash

Else : download a copy of the latest Pi4J Debian/Raspian installer package (.deb) file to your local computer. You can download the Pi4J Debian/Raspian installer package (.deb) using your web browser at the following URL: `https://pi4j.com/download/pi4j-1.2.deb`

Next, you will need to transfer the download installer package over to your Raspberry Pi

4.3 Communication protocol

We need a communication protocol with the STM board to exchange understandable messages. We choose to base our protocol on the MIDI protocol. It's also important that the STM board can understand a MIDI message. Lots of digital music instrument uses MIDI, it is important to be compatible with this standard in order to use the Themis with other instruments.You can find the complete documentation about the MIDI standard at `https://www.midi.org/specifications-old/category/reference-tables`.

A MIDI message is made of three bytes. One status byte and two data bytes. A status byte start with a one and a data byte with a zero. So you have 127 different combinations for each byte. The status byte define what kind of message is it and the data bytes and the data bytes carry the information.

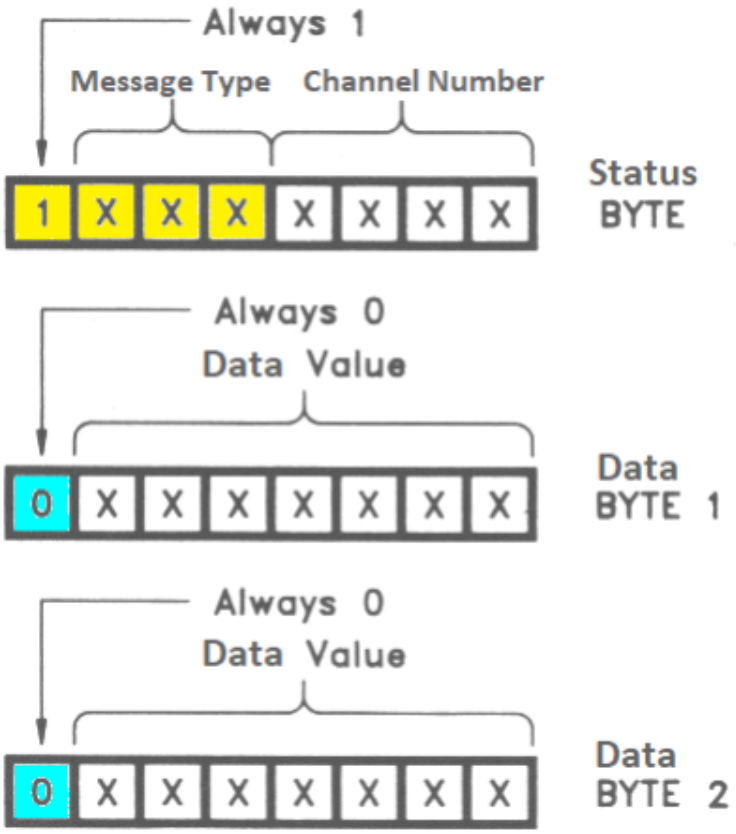


Figure 4.5: MIDI message

The first bit of a byte is always reserved to indicate what kind of byte it is. In case of a status byte the three following bits indicate the type of the message and the four last bits correspond to which channel you want to send the message. The data bytes have different meanings depending on the type of message. It is often a note number the velocity of the note.

Status byte	Data bytes	Description
1000cccc	0nnnnnnn 0vvvvvvvv	Note-off
1001cccc	0nnnnnnn 0vvvvvvvv	Note-on
1010cccc	0nnnnnnn 0vvvvvvvv	Polyphonic aftertouch
1011cccc	0ppppppp 0vvvvvvvv	Control change
1100cccc	0ppppppp	Program change
1101cccc	0vvvvvvv	Channel aftertouch
1110cccc	0bbbbbbb 0hhhhhhh	Pitch bend
1111xxxx	* * *	System

We need to forward the MIDI message to the STM when we receive one as an input (example Note-on or Note-off from a keyboard). But when the user modify a parameter we have to create one. In this case we have to transmit to the STM which parameter have been edited and the new value. We use a Control change with a specific code. The first byte is a Control change status byte the second byte is the parameter ID and the third is the value.

Parameter	Parameter ID
VCO 3340 Detune	0000 0000
VCO 3340 Octave	0000 0001
VCO 3340 Sync	0000 0010
VCO 3340 Wave Shape	0000 0011
VCO 3340 Duty	0000 0100
VCO 13700 Detune	0000 0101
VCO 13700 Octave	0000 0110
VCO 13700 Wave Shape	0000 0111
VCF 3320 Cut Off	0000 1000
VCF 3320 Resonance	0000 1001
VCF 3320 Filter Order	0000 1010
VCF 3320 Keyboard Tracking	0000 1011
VCF 3320 Enveloppe Generator Depth	0000 1100
VCF 3320 Velocity Sensitivity	0000 1101
VCF 3320 ADSR Attack	0000 1110
VCF 3320 ADSR Decay	0000 1111
VCF 3320 ADSR Sustain	0001 0000
VCF 3320 ADSR Release	0001 0001
VCA Enveloppe Generator Depth	0001 0010
VCA Velocity Tracking	0001 0011
VCA ADSR Attack	0001 0100
VCA ADSR Decay	0001 0101
VCA ADSR Sustain	0001 0110
VCA ADSR Release	0001 0111

The value is between 0 and 127 and its meaning depend on the paramecoloc521cter. It can be a boolean (0 for false 127 for true) a numerical value or a position in an enumeration table. You can find the definition of the different parameters in the Java code in the folder *themis/RPi/src/model*.

4.4 SpiTransmitter.java

You can find the class *SpiTransmitter* in the folder *themis/RPi/src/model/spi*. *SpiTransmitter* contains all what you need to communicate with the SPI. There is a constructor for the class initializing a SPI at the frequency of 500 KHz. To create the MIDI messages we can handcraft them by creating three bytes with the adequate meaning. We can also use the class *javax.sound.midi.ShortMessage* which also contains useful constants for MIDI message (all the status byte have

a constant). The method *transmitMidiMessage* that send the message accept as argument a *ShortMessage* or three *short* representing the three bytes of a MIDI message.

The last thing we need is to send directly a message when a parameter is modified. That is the aim of the method*synthParameterEdited*. Thanks to this method the SPI is a listener of the model and each time a parameter in the previous table is modified a message is send containing the parameter ID and the new value of the parameter.

Chapter 5

Analog switches (by Nicolas Ourmet)

5.1 Description

The V411 is a component used as 4 switch. The V411 series of monolithic quad analog switches was designed to provide high speed, low error switching of precision analog signals. Widely used in precision data acquisition and communication systems. Combining low power with high speed.
It is mostly 4 MOSFET controlled in complementary logic.

5.2 Why a switch?

That's the architecture of the VCO and Mixer-part,2 inputs for the VCO LM13700,one for the Subbass-module and the last one for the 3340.

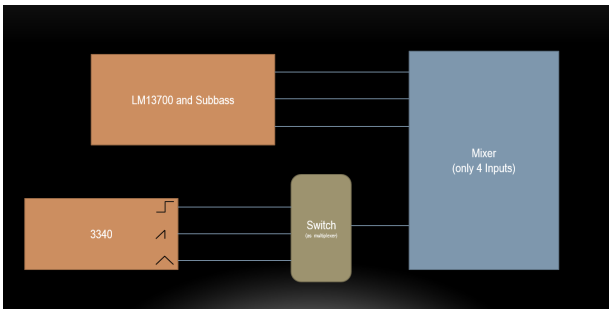


Figure 5.1: architecture

However the 3340 has 3 outputs but only 1 input available to him,that's why the switch is here, and is mainly use as a multiplexer.

5.3 Detailled description

This component is supplied by a symmetric tension ± 15V and a logic tension of +5V.
It is controlled in complementary logic,on with a low state: [0V;0.8V] , off with a high-state: [2.4V;5V] No distorsion are significantly visible on the audible frequency range: [22Hz;22kHz],plus combined with the non-linear effect of transistors, the sound comes out slightly enriched.
The box consists in 4 trio of pins (and 4 supply pins): Switch input, use to bypass or not the signal; the Switch output; and the Logical Input used to control each switch.

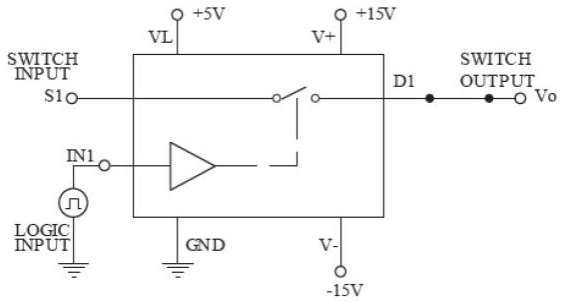


Figure 5.2: architecture

5.4 Bias circuit

Most of the work consisted in sizing the bias circuit. The circuit is relatively simple but required bit of research: High-pass filter R-C to attack the switch with zero-offset signal to avoid discontinuities and ”cracking” sounds when switching signals.
Link capacitor both at the input and output, resistors pin-to-ground to enable the capacitor to discharge and maintain offset at zero.
The Cutoff frequency is equal to 10Hz,like that at 20Hz and more, attenuation is minimum.

Cutoff frequency from a 1st order high-pass RC filter:

$$R_{filter} = R_{gnd}//(R_{Don} + R_{out}) = 5k\Omega \tag{5.1}$$

we got:

$$\omega_c = \frac{1}{R_{filter} * C_{link}} \Leftrightarrow f_{cutoff} = \frac{\omega_c}{2\pi} = \frac{1}{2\pi * R_{filter}C_{link}} \tag{5.2}$$

with:

R_{filtre}	$C_{liaison}$
$5k\Omega$	$3F$

we arrive to : $f_{cutoff} = 10Hz$

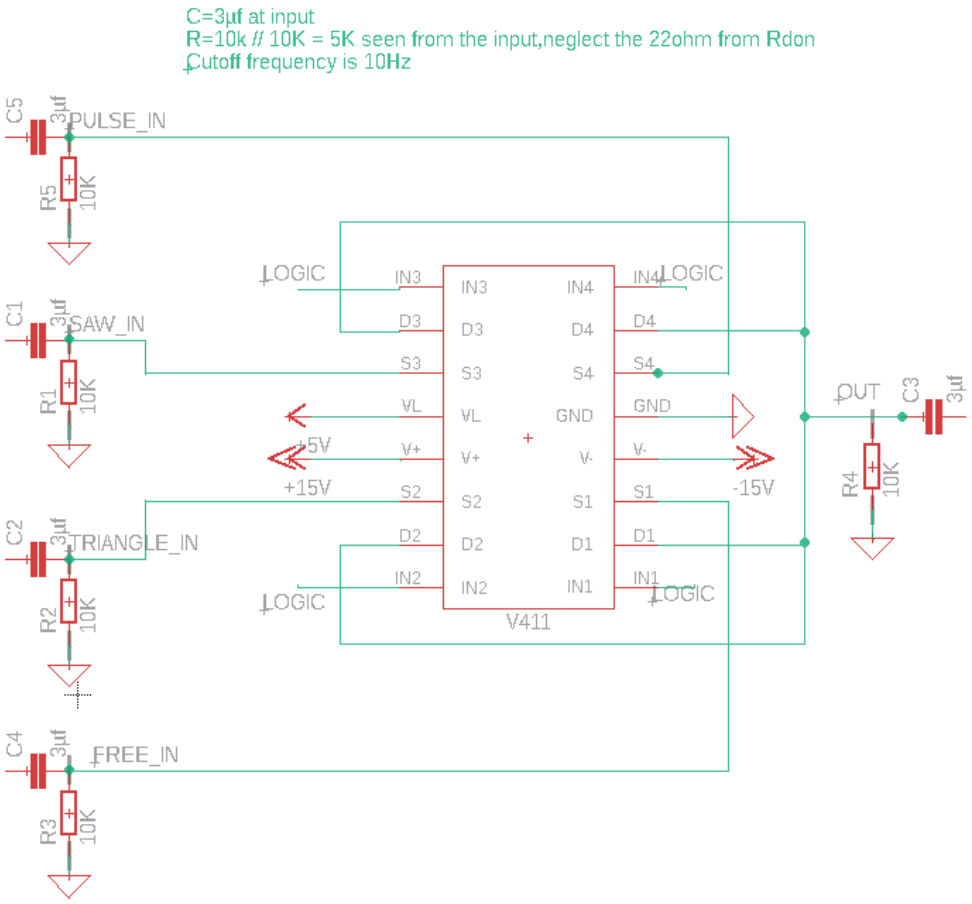


Figure 5.3: V411 Package

5.5 V411 as Multiplexer

Because of 3 inputs and only one output available at a time, V411 is used as multiplexer. The 4 logical inputs are equivalent to the selection bits on a multi-

plexer. The logical inputs will be connected to the I_2C Bus Expander MCP23017 controlled by the STM32.

5.6 MCP23017 to control the Switch

The I_2C Bus Expander is here to control the V411 from the STM32. (general description of the component made at part 4.3.1 by Ronan MARTIN). The component sends a signal a_i on each logical input (4 in total, where $\sum_{i=1}^{4} a_i = 3$ due to the complementary logic command) to have only one passing signal at a time. For this purpose the function" HAL-I2C-Master-Transmit" will mainly be used in this case to unload a control buffer through the MCP and thus control the V411 as a multiplexer.

Chapter 6

VCO calibration algorithm (by Morgan Prioton)

6.1 Context

6.1.1 Interest

The Voltage Controlled Oscillator (VCO) is a device whose output frequency depends on a control voltage (the so-called ”CV” on vintage synths frontpanes!). This device (associated with others like a Voltage Controlled Filter — VCF — and a Voltage Controlled Amplifier — VCA —) can be used to create rich musical signals. This device however — as many analog devices — does experience temperature and humidity drifts and as such, requires precise calibrating if we want to play musically accurate notes !

6.1.2 Components

Three main components intervene in VCO’s calibration : the VCO, the associated DAC and the STM32-NUCLEO-F767ZI. Main links between these components are shown just below :

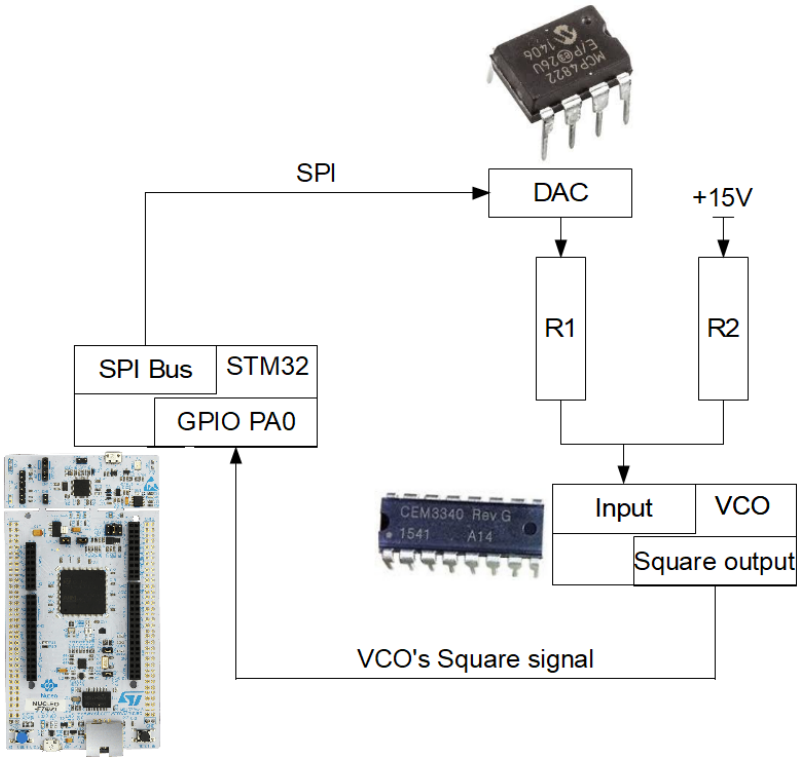


Figure 6.1: Relations between components intervening into VCO’s calibration

The calibration task is centered on C code development on the STM32. One SPI Bus and one GPIO port (GPIO PA0) are used to make that task, more one internal timer of the micro-controller. The SPI Bus is used to make the input control voltage of the VCO by the DAC. Then, the VCO's square signal feed the GPIO PA0 of the STM32. Also, the STM32 analyze the port PA0 in order to detect rising edges and then compute the frequency of the square signal.

6.2 Rising edges detection

6.2.1 Principle

As it is said in the paragraph just above, the VCO's calibration process analyze the VCO's square output signal in order to determine its operation frequency. The principle of rising edge detection on a square signal is to detect a voltage level change from low level to high level. Because of the periodic character of the square signal, it is very simple to know the frequency f of that signal if we know the time T between two rising edges : $f = \frac{1}{T}$.

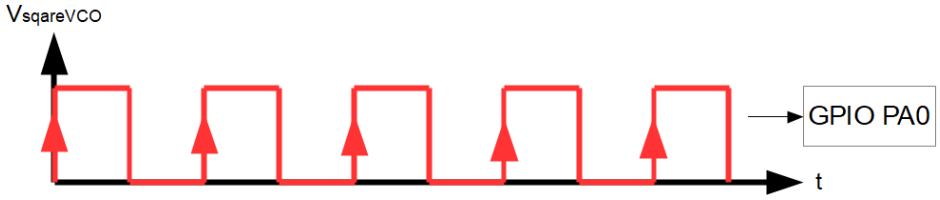


Figure 6.2: VCO's square output signal

6.2.2 Configurations

The GPIO PA0 is feeding by the square output signal of the VCO. The first task the STM32 has to do is to detect rising edges on that signal. For that, it is very simple. On STM32CubeMX, we have to configure the TIMER 2 Channel 1 on Input Capture direct mode :

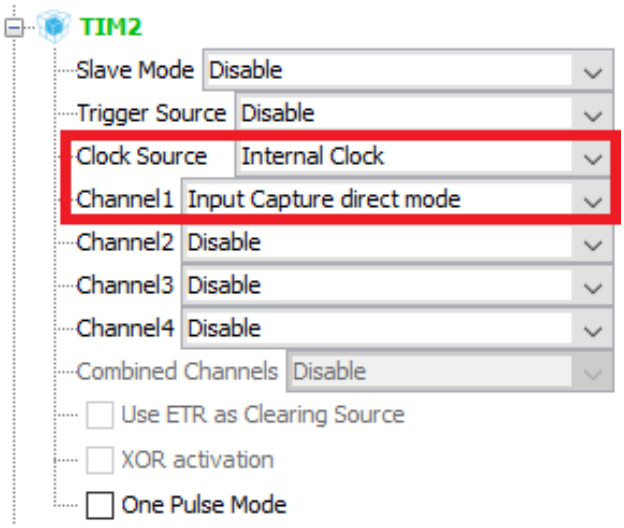


Figure 6.3: Input Capture direct mode configuration

Then, the Input Capture Channel 1 has to be configured. The Polarity Selection is Rising Edge, like it is said in the Principle sub-section :

Figure 6.4: Input Capture Channel 1 configuration

To finish that first configuration, we have to enable TIM2 global interrupt :

Figure 6.5: TIM2 global interrupt enabled

With that configuration, the function ***HAL_TIM_IC_CaptureCallback*** is called at each rising edge on the VCO's square signal. So the first sub-task to know the VCO's operation frequency is done.

6.2.3 Algorithm

The process to detect a rising edge on a square signal is the following :

- Configurations ! First, configure channel 1 of TIMER 2 on Input Capture direct mode with Polarity Selection on Rising Edge. It is possible to use an other configuration like EXTI one, but we need to know the time between two rising edges in order to compute the frequency. So, we need to use a timer for that. Then, enable TIM2 global interrupt.

- At that point, the main program is executed. Now, it is an interrupt mode. So, at each rising edge, an interruption is activated and then, the microprocessor executes the function ***HAL_TIM_IC_CaptureCallback***.

6.3 Signal frequency measurement

6.3.1 Principle

The measurement of a square signal frequency is very simple. Indeed, there is one period between two rising edges. So, if the STM32 counts the time T between two rising edges, it possible to know the frequency f : $f = \frac{1}{T}$.

With the following example, it is easier to understand how process the STM32 to measure the frequency of a square signal :

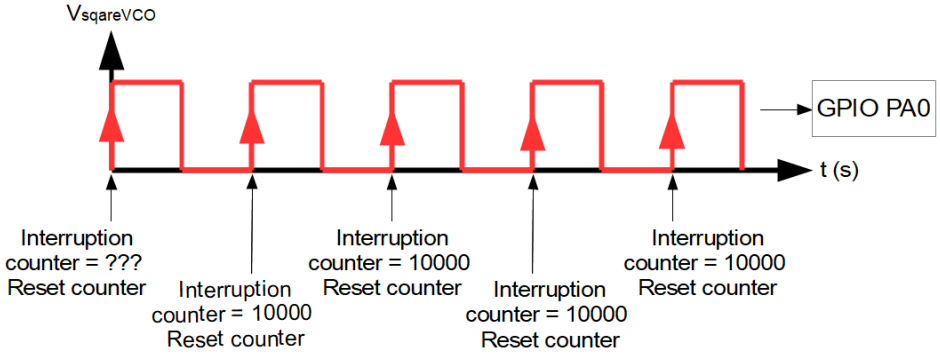


Figure 6.6: VCO's square output signal - rising edge detection and counter

If the counter counts at a frequency of 10MHZ, it means that the duration between two rising edges is 1ms : $T = \frac{CNT}{f_{CNT}} = \frac{10000}{10000000}$. So, the frequency of that signal would be f = 1 000 Hz.

6.3.2 Configurations

The configuration for the rising edge detection is already done. Now, it is the configuration of the TIMER we have to explain. The role of that peripheral is to count at 10MHz between two rising edges. As it is said just above, the computation in order to know the frequency of the signal is the following : $f = \frac{1}{T} = \frac{f_{CNT}}{CNT} = \frac{10000000}{CNT}$.

So we have to configure the TIMER in order to count at the frequency of 10MHz. First, the input frequency of the TIMER 2 is the frequency of APB1 bus times 2, so 90MHz according to the configuration done on STM32CubeMX. So, we have to divide by 9 the frequency in order to count at a frequency of 10MHz. The next scheme show how we have to configure the TIMER 2 :

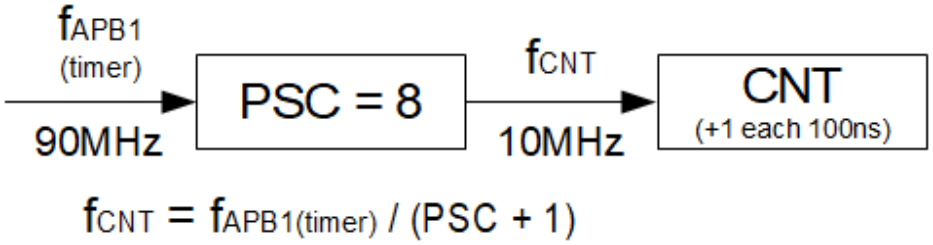


Figure 6.7: TIM2 configuration

Now, we know the value of PSC : PSC = 8. More over, we want the counter to count from 0 to a positive number, so, we want it to count up. The minimum frequency we can hear is 20Hz. So it's a period of 50ms and a max count of 500 000 with a frequency count of 10MHz. That max count is inferior than the Counter Period value, so it justifies the Counter Period value of 10 000 000. Finally, we have to configure the TIMER :

Counter Settings	
Prescaler (PSC - 16 bits value)	8
Counter Mode	Up
Counter Period (AutoReload Register - 32 bits value)	10000000
Internal Clock Division (CKD)	No Division

Figure 6.8: TIM2 configuration

6.3.3 Algorithm

In order to measure the frequency of a square signal, it is necessary to follow these next instructions :

- Configure channel 1 TIMER 2 and TIM2 global interrupt as it is said in sub-section 6.2.3 Algorithm

- Configure TIMER 2 for counting at 10MHz as it is said in the previous sub-section.

- At each rising edge, there is an interrupt and the function ***HAL_TIM_IC_CaptureCallback*** is called. In that function, the processor reads the counter value, resets the counter and computes the following operation : $f = \frac{f_{CNT}}{CNT} = \frac{10000000}{CNT}$. So, the STM32 can store the result of that operation in a variable or can write it on a screen for example.

6.3.4 Resolution and precision

The resolution in terms of duration of that method is shown on the next scheme :

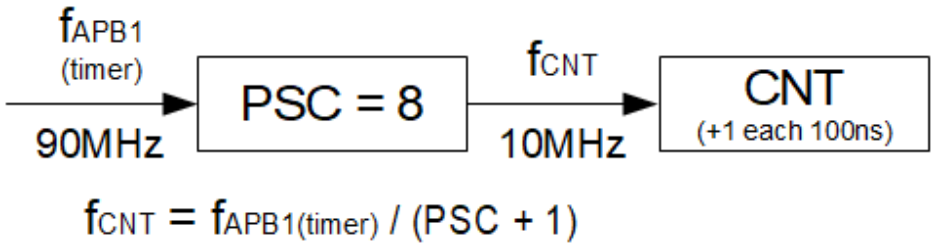


Figure 6.9: TIM2 configuration

Indeed, the resolution of that method is the smallest duration the counter can distinguish, so in that configuration, the resolution is 100ns.

The precision of the result is the reading of the three first digits which are always the same if the measure is done multiple times.

6.4 Calibration method

6.4.1 Algorithm

In order to create the voltage control level of the VCO, it is implemented a 12-bits DAC. So, there is 4096 different voltage control levels. For each of these 4096 different voltage values, it is necessary to know the associated operation frequency of the VCO. The analog output voltage of the DAC is given by the following operation : $V_{DAC} = \frac{V_N}{1000}$. So an array of size 4096 will store all frequency of all voltage control levels. The algorithm is described by the following diagram :

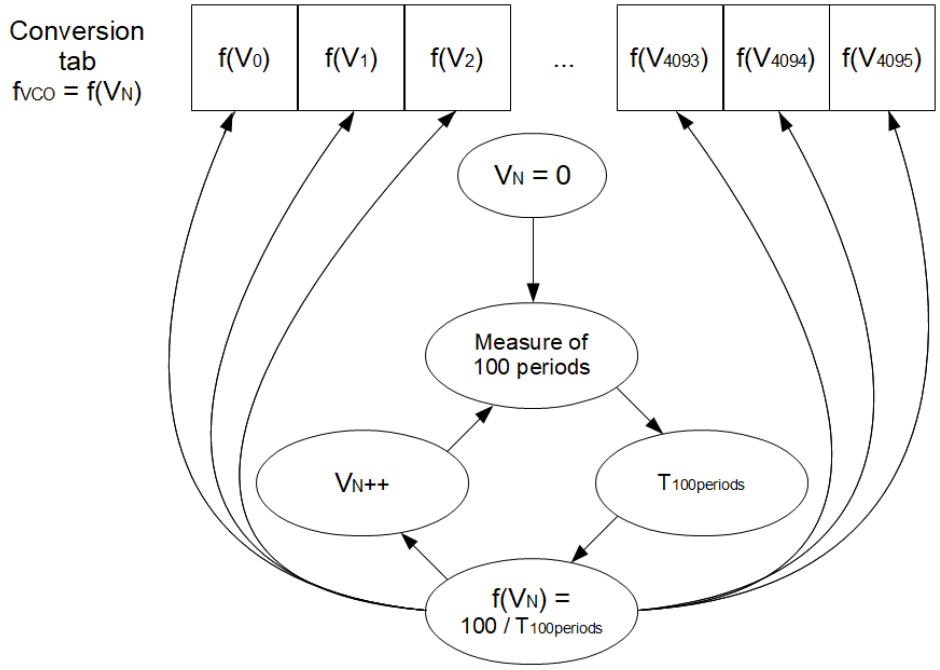


Figure 6.10:

So we understand that for each voltage control level, the STM32 wait for 100 periods in order to average the frequency. This will enhance the result and then, the STM32 could better choose the voltage control level to play a note accuracy.

6.4.2 Resolution and precision

The resolution of the global method is the same as the previous one. That is to say 100ns.

However, the precision of the result is better than the previous one because of the averaging. Thus, the precision of the result is the reading of the four first digits witch are always the same if the measure is done multiple times.

6.5 NVRAM - How to store data for next time ?

6.5.1 Introduction

In order to keep a well-tuned instrument during a show (keep in mind the temperature and humidity drifts), we can imagine that the manufacturer realize different times the calibration process for different temperature and different humidity. Then, we have multiple arrays of values and we have to store it in memory. But the STM32 has only volatile memory. So, we decided to implement a nvSRAM.

6.5.2 How it operates ?

By reading the datasheet, I understood that the memory is protected by wrong write command. So we first have to enable write operations (WREN Opcode 0x06). Then, I tried to read the Status Register (RDSR Opcode 0x5) in order to verify the bit WEL (latched at 1 after WREN), but I always had 0x00. So I read a lot the datasheet but I didn't have enought time to find the solution... So, good luck for the continuation !

Chapter 7

STM32 Information Processing and ADSR Envelope Generation (by Corentin Mantion)

7.1 Introduction

7.1.1 Context and Constraints

Generating the ADSR envelopes was part of a more general task, which was to transcribe the code developed for the Raspberry Pi last year. The project had to be hosted on a STM32 Nucleo instead, because the timing constraints were too short for the RPi to keep up with :
Although the ADSR generation was working fine, the system started to encounter latency when generating sound waves at high frequencies.
Thus, the code was transcribed by S. Reynal in a C file named dac_board.c, which has to control the Digital to Analog Converters on board, as well as the ADSR envelopes generation and peripherals initialization.

On the other hand, signals from the RPi had to be processed in order to trigger the corresponding functions (Drum Machine implementation, parameters changes,...). Some new functions had to be developed in accordance with L. Pineau and H. Peltzer's work. The following chapter will present a few concepts and equipment we will be working with, before detailing the code's writing and behavior.

7.1.2 An ADSR Envelope

An ADSR envelope (Attack Decay, Sustain, Release) works as a modulation of a parameter, for instance a gain or a cutoff frequency.

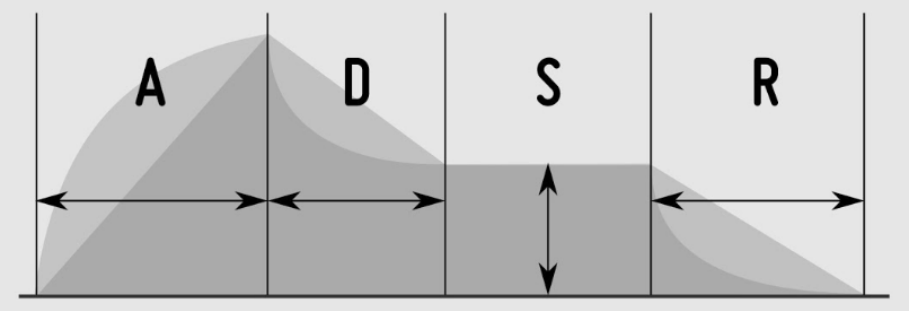


Figure 7.1: General Structure of an ADSR Envelope

The ADSR envelope has four phases, in which the envelope will follow a single behavior based on four parameters :

- Attack : During this phase, the parameter modulated by the envelope will go up from a value of 0 to a maximum value of 1, for a defined attack time.

- Decay : For a defined decay time, the parameter will decrease to a sustain value (comprised between 0 and 1).
- Sustain : This phase will last as long as a note is pressed. The parameter will stay equal to the sustain value.
- Release : Once the note is released, the parameter will slowly decrease to its original value (here, 0) during a defined release time.

7.1.3 DAC - MCP4822 Overview

The board designed to host the STM32 Nucleo also has several Digital to Analog Converters, referenced MCP4822, which will allow the signals generated by the card to be processed by the analog part of the synthesizer. This section will give an overview of their behavior and explain their perks, as well as the design constraints they create.

Package :

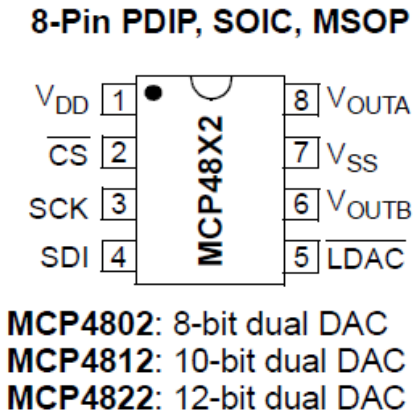


Figure 7.2: MCP4822 Package

The MCP4822 has two analog outputs, which can be synchronized or not thanks to the $\overline{LDAC}$ pin.

Signals Formatting :

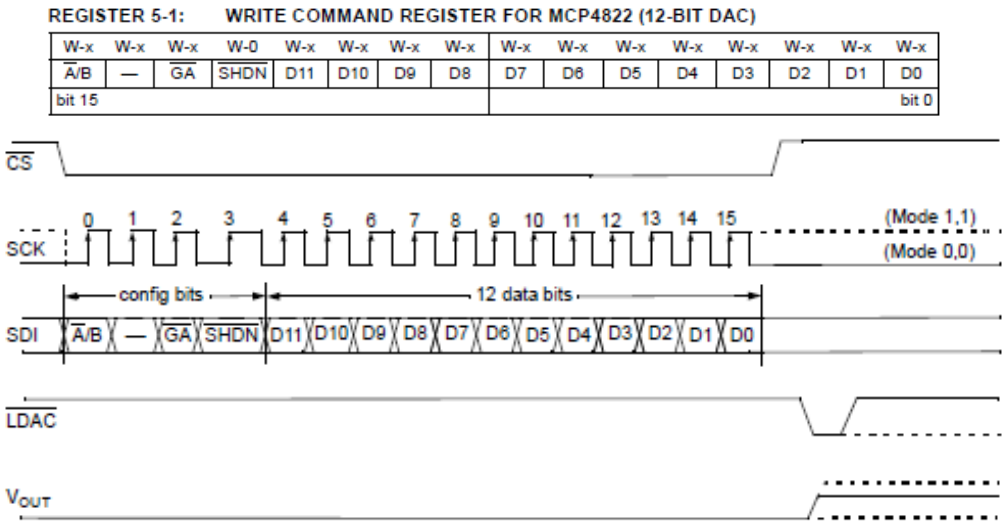


Figure 7.3: Data register and Signal timings

The data has to be sent to the DAC over two channels, one being a clock (SCK), the other being the data (SDI). This format allows the data to be sent over to the DAC using the SPI bus of the Nucleo board. The Chip Select ($\overline{CS}$) pin indicates when the transfer is over, and can be manually dealt with. The $\overline{LDAC}$ pin must be held at 1 during the transfer in order to synchronize the data output over the A and B channels.
The format of the data register requires the Nucleo to send two bytes of data over the SPI bus. The first four bytes of data set the output channel and gain, while the others allow us to determine the output level.

7.1.4 The MIDI Protocol

The Musical Instrument Digital Interface (MIDI for short) is a technical standard describing communications protocol for electronic musical instruments. In our case, it will be our standard choice for communicating between the Raspberry Pi (RPi) and out STM32F767ZI (STM32).

Overview :

A MIDI message is comprised of three bytes (or 24b of data). Each of these three bytes carries a precise information.

Note ON/OFF :

In the event of a note trigger, the message will look like this :

- 1001nnnn : Note ON event.
- 1000nnnn : Note OFF event.

Where nnnn (comprised between 0 and 15) is the MIDI channel number. In our case, only channel number 10 (binary 1010) is reserved for the drum machine inputs.

The following two bytes will be :

- 0kkkkkkk : ”kkkkkkk” being the key note (0-127).
- 0vvvvvvv : ”vvvvvvv” being the key velocity (0-127).

Control Change :

In the event of a control change, the message will be :

- 1011nnnn : ”nnnn” being the channel number (unused here, 0-15).
- 0ccccccc : ”ccccccc” being the controller number (0-127).
- 0vvvvvvv : ”vvvvvvv” being the controller value (0-127).

More information can be found here.

7.2 STM32F767ZI Configuration

In this section, we will see how the STM32 has to be configured using STM32 CubeMX. We will go through the clock configuration, then the interrupts, and finally choose which pins we might use as GPIOs among the rest.

7.2.1 Clocks Configuration

The clock will be configured as following :

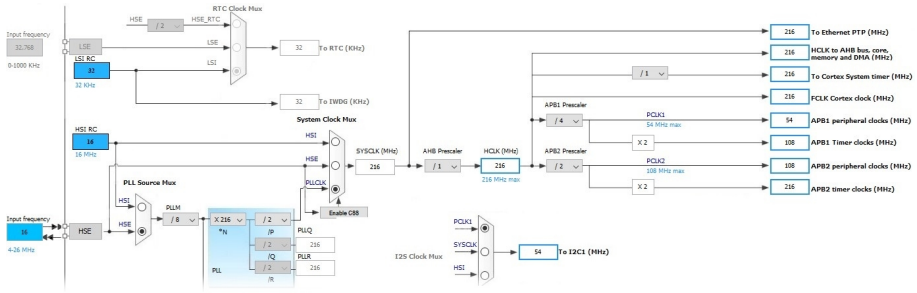


Figure 7.4: Clocks Flowchart on CubeMX

The main constraints that we have to respect here are the timers frequency. The timers will be used in the VCO calibration (see chapter 6), as well as in the envelope generation and the Drum Machine controls. We will have to correctly set up their interrupts in the ”Configuration” section of CubeMX.

7.2.2 Buses and Interrupts

We will have to use two SPI buses, one to receive data from the RPi, and one other to transmit data over to the DACs. We will use the SPI3 to receive data, and the SPI5 to transmit. Also, an I2C bus may have to be set up in order to use the MCP23017, however, in the project's current state, it will be useless. If you need to configure it, follow the flowchart presented above and set the I2C bus frequency to 100kHz.

As you can see, we set up several interrupts here.
Indeed, TIM2 Global Interrupt is vital to VCO Calibration (see chapter 6) and TIM1 Update Interrupts are needed in order to properly update the DACs. SPI5 Global Interrupts must be enabled to change the state of the $\overline{LDAC}$ pin of the DACs once the data has successfully been transmitted. External Interrupts EXTI

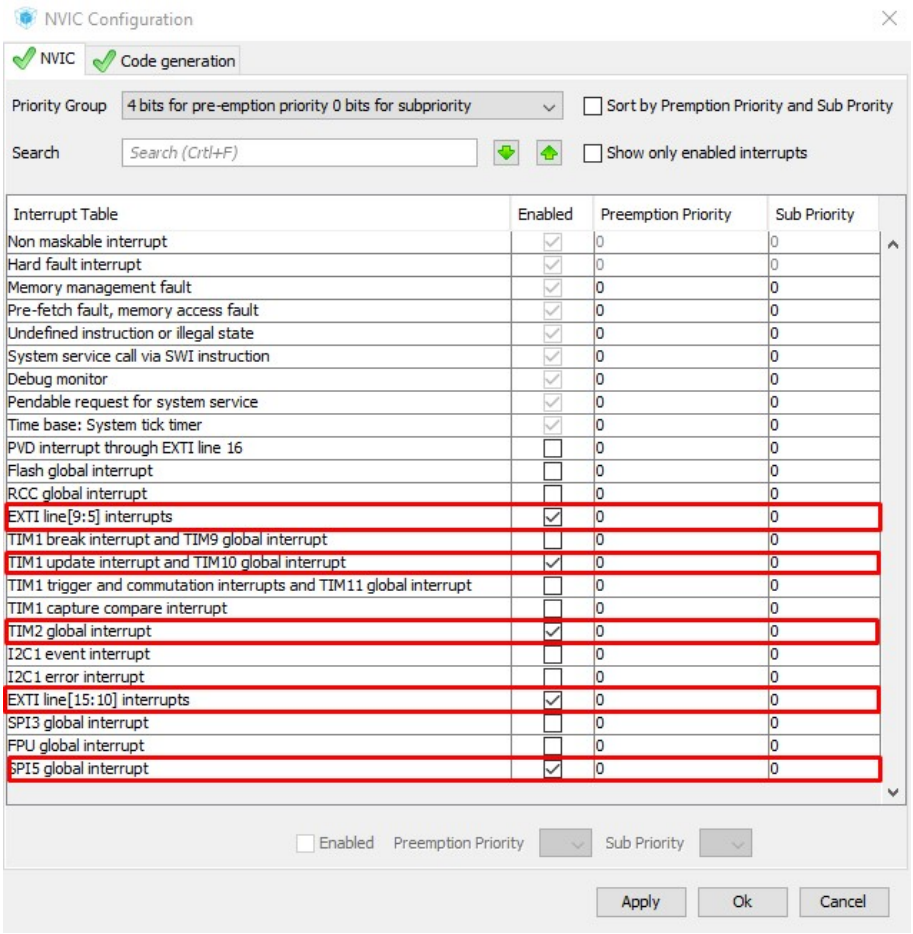


Figure 7.5: NVIC Configuration Tab

must be enabled in order to correctly receive data from the RPi, but also for debug purposes.

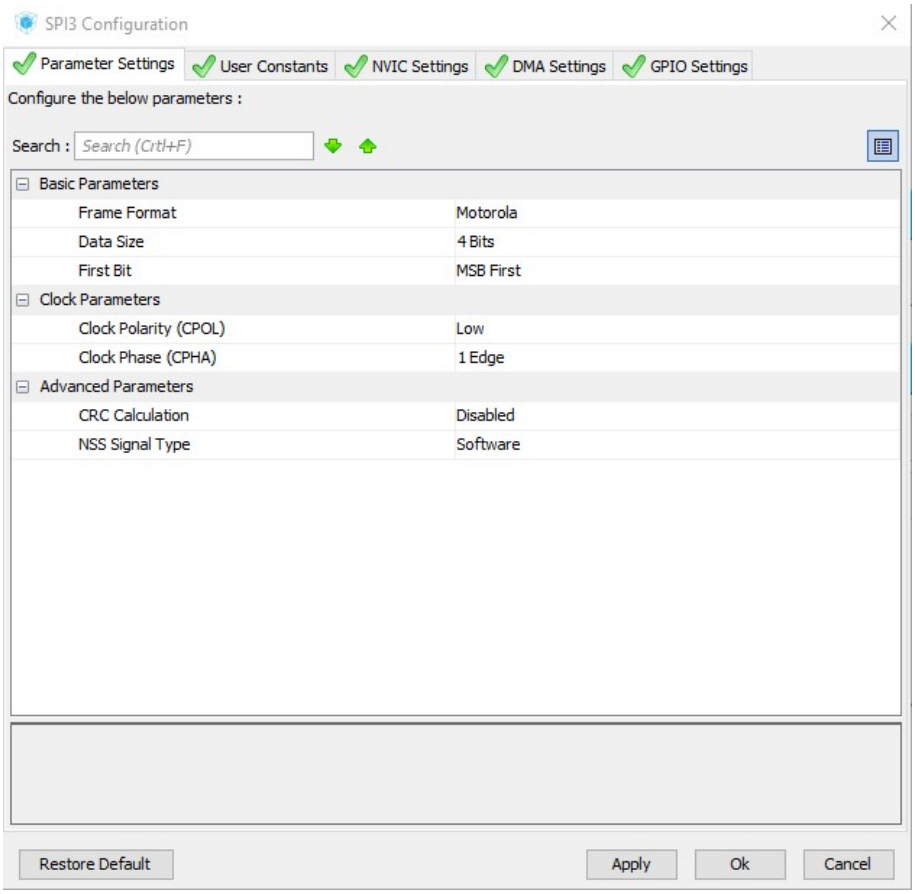


Figure 7.6: SPI3 Configuration Tab

The configuration of the SPI3 bus is set to default.

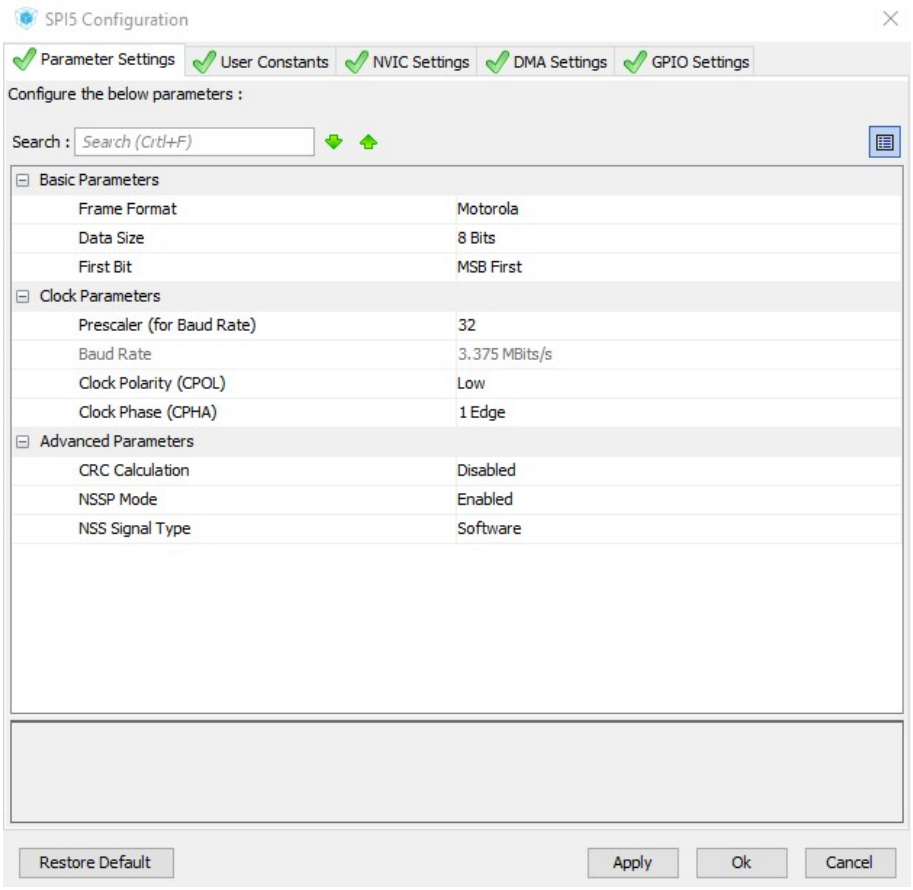


Figure 7.7: SPI5 Configuration Tab

The configuration of the SPI5 bus is set to default, however the data size and the prescaler have been changed in order to match the DACs specifications and to avoid latency.

7.2.3 Pins Configuration

DISCLAIMER : In this section, the pinout will be presented in the state it had during our last demo. The pinout may vary in later versions. Please refer to the next section and the code in ”main.h” for an exhaustive list of all the GPIOs Pins that have been implemented in the code.

After setting up these interrupts and peripherals, we will have to look into some more GPIOs ports in order to trigger the Drum Machine, get synchronization from the RPi, or simply generate white noise for some other components. Here we will list the GPIOs we will need :

- External Interrupt from RPi
- BassDrum Trigger (Drum Machine)
- Rim Shot Trigger (Drum Machine)
- Snare Trigger (Drum Machine)
- Low Tom Trigger (Drum Machine)
- High Tom Trigger (Drum Machine)
- White Noise Output (Drum Machine)
- 2nd Order VCF Switch
- 4th Order VCF Swith
- VCO3340 Sync Pin
- VCO3340 Sawtooth (Waveform)
- VCO3340 Pulse (Waveform)
- VCO3340 Triangle (Waveform)

We will also need some other GPIOs for debug purposes, such as an external interrupt on the blue button of the STM32, or a manual DAC trigger.

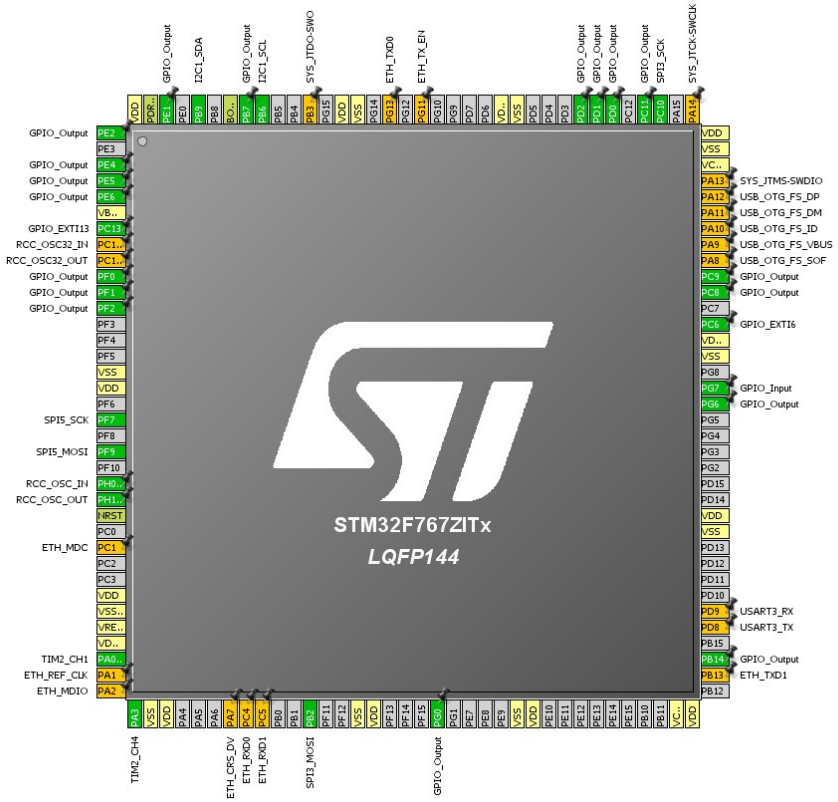
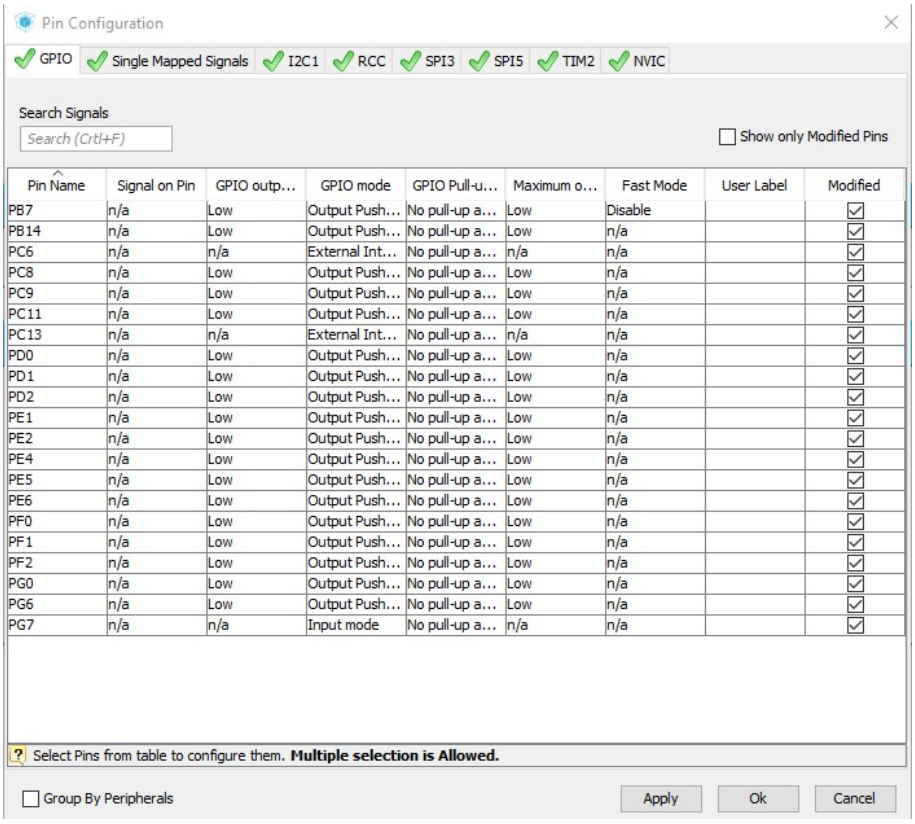


Figure 7.8: STM32 Pinout Overview

After setting up the pins, that is what we obtain (note : Some test pins may not have been set to their reset state).



Pin Name	Signal on Pin	GPIO outp...	GPIO mode	GPIO Pull-u...	Maximum o...	Fast Mode	User Label	Modified
PB7	n/a	Low	Output Push...	No pull-up a...	Low	Disable		☑
PB14	n/a	Low	Output Push...	No pull-up a...	Low	n/a		☑
PC6	n/a	n/a	External Int...	No pull-up a...	n/a	n/a		☑
PC8	n/a	Low	Output Push...	No pull-up a...	Low	n/a		☑
PC9	n/a	Low	Output Push...	No pull-up a...	Low	n/a		☑
PC11	n/a	Low	Output Push...	No pull-up a...	Low	n/a		☑
PC13	n/a	n/a	External Int...	No pull-up a...	n/a	n/a		☑
PD0	n/a	Low	Output Push...	No pull-up a...	Low	n/a		☑
PD1	n/a	Low	Output Push...	No pull-up a...	Low	n/a		☑
PD2	n/a	Low	Output Push...	No pull-up a...	Low	n/a		☑
PE1	n/a	Low	Output Push...	No pull-up a...	Low	n/a		☑
PE2	n/a	Low	Output Push...	No pull-up a...	Low	n/a		☑
PE4	n/a	Low	Output Push...	No pull-up a...	Low	n/a		☑
PE5	n/a	Low	Output Push...	No pull-up a...	Low	n/a		☑
PE6	n/a	Low	Output Push...	No pull-up a...	Low	n/a		☑
PF0	n/a	Low	Output Push...	No pull-up a...	Low	n/a		☑
PF1	n/a	Low	Output Push...	No pull-up a...	Low	n/a		☑
PF2	n/a	Low	Output Push...	No pull-up a...	Low	n/a		☑
PG0	n/a	Low	Output Push...	No pull-up a...	Low	n/a		☑
PG6	n/a	Low	Output Push...	No pull-up a...	Low	n/a		☑
PG7	n/a	n/a	Input mode	No pull-up a...	n/a	n/a		☑

Figure 7.9: GPIOs Configuration

Here is how our GPIOs are configured in the Configuration Tab of CubeMX. Please note that another GPIO pin must be configured as an analog output for the white noise generation.

In order to give you a complete overview of the pins that have been set up on the STM32, here is a summary table (please note that some pins were unused at the time of the first demonstration, so there may be some inconsistencies. Further details regarding the pinout can be found in the next section) :

	CN8 (odd)	CN8 (even)	
1		PC8=A0 LS138	2
3		PC=A1 LS138	4
5		PC10=SPI3_SCK	6
7	3.3V	PC11=A2 LS138	8
9	5V	PC12	10
11	GND	PD2 (411 SYNC)	12
13	GND	PG2	14
15		PG3	16

	CN9 (odd)	CN9 (even)	
1	PA3=TIM2 CH4	PD7	2
3	PC0	PD6	4
5	PC3	PD5	6
7	PF3	PD4	8
9	PF5/PB9	PD3	10
11	PF10/PB8	GND	12
13	GND	PE2 (vcf 4th order)	14
15	PA7	PE4 (vcf 2nd order)	16
17	PF2 (high tom)	PE5 (kick)	18
19	PF1 (low tom)	PE6 (411 TRI)	20
21	PF0 (rim shot)	PE3	22
23	GND	PF8	24
25	PD0 (411 SAW)	PF7=CLK SPI5	26
27	PD1	PF9=MOSI SPI5	28
29	PG0=CS manuel DAC	PG1	30

	CN7 (odd)	CN7 (even)	
1	PC6=CE from rpi	PB8 2	
3	PB15	PB9	4
5	PB13	-	6
7	PB12	GND	8
9	PA15	PA5	10
11	PC7	PA6	12
13	PB5	PA7	14
15	PB3	PD14	16
17	PA4	PD15	18
19	PB4	PF12	20

	CN10 (odd)	CN10 (even)	
1	-	PF13	2
3	-	PE9	4
5	GND	PE11	6
7	PB1	PF14	8
9	PC2	PE13	10
11	PF4	PF15	12
13	PB6	PG14	14
15	PB2=SPI3 MOSI	PG9	16
17	-	PE8	18
19	PD13	PE7	20
21	PD12	-	22
23	PD11	PE10	24
25	PE2	PE12	26
27	-	PE14	28
29	PA0=TIM2_CH1	PE15	30
31	PB0	PB10	32
33	PE0	PB11	34

7.3 Header Files and Concepts

In this section, we will show how our C header files were defined in order to suit our needs and ease code reading. Please note that our code is available on our GitHub repository.

main.h :

The header file ”main.h” will host several pieces of information regarding the pins names in the code. It can also serve as a reminder on how the pins have to be set up in the previous section.

```
/* Private define ------------------------------------------------------------*/

#define VCF_4THORDER_Pin GPIO_PIN_2
#define VCF_4THORDER_GPIO_Port GPIOE
#define VCF_2NDORDER_Pin GPIO_PIN_4
#define VCF_2NDORDER_GPIO_Port GPIOE
#define DRUM_KICK_Pin GPIO_PIN_5
#define DRUM_KICK_GPIO_Port GPIOE
#define TRI_3340_Pin GPIO_PIN_6
#define TRI_3340_GPIO_Port GPIOE
#define USER_Btn_Pin GPIO_PIN_13
#define USER_Btn_GPIO_Port GPIOC
#define USER_Btn_EXTI_IRQn EXTI15_10_IRQn
#define DRUM_RIM_Pin GPIO_PIN_0
#define DRUM_RIM_GPIO_Port GPIOF
#define DRUM_LOWTOM_Pin GPIO_PIN_1
#define DRUM_LOWTOM_GPIO_Port GPIOF
#define DRUM_HIGHTOM_Pin GPIO_PIN_2
#define DRUM_HIGHTOM_GPIO_Port GPIOF
#define MCO_Pin GPIO_PIN_0
#define MCO_GPIO_Port GPIOH
#define RMII_MDC_Pin GPIO_PIN_1
#define RMII_MDC_GPIO_Port GPIOC
#define RMII_REF_CLK_Pin GPIO_PIN_1
#define RMII_REF_CLK_GPIO_Port GPIOA
#define RMII_MDIO_Pin GPIO_PIN_2
#define RMII_MDIO_GPIO_Port GPIOA
#define RMII_CRS_DV_Pin GPIO_PIN_7
#define RMII_CRS_DV_GPIO_Port GPIOA
#define RMII_RXD0_Pin GPIO_PIN_4
#define RMII_RXD0_GPIO_Port GPIOC
#define RMII_RXD1_Pin GPIO_PIN_5
#define RMII_RXD1_GPIO_Port GPIOC
#define DAC_CS_Pin GPIO_PIN_0
#define DAC_CS_GPIO_Port GPIOG
#define RMII_TXD1_Pin GPIO_PIN_13
#define RMII_TXD1_GPIO_Port GPIOB
#define LD3_Pin GPIO_PIN_14
#define LD3_GPIO_Port GPIOB
#define STLK_RX_Pin GPIO_PIN_8
#define STLK_RX_GPIO_Port GPIOD
#define STLK_TX_Pin GPIO_PIN_9
#define STLK_TX_GPIO_Port GPIOD
#define USB_PowerSwitchOn_Pin GPIO_PIN_6
#define USB_PowerSwitchOn_GPIO_Port GPIOG
```

```
#define USB_OverCurrent_Pin GPIO_PIN_7
#define USB_OverCurrent_GPIO_Port GPIOG
#define CE_RPi_Pin GPIO_PIN_6
#define CE_RPi_GPIO_Port GPIOC
#define CE_RPi_EXTI_IRQn EXTI9_5_IRQn
#define LS138_A0_Pin GPIO_PIN_8
#define LS138_A0_GPIO_Port GPIOC
#define LS138_A1_Pin GPIO_PIN_9
#define LS138_A1_GPIO_Port GPIOC
#define USB_SOF_Pin GPIO_PIN_8
#define USB_SOF_GPIO_Port GPIOA
#define USB_VBUS_Pin GPIO_PIN_9
#define USB_VBUS_GPIO_Port GPIOA
#define USB_ID_Pin GPIO_PIN_10
#define USB_ID_GPIO_Port GPIOA
#define USB_DM_Pin GPIO_PIN_11
#define USB_DM_GPIO_Port GPIOA
#define USB_DP_Pin GPIO_PIN_12
#define USB_DP_GPIO_Port GPIOA
#define TMS_Pin GPIO_PIN_13
#define TMS_GPIO_Port GPIOA
#define TCK_Pin GPIO_PIN_14
#define TCK_GPIO_Port GPIOA
#define LS138_A2_Pin GPIO_PIN_11
#define LS138_A2_GPIO_Port GPIOC
#define SAW_3340_Pin GPIO_PIN_0
#define SAW_3340_GPIO_Port GPIOD
#define DRUM_SNARE_Pin GPIO_PIN_1
#define DRUM_SNARE_GPIO_Port GPIOD
#define SYNC_3340_Pin GPIO_PIN_2
#define SYNC_3340_GPIO_Port GPIOD
#define RMII_TX_EN_Pin GPIO_PIN_11
#define RMII_TX_EN_GPIO_Port GPIOG
#define RMII_TXD0_Pin GPIO_PIN_13
#define RMII_TXD0_GPIO_Port GPIOG
#define SW0_Pin GPIO_PIN_3
#define SW0_GPIO_Port GPIOB
#define LD2_Pin GPIO_PIN_7
#define LD2_GPIO_Port GPIOB
#define PULSE_3340_Pin GPIO_PIN_1
#define PULSE_3340_GPIO_Port GPIOE
```

Our custom code starts one line 48 of ”main.h”. Here, you can see that all GPIO Pins have been given an explicit name.

midi.h :

In ”midi.h”, we will define structure designed to store the midi parameters of a note, as well as useful enums and constants :

```
#ifndef MIDI_H_
#define MIDI_H_


/*
 *The note being played, between 0 and 127 (MIDI format)
 *and its velocity (could be the last note played if it is
 *set to "OFF"
 */
typedef struct {
  int note; // 0-127
  int velocity; // 0-127
} MidiNote;

/*
 * An enum for the state machine that processes MIDI messages three by three
 */
typedef enum {
                WAITING_FOR_BYTE1, // waiting for byte #1
                WAITING_FOR_BYTE2,  // waiting for byte #2
                WAITING_FOR_BYTE3// waiting for byte #3
} midi_receiver_state_t;

/*
 * A collection of constants for MIDI status bytes
 */
#define NOTE_ON                 0x90
#define NOTE_OFF                0x80
#define CONTROL_CHANGE          0xB0
#define PITCH_BEND              0xE0


#endif /* MIDI_H_ */
```

adsr.h :

In ”adsr.h”, we will define constants and structures allowing us to handle the envelope generation easily, but we will also set up some other components for the drum machine or the VCO that could be moved to another header file, however due to timing constraints it could not be done in time. Nonetheless, those sections are quite easy to understand and their behavior will be detailed as well.

```
#ifndef ADSR_H_
#define ADSR_H_

// defines for ADSR envelopes default values (times are in ms)
#define DEF_ATTACK_TIME 10
#define DEF_DECAY_TIME 400
#define DEF_RELEASE_TIME 200

#define DEF_ATTACK_TIME_VCF 500
#define DEF_DECAY_TIME_VCF 200
#define DEF_RELEASE_TIME_VCF 200

#define MAX_ATTACK_TIME 1000
#define MAX_DECAY_TIME 1000
#define MAX_RELEASE_TIME 5000
#define MAX_SUSTAIN_LVL 1

#define MAX_VC_SENSI 1
#define MAX_MIXER 1
// sets the maximum permitted voltage shift in % :
#define MAX_KBD_TRACKING 0.3


#define DEF_SUSTAIN_LVL 0.5
#define DEF_SUSTAIN_LVL_VCF 0.5
#define DEF_VELOCITY_SENSITIVITY_VCA 0.1
#define DEF_VELOCITY_SENSITIVITY_VCF 0.0
#define DEF_KBD_TRACKING 0.0
#define DEF_ENV_AMOUNT 0.9
#define DEF_CUTOFF 0.8
#define DEF_RESONANCE 0.0

#define DEF_VCO_3340_PWM_DUTY 0.5
```

At the beginning of this file, we set up all the constants we will use later on in the source file. The default times for each stage of the ADSR envelope are defined in the first section, as well as their maximum values. Some other parameters like sensitivity, keyboard tracking, resonance or PWM duty are also defined, allowing us to use them later.

```
/*
 * ADSR env params
 */
typedef struct {
    double attackTimeMs;    //
    double decayTimeMs;     //
    double sustainLevel;     // b/w 0.0 et 1.0
    double releaseTimeMs;   //
} AdsrParams;
```

This is the structure containing the parameters of an ADSR envelope. It can be defined by only four values.

```
/*
 * enum for the various state machine states
 */
typedef enum {
  IDLE,         // 0
  ATTACK,       // 1 ... Lasts 5 Tau
  DECAY,        // 2 ... Lasts until Note Off
  RELEASE       // 3
} AdsrMachineState;
```

Our ADSR envelope generation will work using a state machine, with each state corresponding to one section of the ADSR envelope. This enum here will allow us to give explicit names to each state.

```
/* global parameters cutoff and Q influences the
 *shape of the VCF adsr envelope
 */
typedef struct {
  double vcfCutoff; // between 0 and 100%
  double vcfResonance; // between 0 and 100%
} GlobalSynthParams;
```

In this structure, we will only store the cutoff and resonance of the VCF.

```
/*
  parameters for the generation of the VCA envelopes
*/
typedef struct {
  AdsrMachineState machineState; // current state of the state machine
  AdsrParams* adsrParam; // adsr env params
  double amplitude; // current CV value
  double velocitySensitivity;
  /* 0-100% ; actual CV is modulated by velocity
  *depending on this parameter (0= no mod, 1=full mod)
  */

  /* linear enveloppes for V2140D (aka in dB) */
  double tmpDelta;
  /* env value gets increased by this quantity
   *at each time step (this is dx/dt * 1ms)
   */
  double tmpTargetLevel;
  /* target level inside each phase (ex: 1.0 for the A phase,
   *sustain for the D phase, 0 for the R phase)
   */

  /* Exponential enveloppes for LM13700 based VCA with no exp conv : */
  //double tmpExp;
  /* stores exp(-t/tau) on a temporary basis to speed up
   *computation for exponential envelopes
   */
  //double mulFactorAttack; // exp(-T/tau_a), where T=timer period (TIMER_PERIOD)
  //double mulFactorDecay; // exp(-T/tau_d)
  //double mulFactorRelease; // exp(-T/tau_r)
} StateMachineVca;

/*
  parameters for the generation of the VCF envelopes
*/
typedef struct {
  AdsrMachineState machineState; // current state of the state machine
  AdsrParams* adsrParam; // adsr env params
  int t; // time, reset at the beginning of each phase
  int tMax;
  // max time for the current phase, if relevant (i.e. irrelevant for sustain)
  double cutoffFrequency; // current CV value
  double velocitySensitivity; // 0-100% ; filter sensitivity to velocity
  double kbdTracking; // 0-100%, filter sensitivity to current note freq
  double envAmount; // 0-100%, filter sensitivity to envelope
  double tmpDelta;
  /* env value gets increased by this quantity
   *at each time step (this is dx/dt * 1ms)
   */
  double tmpTargetLevel;
  /* target level at end of each phase,
   *depends on env_amount, sustain and velocity_mul_factor
   */
  double tmpKbdtrackingShiftFactor;
  // global shift (aka voltage addition) due to kbd_tracking

  /* exponential env generation:
  double tmpExp; // stores exp(-t/tau) on a temporary basis
  double mulFactorAttack; // exp(-T/tau_a), where T=timer period (TIMER_PERIOD)
  double mulFactorDecay; // exp(-T/tau_d)
  double mulFactorRelease; // exp(-T/tau_r)
```

```
  */
  double tmpIncrease;
  // env value gets increased by this quantity at each time step
} StateMachineVcf;
```

Here, we will define in two different structures that will follow the same organization the parameters and variables we will use to properly generate our envelopes. In the case of the VCA, we will store the state of the machine state, the time values our envelope will have to follow using the structure "AdsrParams" defined earlier, and store its amplitude. We will also store the velocity sensitivity separately.
Then, we will store internal variables tmpDelta and tmpTargetLevel which will allow the amplitude of the envelope to be updated in accordance with the velocity sensitivity parameter.
The commented code below was used to generate exponential envelopes, when we did not have a log-scale converter. However, now, we are able to use linear envelopes. This can still be implemented if one wishes to modify the envelope shape, for instance.
The code for the VCF envelope takes into account the other parameters like keyboard tracking or the filter sensitivity.

```
/*
 * struct for VCO parameters
 */
typedef struct {
        double detune; //-50%-50% of one tone
        int octave; // can be positive or negative
} VcoParameters;
```

This structure hosts two parameters that will allow to dynamically modify the tone of a note on two different scales.

```
/* struct for drums */
typedef struct {
        int bassdrumCounter;
        int rimshotCounter;
        int snareCounter;
        int lowtomCounter;
        int hightomCounter;
} DrumTriggers;

#define          BASS_DRUM_NOTE 36
#define RIMSHOT_NOTE 37
#define SNARE_NOTE 38
#define LOWTOM_NOTE 41
#define HIGHTOM_NOTE 48


#endif /* ADSR_H_ */
```

Finally, this structure stores the counters that will allow the STM32 to reset the triggers for each component of the drum machine. Also, five constants are defined, holding the note value in the MIDI standard of each component.

dac_board.h :

The file ”dac_board.h” also defines constants vital to the generation of ADSR envelopes, while setting up two enums and the prototypes for all functions in our C source file.

```
#ifndef DAC_BOARD_H_
#define DAC_BOARD_H_


// timers
#define US        * 1
#define MS        * 1000 US
#define HTIM1_INPUT_FREQ 108000000.0
#define HTIM1_PRESCALER 100.0
#define HTIM1_PERIOD 50.0
// for Timer 2, see vco_calibration.h:
#define TIMER_PERIOD      (HTIM1_PRESCALER * HTIM1_PERIOD/HTIM1_INPUT_FREQ)
// 20kHz, 50us
#define ADSR_TIMER_PERIOD_FACTOR 20
#define ADSR_TIMER_PERIOD (ADSR_TIMER_PERIOD_FACTOR*TIMER_PERIOD) // 1ms = 0.001
#define ADSR_TIMER_PERIOD_MS (1000.0*ADSR_TIMER_PERIOD) // around 1ms

// defines for the MCP4822 device
#define MCP4822_CHANNEL_A                0x30    // 0011 A\/B=0 RES=0 GA\=1 SHDN\=1
#define MCP4822_CHANNEL_B                0xB0    // 1011
#define MCP4822_CHANNEL_A_GAIN2          0x10    // 0001 A\/B=0 RES=0 GA\=0 SHDN\=1
#define MCP4822_CHANNEL_B_GAIN2          0x90    // 1001


//Channels
#define DRUM_CHANNEL 10
```

Here, we will store the prescaler values of our timers in order to construct a time base that will allow us process events and create envelopes with high fidelity. We then store useful values for the implementation of the DACs (MCP4822), which will allow us to choose easily their channel A or B, as well as their gain. Then, finally, we define the Channel for the Drum Machine.

```
/*
 * enumation of available DACs on the board
 */
typedef enum {
          DAC_VCO_13700,                  // LM13700 VCO
          DAC_VCO_3340_FREQ,                        // CEM3340 VCO: frequency
          DAC_VCO_3340_PWM_DUTY,                    // CEM3340 VCO: PWM duty cycle
          DAC_VCF_CUTOFF,                 // AS3320 cutoff CV
          DAC_VCF_RES,                    // AS3320 resonance CV
          DAC_V2140D_IN1,                 // V2140D quad vca input #1
          DAC_V2140D_IN2,                 // etc
          DAC_V2140D_IN3,
          DAC_V2140D_IN4,
          //DAC_VCO_DIG,     // Digitally generated waveform, not used yet
          DAC_V2140D_IN5,
          DAC_V2140D_IN6,
          DAC_V2140D_IN7,
          DAC_V2140D_IN8,
          DAC_NOISE,
          DAC_EN_RABE_A,
          DAC_EN_RABE_B,
          /* the following are aliases to mixer inputs
           *(see function dacWrite for details on how they are being used)
           */
          DAC_V2140D_3340_LVL,
          DAC_V2140D_13700_TRI_LVL,
          DAC_V2140D_13700_SQU_LVL,
          DAC_V2140D_13700_SUBBASS_LVL,
          DAC_V2140D_FM_LVL,
          DAC_V2140D_RINGMOD_LVL,
          DAC_V2140D_SH_LVL,
          DAC_V2140D_VCA              // was LM13700 VCA, now carried out by V2140D
} Dac;
```

This enum here will give each DAC a specific name according to its purpose. Their names are quite self-explanatory.

```
/* tunable MIDI CC parameters ; parameters are numbered
 *from 0 in the order they are given in the enum
 */
typedef enum {
        DETUNE_3340, // N/A
        OCTAVE_3340, // 0 1 2 3
        SYNC_3340, // 0 ou 127
        WAVE_3340,  // 0 1 2
        PWM_3340,// 0-127
        LEVEL_3340, // 0-127 mixer 1

        DETUNE_13700, // N/A
        OCTAVE_13700, // 0 1 2 3
        WAVE_13700, // 0-127 between square/triangle (mixer 2 mixer 3)
        LEVEL_13700, // 0-127 mixer 1

        VCF_CUTOFF,
        VCF_RESONANCE,
        VCF_ORDER,
        VCF_KBDTRACKING,
        VCF_EG,
        VCF_VELOCITY_SENSITIVITY,
        VCF_ATTACK,
        VCF_DECAY,
        VCF_SUSTAIN,
        VCF_RELEASE,

        VCA_EG,
        VCA_VELOCITY_SENSITIVITY,
        VCA_ATTACK,
        VCA_DECAY,
        VCA_SUSTAIN,
        VCA_RELEASE,

        // Add other parameters here
} MidiCCParam;
```

Here, this enum will list all the parameters we will be able to set on the STM32. Those parameters have to follow the same order than the list on the RPi. Thus, this list have been created with the help of L. Pineau and H. Peltzer.

```
/* Private function prototypes -----------------------------------------------*/

void dac4822ABWrite(int word12bits, int chip, int channelAB);
void dacSelect(int chipNumber);
void dacWrite(int word12bits, Dac targetDac);
void initSynthParams();
void muteAllDACs();
void dacVcaWrite(double amp);
void dacVcfCutoffWrite(double cutoff);
void updateVCAEnveloppeStateMachine();
void updateVCFEnveloppeStateMachine();
void midiNoteOnHandler();
void midiNoteOffHandler();
//void midiFromSpiMessageHandler(uint8_t byte); @deprecated
void playDemo();
void processIncomingMidiMessage(uint8_t status, uint8_t data1, uint8_t data2);
void setSynthParam(uint8_t id, uint8_t value);
void setMidiCCParam(MidiCCParam param, uint8_t value);
void playDrumMachine(uint8_t data1, uint8_t data2);
void updateDrumMachine();

#endif /* DAC_BOARD_H_ */
```

Here, we list all the function prototypes that will be found in our C source file. Here is a short summary of their behavior (note : The functions and their behavior will be fully commented in the next section) :

- dac4822ABWrite() : Writes a 12 bits word onto a specified channel of a specified chip.
- dacSelect() : Select a DAC using a specified chip number by sending a signal to the 74LS128 decoder.
- dacWrite() : Write a 12 bits word on a specified DAC (using the enum).
- initSynthParams() : Sets the synthesizer parameters to defaults.
- muteAllDACs() : Mute all DACs in order to initialize the synthesizer.
- dacVCAWrite() : Write a specified amplitude value onto the appropriate DAC.
- dacVCFCutoffWrite() : Write a specified cutoff value onto the appropriate DAC.
- updateVCAEnveloppeStateMachine() : Updates the VCA Envelope.
- updateVCFEnveloppeStateMachine() : Updates the VCF Envelope.

- midiNoteOnHandler() : Called when a midi note is pressed, initialize the state machine to IDLE state.
- midiNoteOffHandler() : Called when a midi note is released, force the state machine to RELEASE state.
- midiFromSpiMessageHandler() : Deprecated, used to translate a message on the SPI buffer to MIDI.
- playDemo() : Called after initialization, served as demo and debug purposes. It will not be covered in the next section.
- processIncomingMidiMessage() : Called on external interrupt from the RPi, calls the appropriate function depending on the nature of the MIDI message.
- setSynthParam() : Deprecated, used to set a specified synthesizer parameter to a specified value or state.
- setMidiCCParam() : Sets a specified synthesizer parameter in the MidiCCParam structure to a specified value or state.
- playDrumMachine() : Once called, triggers the appropriate GPIO for an existing Drum Machine note.
- updateDrumMachine() : Resets the trigger of an element of the drum machine once it has been held for a certain amount of time.

7.4 Source Files and Comments

In this section, we will describe how every single function implemented in the STM32F767ZI works and why they were designed this way, hopefully giving some useful piece of information for anyone willing to continue this project.

DAC writers :

In this section, you will find :

- dacSelect()
- dac4822ABWrite()
- dacWrite()
- muteAllDACs()
- dacVCAWrite()
- dacVCFCutoffWrite()

- HAL$_SPI_TxCpltCallback()$

dacSelect(int chipNumber)

```
/**
 * sends the appropriate address to the 74LS128 3-to-8 decoder
 * so that the corresponding MCP4822 chip is activated when CS is asserted later
 */
void dacSelect(int chipNumber){

        // selects a DAC: A0=PC4, A1=PB1, A2=PC5 and address=A2.A1.A0
        chipNumber &= 0x07;

        switch (chipNumber){
        case 0:
                //GPIOC->BSRR = (1<<(16+2)); // reset A2=PC2
                //GPIOB->BSRR = (1<<(16+11)); // reset A1=PB11
                //GPIOC->BSRR = (1<<(16)); // reset A0=PC0
                HAL_GPIO_WritePin(LS138_A2_GPIO_Port,
                    LS138_A2_Pin, GPIO_PIN_RESET);
                HAL_GPIO_WritePin(LS138_A1_GPIO_Port,
                    LS138_A1_Pin, GPIO_PIN_RESET);
                HAL_GPIO_WritePin(LS138_A0_GPIO_Port,
                    LS138_A0_Pin, GPIO_PIN_RESET);
                break;
        case 1:
                //GPIOC->BSRR = (1<<(16+2)); // reset A2=PC2
                //GPIOB->BSRR = (1<<(16+11)); // reset A1=PB11
                //GPIOC->BSRR = (1<<(0));          // set A0=PC0
                HAL_GPIO_WritePin(LS138_A2_GPIO_Port,
                    LS138_A2_Pin, GPIO_PIN_RESET);
                HAL_GPIO_WritePin(LS138_A1_GPIO_Port,
                    LS138_A1_Pin, GPIO_PIN_RESET);
                HAL_GPIO_WritePin(LS138_A0_GPIO_Port,
                    LS138_A0_Pin, GPIO_PIN_SET);
                break;
        case 2:
                //GPIOC->BSRR = (1<<(16+2)); // reset A2=PC2
                //GPIOB->BSRR = (1<<(11));     // set A1=PB11
                //GPIOC->BSRR = (1<<(16)); // reset A0=PC0
                HAL_GPIO_WritePin(LS138_A2_GPIO_Port,
                    LS138_A2_Pin, GPIO_PIN_RESET);
                HAL_GPIO_WritePin(LS138_A1_GPIO_Port,
                    LS138_A1_Pin, GPIO_PIN_SET);
                HAL_GPIO_WritePin(LS138_A0_GPIO_Port,
                    LS138_A0_Pin, GPIO_PIN_RESET);
                break;
        case 3:
                //GPIOC->BSRR = (1<<(16+2)); // reset A2=PC2
                //GPIOB->BSRR = (1<<(11));     // set A1=PB11
                //GPIOC->BSRR = (1<<(0));     // set A0=PC0
                HAL_GPIO_WritePin(LS138_A2_GPIO_Port,
                    LS138_A2_Pin, GPIO_PIN_RESET);
                HAL_GPIO_WritePin(LS138_A1_GPIO_Port,
                    LS138_A1_Pin, GPIO_PIN_SET);
                HAL_GPIO_WritePin(LS138_A0_GPIO_Port,
                    LS138_A0_Pin, GPIO_PIN_SET);
                break;
        case 4:
                //GPIOC->BSRR = (1<<(2));     // set A2=PC2
                //GPIOB->BSRR = (1<<(16+11)); // reset A1=PB11
                //GPIOC->BSRR = (1<<(16)); // reset A0=PC0
```

```
                HAL_GPIO_WritePin(LS138_A2_GPIO_Port,
                    LS138_A2_Pin, GPIO_PIN_SET);
                HAL_GPIO_WritePin(LS138_A1_GPIO_Port,
                    LS138_A1_Pin, GPIO_PIN_RESET);
                HAL_GPIO_WritePin(LS138_A0_GPIO_Port,
                    LS138_A0_Pin, GPIO_PIN_RESET);
                break;
        case 5:
                //GPIOC->BSRR = (1<<(2));      // set A2=PC2
                //GPIOB->BSRR = (1<<(16+11)); // reset A1=PB11
                //GPIOC->BSRR = (1<<(0));           // set A0=PC0
                HAL_GPIO_WritePin(LS138_A2_GPIO_Port,
                    LS138_A2_Pin, GPIO_PIN_SET);
                HAL_GPIO_WritePin(LS138_A1_GPIO_Port,
                    LS138_A1_Pin, GPIO_PIN_RESET);
                HAL_GPIO_WritePin(LS138_A0_GPIO_Port,
                    LS138_A0_Pin, GPIO_PIN_SET);
                break;
        case 6:
                //GPIOC->BSRR = (1<<(2));      // set A2=PC2
                //GPIOB->BSRR = (1<<(11));     // set A1=PB11
                //GPIOC->BSRR = (1<<(16)); // reset A0=PC0
                HAL_GPIO_WritePin(LS138_A2_GPIO_Port,
                    LS138_A2_Pin, GPIO_PIN_SET);
                HAL_GPIO_WritePin(LS138_A1_GPIO_Port,
                    LS138_A1_Pin, GPIO_PIN_SET);
                HAL_GPIO_WritePin(LS138_A0_GPIO_Port,
                    LS138_A0_Pin, GPIO_PIN_RESET);
                break;
        case 7:
                //GPIOC->BSRR = (1<<(2));      // set A2=PC2
                //GPIOB->BSRR = (1<<(11));     // set A1=PB11
                //GPIOC->BSRR = (1<<(0));     // set A0=PC0
                HAL_GPIO_WritePin(LS138_A2_GPIO_Port,
                    LS138_A2_Pin, GPIO_PIN_SET);
                HAL_GPIO_WritePin(LS138_A1_GPIO_Port,
                    LS138_A1_Pin, GPIO_PIN_SET);
                HAL_GPIO_WritePin(LS138_A0_GPIO_Port,
                    LS138_A0_Pin, GPIO_PIN_SET);
                break;
        }
}
```

We have 8 DACs in total on our board. Each DAC has two separate channels. This function here sends a signal to the 74LS128 3-to-8 decoder over three GPIO pins. It does a basic decimal to binary conversion and sends the binary code over those GPIOs using HAL functions. The commented code in each case does the same thing as the HAL functions, however by modifying the registry directly. This function uses the GPIO pins defines we did earlier in ”main.h”.

dac4822ABWrite(int word12bits, int chip, int channelAB)

```
void dac4822ABWrite(int word12bits, int chip, int channelAB){

        if (word12bits < 0) word12bits = 0;
        word12bits &= 0xFFF;

        // DAC select :
        dacSelect(chip);

        txSpiDacsBuff[0]=(((word12bits >> 8) & 0x0F)) | channelAB;
        txSpiDacsBuff[1]=(word12bits & 0xFF);

        HAL_GPIO_WritePin(DAC_CS_GPIO_Port, DAC_CS_Pin, GPIO_PIN_RESET);
        HAL_SPI_Transmit_IT(hspiDacs, txSpiDacsBuff, 2);

}
```

First, we will check if the message we are trying to send is negative, to avoid errors. Then, we will use a mask to format it, select the appropriate DAC using the chip number we have entered as argument, and write our words into the SPI buffers. They can be modified in order to send the data to the appropriate channel. After that, they will simply be sent using HAL functions, resetting the Chip Select Pin before that, in order to tell the DAC that data will be written.

dacWrite(int word12bits, Dac targetDac)

```
/**
 * Write the given word to the given DAC
 */
void dacWrite(int word12bits, Dac targetDac){

        switch (targetDac){
                case DAC_VCO_13700:
                        dac4822ABWrite(word12bits, 0,  MCP4822_CHANNEL_A_GAIN2);
                        break;

                case DAC_NOISE:
                        dac4822ABWrite(word12bits, 0, MCP4822_CHANNEL_B);
                        break;

                case DAC_V2140D_13700_SQU_LVL:
                case DAC_V2140D_IN3 :
                        dac4822ABWrite(word12bits, 1, MCP4822_CHANNEL_A);
                        break;

                case DAC_V2140D_13700_SUBBASS_LVL:
                case DAC_V2140D_IN4 :
                        dac4822ABWrite(word12bits, 1, MCP4822_CHANNEL_B);
                        break;

                case DAC_V2140D_3340_LVL:
                case DAC_V2140D_IN1 :
                        dac4822ABWrite(word12bits, 2, MCP4822_CHANNEL_A);
                        break;

                case DAC_V2140D_13700_TRI_LVL:
                case DAC_V2140D_IN2 :
                        dac4822ABWrite(word12bits, 2, MCP4822_CHANNEL_B);
                        break;

                case DAC_VCO_3340_FREQ :
                        //todo if (word12bits > VCO3340_MAX_INPUT_CV)
                        // then word12bits = VCO3340_MAX_INPUT_CV;
                        dac4822ABWrite(word12bits, 3, MCP4822_CHANNEL_A_GAIN2);
                        break;

                case DAC_VCO_3340_PWM_DUTY :
                        dac4822ABWrite(word12bits, 3, MCP4822_CHANNEL_B);
                        break;

                case DAC_V2140D_SH_LVL:
                case DAC_V2140D_IN7 :
                        dac4822ABWrite(word12bits, 4, MCP4822_CHANNEL_A);
                        break;

                case DAC_V2140D_VCA :
                case DAC_V2140D_IN8:
                        dac4822ABWrite(word12bits, 4, MCP4822_CHANNEL_B);
                        break;

                case DAC_V2140D_FM_LVL:
                case DAC_V2140D_IN5 :
                        dac4822ABWrite(word12bits, 5, MCP4822_CHANNEL_A);
                        break;
```

```
                case DAC_V2140D_RINGMOD_LVL:
                case DAC_V2140D_IN6 :
                        dac4822ABWrite(word12bits, 5, MCP4822_CHANNEL_B);
                        break;

                case DAC_VCF_CUTOFF:
                        dac4822ABWrite(word12bits, 6, MCP4822_CHANNEL_A);
                        break;

                case DAC_VCF_RES :
                        dac4822ABWrite(word12bits, 6, MCP4822_CHANNEL_B);
                        break;

                case DAC_EN_RABE_A:
                        dac4822ABWrite(word12bits, 7, MCP4822_CHANNEL_A);
                        break;

                case DAC_EN_RABE_B :
                        dac4822ABWrite(word12bits, 7, MCP4822_CHANNEL_B);
                        break;

        }
}
```

This function will write a 12 bits word onto the appropriate DAC using the Dac structure defined earlier. It consists in a simple switch case structure, in which we call dac4822ABWrite(). It adds an abstraction layer, allowing for easier programming later on.

muteAllDACs()

```
/**
 * Write default values to all DACs (to be done before starting htim1)
 * This ensures that the synthesizer is in the right state when starting
 */
void muteAllDACs(){
        dacVcaWrite(0.0); // makes sure we don't hear anything
        HAL_Delay(1); // wait 1ms for transfer to complete
                      //(could be lower but HAL_Delay can't go below)
        dacVcfCutoffWrite(0.0); // makes sure filter is off
        HAL_Delay(1);
}
```

This is quite self-explanatory, the VCA and VCF DACs are set to a reset state in order to properly initialize the synthesizer.

dacVcaWrite(double amplitude)

```
/**
 *  writes the given amplitude to the VCA control
 *  voltage through the appropriate DAC
 *  @param amplitude must be b/w 0 and 1
 */
void dacVcaWrite(double amplitude){
        if (amplitude < 0.0) amplitude = 0;
        else if (amplitude > 1.0) amplitude = 1.0;

        int i = (int)((1.0-amplitude) * 4095);
        /* 1-amplitude => because we have OpAmp
         *inverters somewhere in the path!
         */
        //int i = (int)(amplitude * 4095); // DEBUG
        dacWrite(i, DAC_V2140D_VCA);
        //dacWrite(4095, DAC_VCA);
}
```

Once again, this one is quite easy to understand. We only call dacWrite() with the appropriate arguments in order to write an amplitude value onto the DAC handling the VCA envelope. We also cast it from double to integer and format it so it works properly with the analog circuit.

dacVcfCutoffWrite(double cutoff)

```
/**
 *  write the given cutoff frequency to the VCF
 * control voltage through the appropriate DAC
 *  @param cutoff frequency value must be b/w 0 (min) and 1 (max)
 */
void dacVcfCutoffWrite(double cutoff){

        if (cutoff > 1.0) cutoff = 1.0;
        else if (cutoff < 0.0) cutoff = 0.0;
        int i = (int)((1.0-cutoff) * 4095);
        //int i = (int)(cutoff * 4095); // DEBUG
        /* TODO : actual cutoff CV is
         * GLOBAL CUTOFF + ENVELOPPE * EG_DEPTH parameter
         */
        dacWrite(i, DAC_VCF_CUTOFF);
}
```

This one works similarly to dacVcaWrite(), formatting a cutoff frequency and sending it to the appropriate DAC using previously coded functions. However, please not that the EG_DEPTH parameter is not yet implemented, but that the variables needed are already defined in header files.

HAL_SPI_TxCpltCallback(SPI_HandleTypeDef *hspi)

```
/**
 * Callback implementation for the SPI peripheral "end of transfer" interruption
 */
void HAL_SPI_TxCpltCallback(SPI_HandleTypeDef *hspi ){
        //__NOP();
        if (hspi == hspiDacs){ // Verify it is the right SPI bus
                //GPIOB->BSRR = (1<<1); //  Sets CS to 1 to validate
                HAL_GPIO_WritePin(DAC_CS_GPIO_Port, DAC_CS_Pin, GPIO_PIN_SET);
        }
}
```

Once data has successfully been transferred over to a DAC, this function is called by an interruption and resets the Chip Select Pin to its original state to validate the data transfer and ready the card for the next one.

Communication Protocol with RPi :

This section has been completed with the help of H. Peltzer, you will find more information in his section (chapter 4). However, we will still briefly describe the behavior of these functions :

- HAL_GPIO_EXTI_Callback()
- processIncomingMidiMessage()

Later on, we will explain the following functions, also involved in the processing of an incoming message from the RPi :

- midiNoteOnHandler()
- midiNoteOffHandler()

HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)

```
// called every time level changes on pin GPIO_Pin
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin){

        // RPi as note trigger
        if (GPIO_Pin == CE_RPi_Pin){ // CE0 from RaspBerryPi (PC6)

                //testCounter++;
                //HAL_SPI_Receive(hspiMidi,rx_spiMidi_buff,3,HAL_MAX_DELAY);
                // read three MIDI bytes in a row
                //Reactivate if connected to RPi
                //testCounter = rx_spi3_buff[0];
                //midiFromSpiMessageHandler(rx_spi3_buff[0]);
                //processIncomingMidiMessage(rx_spiMidi_buff[0],
                //      rx_spiMidi_buff[1],rx_spiMidi_buff[2]);
        }

        // button blue as note trigger
        else if (GPIO_Pin == USER_Btn_Pin){ // PC13

          blueButtonFlag = !blueButtonFlag;
          //blueButtonFlag = HAL_GPIO_ReadPin(USER_Btn_GPIO_Port,
          //                          USER_Btn_Pin);
          demoMode = blueButtonFlag;
          //HAL_GPIO_WritePin(GPIOB, LD3_Pin, demoMode == 1 ?
          //                          GPIO_PIN_SET : GPIO_PIN_RESET);

          else if (blueButtonFlag == 0){ // note OFF
                  //processIncomingMidiMessage(NOTE_OFF, 50, 100);
                  demoMode = 0;
          }*/
        }
}
```

This function is essentially called when a GPIO input with global interrupt is triggered. This could be the RPi, or the blue button for debug purposes. The code contains a lot of deprecated and debug lines, however what it has to do is to read the MIDI message stored on the SPI buffer by calling processIncomingMidiMessage(), giving as input arguments the three elements of the SPI buffer. This code can be modified at will to suit your debug needs.

processIncomingMidiMessage(uint8_t statusChannel, uint8_t data1, uint8_t data2)

```
/**
 * Generally called when a MIDI message of three
 * successive bytes has been received on the SPI bus,
 * @param status a MIDI status byte, e.g.,
 * MIDI CC or NOTE ON
 */
void processIncomingMidiMessage(uint8_t statusChannel,
                        uint8_t data1, uint8_t data2){

        int channel = statusChannel & 0x0F;
        int status = statusChannel & 0xF0;

        switch (status){
        case NOTE_ON :
                if (channel == DRUM_CHANNEL){
                        playDrumMachine(data1, data2);
                }
                else {
                        midiNote.note = data1;
                        midiNote.velocity = data2;
                        midiNoteOnHandler();
                }
                break;

        case NOTE_OFF :
                midiNoteOffHandler();
                break;

        case CONTROL_CHANGE:
                setMidiCCParam(data1, data2);
                break;
        }
}
```

This function will read the MIDI message it has received as arguments, and choose which function it has to call. It will use simple comparisons and masks with predefined constants in order to determine if it is a Control Change (calls setMidiCCParam()), or a Note On/Off event (calls midiNoteOnHandler() or midiNoteOffHandler() respectively). In the case of a Note On, it will verify if the channel number given as input is 10. If it is, it will call the playDrumMachine() function instead of midiNoteOnHandler().

midiNoteOnHandler()

```
/**
 *  Prepare the envelope state machines following a MIDI NOTE ONE message
 */
void midiNoteOnHandler(){

        // switch on LED so that we can monitor enveloppe level TODO : pwm !
        HAL_GPIO_WritePin(GPIOB, LD2_Pin, GPIO_PIN_SET);

        // ---------- VCA dyn parameters ----------
        stateMachineVca.amplitude=0.0;
        stateMachineVca.tmpTargetLevel = ((1.0-
            stateMachineVca.velocitySensitivity) + (midiNote.velocity/127.)
            *stateMachineVca.velocitySensitivity);
        stateMachineVca.tmpDelta = ADSR_TIMER_PERIOD_MS
            * stateMachineVca.tmpTargetLevel / vcaAdsr.attackTimeMs;
            // prepare dx for the attack phase of x(t)

        // ---------- VCF dyn parameters ----------

        double velocityMulFactor = ((1.0-
            stateMachineVcf.velocitySensitivity) + (midiNote.velocity/127.)
            *stateMachineVcf.velocitySensitivity);
        stateMachineVcf.t = 0;
        stateMachineVcf.tMax = vcfAdsr.attackTimeMs / (ADSR_TIMER_PERIOD_MS);
        stateMachineVcf.tmpTargetLevel = stateMachineVcf.envAmount
            * velocityMulFactor ;
        stateMachineVcf.tmpDelta = (stateMachineVcf.tmpTargetLevel
            -globalParams.vcfCutoff) / stateMachineVcf.tMax;
            // prepare dx for the A phase of x(t)
        stateMachineVcf.tmpKbdtrackingShiftFactor = (midiNote.note - 64)/64.0
            * stateMachineVcf.kbdTracking * MAX_KBD_TRACKING;
        stateMachineVcf.cutoffFrequency = globalParams.vcfCutoff;
            // starts at global cutoff value


        // prepare state machines:
        stateMachineVca.machineState=ATTACK; // force vca machine state to ATTACK
        stateMachineVcf.machineState=ATTACK; // force vcf machine state to ATTACK

        //midi_note.note += 10; // DEBUG
        //if (  midi_note.note >= 4095) midi_note.note = 0;
}
```

This function sets up several variables in order to trigger the state machine for the envelope generation.
At first, it only switches on a LED for debug purposes.
Then it will calculate the target level for the VCA by taking into account the velocity sensitivity parameter (and thus the velocity stored in the midiNote structure), and normalize the data. It will then calculate the step value by which the envelope has to increment on each iteration.
The same process will be carried out for the VCF, creating a variable velocityMulFactor which will be used to calculate how much the velocity will change the

cutoff frequency of the VCF. It will also initialize all the variables used to track time inside the state machine.
Finally, the state machine will be forced into its ATTACK state both for the VCA and VCF. Please refer to the appropriate section for a complete description of the state machine (section 7.4).

midiNoteOffHandler()

```
/**
 *  Prepare the enveloppes state machines following a MIDI NOTE ONE message
 */
void midiNoteOffHandler(){

        HAL_GPIO_WritePin(GPIOB, LD2_Pin, GPIO_PIN_RESET);

        stateMachineVca.tmpTargetLevel = 0.0;
        stateMachineVca.tmpDelta = - ADSR_TIMER_PERIOD_MS
            * stateMachineVca.amplitude / vcaAdsr.releaseTimeMs;
            // prepare dx for the R phase of x(t)

        stateMachineVcf.t=0;
        stateMachineVcf.tMax = vcfAdsr.releaseTimeMs / (ADSR_TIMER_PERIOD_MS);
        stateMachineVcf.tmpTargetLevel = globalParams.vcfCutoff;
        stateMachineVcf.tmpDelta = ADSR_TIMER_PERIOD_MS
            * (stateMachineVcf.tmpTargetLevel-stateMachineVcf.cutoffFrequency)
            / stateMachineVcf.tMax;
            // prepare dx for the R phase of x(t)

        stateMachineVca.machineState=RELEASE;
        // force vca machine state to RELEASE
        stateMachineVcf.machineState=RELEASE;
        // force vcf machine state to RELEASE
}
```

First, the LED that has been turned on by the midiNoteOnHandler() is switched off.
Then, since the Note Off signal will force the state machine into its RELEASE state, the function must set up the variables correctly. It will calculate the amplitude step for each envelope using the predefined time constants.
Finally, the state machine will be forced into its RELEASE state.

Control Change :

In this section, we go through the working of two functions :

- initSynthParams()
- setMidiCCParam()

initSynthParams()

```
/**
 * Initialize additionnal parameters and all DAC values
 */
void initSynthParams(){

        // wavetable init:
        //int i;
        /* for (i=0; i<WAVE_TABLE_LEN; i++) waveTable[i] =
         * 1000 * (1.0 + sin(0.0628 * i));
         */

        muteAllDACs();

        dacWrite((int)(2000), DAC_VCO_3340_FREQ);
        HAL_Delay(1); /* wait 1ms for transfer to complete
                       * (could be lower but HAL_Delay can't go below)
                       */
        dacWrite((int)(4095.0 * 2.0 * DEF_VCO_3340_PWM_DUTY),
                                        DAC_VCO_3340_PWM_DUTY);
        HAL_Delay(1);
        setMidiCCParam(WAVE_3340, 0);
        setMidiCCParam(SYNC_3340, 0);
        setMidiCCParam(VCF_ORDER, 0);

}
```

This function will initialize all parameters to their corresponding default values. The wavetable init part is not implemented, as we do not use the digital waveform in this project. This function will call setMidiCCParam(), whose behavior will be described right below.

setMidiCCParam(MidiCCParam param, uint8_t value)

```
/**
 * Updates the appropriate parameter of the ADSR enveloppec
 * @param value b/w 0 and 127
 */
void setMidiCCParam(MidiCCParam param, uint8_t value){

        switch (param){
        case VCA_ATTACK:
                vcaAdsr.attackTimeMs = ((value+1)/127.) * MAX_ATTACK_TIME;;
                //stateMachineVca.mulFactorAttack=exp(-1000.0*ADSR_TIMER_PERIOD
                //      /vcaAdsr.attackTimeMs); @deprecated exponential enveloppes
                break;

        case VCA_DECAY:
                vcaAdsr.decayTimeMs = ((value+1)/127.) * MAX_DECAY_TIME;
                //stateMachineVca.mulFactorDecay=exp(-1000.0*ADSR_TIMER_PERIOD
                //      /vcaAdsr.decayTimeMs); @deprecated exponential enveloppes
                break;

        case VCA_SUSTAIN:
                vcaAdsr.sustainLevel = (value/127.) * MAX_SUSTAIN_LVL;
                break;

        case VCA_RELEASE:
                vcaAdsr.releaseTimeMs = ((value+1)/127.) * MAX_RELEASE_TIME;
                //stateMachineVca.mulFactorRelease=exp(-1000.0*ADSR_TIMER_PERIOD
                //      /vcaAdsr.releaseTimeMs); @deprecated exponential enveloppes
                break;

        case VCF_ATTACK:
                vcfAdsr.attackTimeMs = ((value+1)/127.) * MAX_ATTACK_TIME;
                //stateMachineVcf.mulFactorAttack=exp(-1000.0*ADSR_TIMER_PERIOD
                //      /vcfAdsr.attackTimeMs); @deprecated exponential enveloppes
                break;

        case VCF_DECAY:
                vcfAdsr.decayTimeMs = ((value+1)/127.) * MAX_DECAY_TIME;
                //stateMachineVcf.mulFactorDecay=exp(-1000.0*ADSR_TIMER_PERIOD
                //      /vcfAdsr.decayTimeMs); @deprecated exponential enveloppes
                break;

        case VCF_SUSTAIN:
                vcfAdsr.sustainLevel = (value/127.) * MAX_SUSTAIN_LVL;
                break;

        case VCF_RELEASE:
                vcfAdsr.releaseTimeMs = ((value+1)/127.) * MAX_RELEASE_TIME;;
                //stateMachineVcf.mulFactorRelease=exp(-1000.0*ADSR_TIMER_PERIOD
                //      /vcfAdsr.releaseTimeMs); @deprecated exponential enveloppes
                break;

        case VCA_VELOCITY_SENSITIVITY:
                stateMachineVca.velocitySensitivity = (value/127.) * MAX_VC_SENSI;
                break;

        case VCF_VELOCITY_SENSITIVITY:
                stateMachineVcf.velocitySensitivity = (value/127.) * MAX_VC_SENSI;
                break;
```

```
case VCF_RESONANCE:
        dacWrite((int) 4095 * (value/127.) * MAX_MIXER, DAC_VCF_RES);
        break;

case VCF_CUTOFF:
        dacWrite((int) 4095 * (value/127.) * MAX_MIXER, DAC_VCF_CUTOFF);
        break;

case PWM_3340:
        dacWrite((int) 4095 * (value/127.), DAC_VCO_3340_PWM_DUTY);
break;

case SYNC_3340:
        HAL_GPIO_WritePin(SYNC_3340_GPIO_Port, SYNC_3340_Pin,
                        value==127 ? GPIO_PIN_SET:GPIO_PIN_RESET);
break;

case VCF_ORDER :
        if (value == 0){ // 2nd order
                HAL_GPIO_WritePin(VCF_4THORDER_GPIO_Port,
                                VCF_4THORDER_Pin, GPIO_PIN_RESET);
                HAL_GPIO_WritePin(VCF_2NDORDER_GPIO_Port,
                                VCF_2NDORDER_Pin, GPIO_PIN_SET);
        }
        else if (value ==1){ // 4th order
                HAL_GPIO_WritePin(VCF_2NDORDER_GPIO_Port,
                                VCF_2NDORDER_Pin, GPIO_PIN_RESET);
                HAL_GPIO_WritePin(VCF_4THORDER_GPIO_Port,
                                VCF_4THORDER_Pin, GPIO_PIN_SET);
        }
break;

case WAVE_3340 :
        if (value == 0){ // pulse
                HAL_GPIO_WritePin(TRI_3340_GPIO_Port,
                                TRI_3340_Pin, GPIO_PIN_RESET);
                HAL_GPIO_WritePin(SAW_3340_GPIO_Port,
                                SAW_3340_Pin, GPIO_PIN_RESET);
                HAL_GPIO_WritePin(PULSE_3340_GPIO_Port,
                                PULSE_3340_Pin, GPIO_PIN_SET);
        }
        else if (value == 1){ // triangle
                HAL_GPIO_WritePin(SAW_3340_GPIO_Port,
                                SAW_3340_Pin, GPIO_PIN_RESET);
                HAL_GPIO_WritePin(PULSE_3340_GPIO_Port,
                                PULSE_3340_Pin, GPIO_PIN_RESET);
                HAL_GPIO_WritePin(TRI_3340_GPIO_Port,
                                TRI_3340_Pin, GPIO_PIN_SET);
        }
        else if (value == 2){ // sawtooth
                HAL_GPIO_WritePin(TRI_3340_GPIO_Port,
                                TRI_3340_Pin, GPIO_PIN_RESET);
                HAL_GPIO_WritePin(PULSE_3340_GPIO_Port,
                                PULSE_3340_Pin, GPIO_PIN_RESET);
                HAL_GPIO_WritePin(SAW_3340_GPIO_Port,
                                SAW_3340_Pin, GPIO_PIN_SET);
        }
break;

case OCTAVE_3340 :
        vco3340.octave = value;
```

```
        break;

        case LEVEL_3340 :
                dacWrite((int) 4095 * (value/127.) * MAX_MIXER, DAC_V2140D_IN1);
        break;

        case DETUNE_13700 :
                vco13700.detune = value;
        break;

        case DETUNE_3340 :
                vco3340.detune = value;
        break;

        case LEVEL_13700 :
                // TODO : Voir comment implementer vis-a-vis des mixers
        break;

        case OCTAVE_13700 :
                vco13700.octave = value;
        break;

        case WAVE_13700 :
                dacWrite((int) 4095 * (value/127.)
                                * MAX_MIXER, DAC_V2140D_IN2);
                dacWrite((int) 4095 * (1 - value/127.)
                                * MAX_MIXER, DAC_V2140D_IN3);
        break;

        case VCF_KBDTRACKING :
                //TBA
        break;

        case VCA_EG :
                //TBA
        break;

        case VCF_EG :
                //TBA
        break;
        }
}
```

This function adds an important abstraction layer when a value has to be updated. The first argument param corresponds to a parameter in the enum MidiCCParam, which will have an explicit name. The value of the param, between 0 and 127, follows the MIDI standard.
A simple switch case structure will allow the function to modify only one parameter at a time, unsing the predefined constants to normalize and verify the validity of the parameter value. Then, depending on the parameter, it will either modify a variable (e.g VCA Sustain Level), write in a DAC (e.g PWM Duty Cycle), or write on GPIO Pins (e.g VCO 3340 Waveform). All the constants names are designed to ease code reading and modification.

ADSR State Machine :

The ADSR State Machine works by cycling between four states : ATTACK, DECAY, RELEASE and IDLE.
By default, the state machine will wait in the IDLE state until a note is pressed, receiving a NOTE ON message. The state machine will then proceed to the ATTACK state, then the DECAY state, and finally the RELEASE state, before returning to its original IDLE state. However, if a note is released during the cycle, the machine will be forced into its RELEASE state directly. The parameters used in each state are defined at the beginning of "dac_board.c" :

```
// ------------ adsr enveloppes ------------
int adsrInterruptCounter=0;

AdsrParams vcaAdsr = {
                .attackTimeMs = DEF_ATTACK_TIME,
                .decayTimeMs = DEF_DECAY_TIME,
                .releaseTimeMs = DEF_RELEASE_TIME,
                .sustainLevel = DEF_SUSTAIN_LVL
};
StateMachineVca stateMachineVca = {
                //.t = 0,
                .velocitySensitivity = DEF_VELOCITY_SENSITIVITY_VCA,
                .machineState=IDLE,
                .adsrParam = &vcaAdsr
};
AdsrParams vcfAdsr = {
                .attackTimeMs = DEF_ATTACK_TIME_VCF,
                .decayTimeMs = DEF_DECAY_TIME_VCF,
                .releaseTimeMs = DEF_RELEASE_TIME_VCF,
                .sustainLevel = DEF_SUSTAIN_LVL_VCF
};
StateMachineVcf stateMachineVcf = {
                .t = 0,
                .tMax=0,
                .velocitySensitivity = DEF_VELOCITY_SENSITIVITY_VCF,
                .kbdTracking = DEF_KBD_TRACKING,
                .envAmount = DEF_ENV_AMOUNT,
                .machineState=IDLE,
                .adsrParam = &vcfAdsr
};
GlobalSynthParams globalParams = { .vcfCutoff = DEF_CUTOFF,
                                   .vcfResonance=DEF_RESONANCE };
```

These parameters are created following the structures defined in "adsr.h" and initialized to their default values. They can be modified using the previously explained function setMidiCCParam().

The working principle of the state machine can be summarized here :

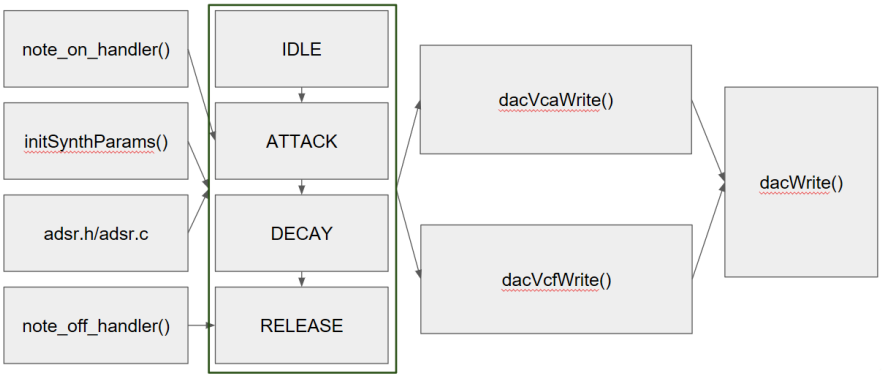


Figure 7.10: General Structure of the State Machine

The processing time of our STM32 is divided into 20 timer calls. Each TIM1 interrupt will trigger an update of a precise process. In our case, updating the VCA and VCF state machine takes two of these timer calls.

In this section, the following functions will be explained :

- HAL_TIM_PeriodElapsedCallback()
- updateVCAEnveloppeStateMachine()
- updateVCFEnveloppeStateMachine()

HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)

```
/**
 * Callback for the TIMER 1 "end of period" interruption
 * Currently PERIOD = 1/20kHz = 50us
 * - every 50us we push a new sample to the DIG VCO dac (sample freq = 20kHz)
 * - every 1ms = 20 * 50us (i.e. every 20 calls) we push a new sample to all
 * other enveloppes DAC (1kHz sample freq)
 * BUT since after each SPI bus write we must wait (around 16 * 1/3MHz = 5us)
 * for the transfer to complete before writing a new word,
 * it'd be a waste of time... so there's a clever trick that consists in
 * writing one distinct enveloppe at each timer call
 * (since there are 20 timer calls b/w every enveloppe update, we
 * could update up to 20 distinct enveloppes this way!
 * of course there are only 15 available DAC's on the board, so we do
 * nothing during the last 5 calls anyway)
 */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim){

        if (htim == htimEnveloppes){ // on check que c'est le bon TIMER

                HAL_GPIO_WritePin(GPIOB, LD3_Pin,
                                demoMode == 1 ? GPIO_PIN_SET : GPIO_PIN_RESET);


                //HAL_GPIO_TogglePin(GPIOB, LD3_Pin);
                //testCounter+=10;
                //if (testCounter > 4000) testCounter = 0;
                //dacWrite(testCounter, DAC_VCO_3340_FREQ);

                // always write one WAVETABLE sample
                //waveTableCounter++;
                //if (waveTableCounter == WAVE_TABLE_LEN) waveTableCounter = 0;
                //dacWrite(wave_table[waveTableCounter], DAC_VCO_DIG);

                // white noise for the drum machine :
                /* debug 2/4/19 int noise = (int)(4096.0 * rand() / RAND_MAX);
                dacWrite(noise, DAC_NOISE); */
                updateDrumMachine();



                // once every ADSR_TIMER_PERIOD, compute then write
                // all ADSR enveloppes + VCO CV's,
                // yet see trick above in this function documentation
                adsrInterruptCounter++;
                if (adsrInterruptCounter == ADSR_TIMER_PERIOD_FACTOR) {
                        adsrInterruptCounter = 0;
                        if (demoMode==1) playDemo();
                }

                // ALWAYS update VCO *before* VCA so that
                // we won't hear the note jump
                switch (adsrInterruptCounter){
                        case 0:
                                // first convert midi note to appropriate
                                // DAC voltage according to calibration:
                                //midiToVCO13700CV[midiNote.note];
                                //dacWrite(midiToVCO13700CV[midiNote.note + 12
                                //  * (octave_mul_13700 - 1)], DAC_VCO_13700);
```

```
                                dacWrite(midiToVCO13700CV[midiNote.note],
                                //          DAC_VCO_13700);
                                //dacWrite(testCounter, DAC_VCO_3340_FREQ);
                                //dacWrite(testCounter, DAC_VCF_CUTOFF);
                                break;
                        case 1:
                                //dac = midiToVCO3340CV[midiNote.note];
                                //dacWrite(midiToVCO3340CV[midi_note.note + 12
                                //  * (octave_mul_3340 - 1)], DAC_VCO_3340_FREQ);
                                dacWrite(midiToVCO3340CV[midiNote.note],
                                //        DAC_VCO_3340_FREQ);
                                //dacWrite(2000, DAC_VCO_3340_FREQ);
                                break;
                        case 2:
                                updateVCAEnveloppeStateMachine();
                                break;
                        case 3:
                                updateVCFEnveloppeStateMachine();

                                break;

                        // todo : mixers, etc
                }
        }
}
```

Although complex, the comments describing this function are explaining it quite well. This function will be called on each TIM1 interrupt. On every TIM1 interrupt, many instructions were set for debug purposes (e.g demo mode), however, some may still be implemented later (e.g white noise generation on an Analog GPIO, digital wavetable). You may also note that updateDrumMachine() is called on every iteration. The behavior of this function will be described in the next section.
After that, a counter will be incremented, allowing the function to choose between 20 processes to update. One will be updated for a given call, and 4 only are implemented for now. The first two calls will update the tone generated by the VCO, and the 3rd and 4th calls will update the VCA envelope and the VCF envelope, respectively.

updateVCAEnveloppeStateMachine()

```
/**
 * Updates the state machines associated with the generation of the VCA ADSR
 * enveloppes, then write it to the appropriate DAC
 * This method should be called from the timer handler (every ms or so)
 */
void updateVCAEnveloppeStateMachine(){

  switch (stateMachineVca.machineState){

          case IDLE:
                break;

          case ATTACK:
                dacVcaWrite(stateMachineVca.amplitude);
                stateMachineVca.amplitude += stateMachineVca.tmpDelta;
                //stateMachineVca.t++;
                if (stateMachineVca.amplitude >= stateMachineVca.tmpTargetLevel){
                // prepare dyn params for DECAY phase:
                          stateMachineVca.tmpTargetLevel *= vcaAdsr.sustainLevel;
                          // modulate sustain level with velocity factor
                      stateMachineVca.tmpDelta = ADSR_TIMER_PERIOD_MS *
                            (stateMachineVca.tmpTargetLevel
                                -stateMachineVca.amplitude)
                            / (vcaAdsr.decayTimeMs);
                            // prepare dx for the attack phase of x(t)
                          stateMachineVca.machineState = DECAY;
                }
                break;

          case DECAY:

                if (stateMachineVca.amplitude > stateMachineVca.tmpTargetLevel) {
                        stateMachineVca.amplitude += stateMachineVca.tmpDelta;
                        dacVcaWrite(stateMachineVca.amplitude);
                }
                // else stays on sustain plateau until NOTE OFF occurs
                break;

          case RELEASE :

                if (stateMachineVca.amplitude > 0.0) {
                // stateMachineVca.tmpTargetLevel)
                        stateMachineVca.amplitude += stateMachineVca.tmpDelta;
                        // else stays on sustain plateau until NOTE OFF occurs
                        dacVcaWrite(stateMachineVca.amplitude);
                }
                else stateMachineVca.machineState = IDLE;
                break;
  }

}
```

This is the function updating the ADSR envelope for the VCA. Depending on the state of the ADSR envelope, the function will behave differently :
Indeed, in the IDLE state, nothing will happen. We are waiting for a note input. In

the ATTACK state, we will increment the amplitude by a value delta, calculated before and taking into account the velocity sensitivity. If the amplitude value arrives above the predetermined ceiling, we will calculate how the envelope should behave in the DECAY state, and then force the machine in the DECAY state.
In DECAY state, the amplitude will be linearly decreased to a predetermined sustain value, and held at this value until an Note Off signal is received. Only then it will go into a RELEASE state.
In RELEASE state, the amplitude will gradually decrease to zero, following predetermined rules. Once it has reached zero, the machine will go to its IDLE state once again.

updateVCFEnveloppeStateMachine()

```
/**
 * Updates the state machines associated with the
 * generation of the VCF ADSR enveloppes,
 * then write it to the appropriate DAC
 */
void updateVCFEnveloppeStateMachine(){

  switch (stateMachineVca.machineState){

          case IDLE:
                  break;

          case ATTACK:
                  dacVcfCutoffWrite(stateMachineVcf.cutoffFrequency
                        + stateMachineVcf.tmpKbdtrackingShiftFactor);
                  stateMachineVcf.cutoffFrequency += stateMachineVcf.tmpDelta;
                  stateMachineVcf.t++;
                  if (stateMachineVcf.t >= stateMachineVcf.tMax){
                          // prepare dyn params for DECAY phase:
                          stateMachineVcf.t=0;
                          stateMachineVcf.tMax = vcfAdsr.decayTimeMs
                               / (ADSR_TIMER_PERIOD_MS);
                          stateMachineVcf.tmpTargetLevel = globalParams.vcfCutoff
                               + (stateMachineVcf.tmpTargetLevel
                               -globalParams.vcfCutoff)
                               *vcfAdsr.sustainLevel;
                                // modulate sustain level with velocity factor
                          stateMachineVcf.tmpDelta =
                             (stateMachineVcf.tmpTargetLevel-
                             stateMachineVcf.cutoffFrequency)
                             / (stateMachineVcf.tMax);
                             // prepare dx for the attack phase of x(t)
                          stateMachineVcf.machineState = DECAY;
                  }
                  break;

          case DECAY:
                  if (stateMachineVcf.t <= stateMachineVcf.tMax) {
                          stateMachineVcf.t++;
                          stateMachineVcf.cutoffFrequency +=
                                stateMachineVcf.tmpDelta;
                          dacVcfCutoffWrite(stateMachineVcf.cutoffFrequency
                                + stateMachineVcf.tmpKbdtrackingShiftFactor);
                  }
                  // else stays on sustain plateau until NOTE OFF occurs
                  break;

          case RELEASE :
                  stateMachineVcf.t++;
                  if (stateMachineVcf.t <= stateMachineVcf.tMax) {
                          stateMachineVcf.cutoffFrequency
                                += stateMachineVcf.tmpDelta;
                                // else stays on sustain plateau
                                // until NOTE OFF occurs
                          dacVcfCutoffWrite(stateMachineVcf.cutoffFrequency
                                + stateMachineVcf.tmpKbdtrackingShiftFactor);
                  }
                  else stateMachineVcf.machineState = IDLE;
```

```
                    break;
  }

}
```

This function works similarly to the previous one, the only difference being the time value that is being used to keep track of the state of the ADSR envelope, since we do not have target values. Otherwise, the parameters are modified following the same principle.

Drum Machine :

In this section, we will explain how we generated the trigger signals for the Drum Machine created by T. Poul (see chapter 13).

There are two functions handling this part :

- playDrumMachine()
- updateDrumMachine()

(uint8_t data1, uint8_t data2)

```
/**
 * Triggers a GPIO output to play a drum sound
 * @param data1, data2, last two bytes of midi message
 */

void playDrumMachine(uint8_t data1, uint8_t data2){
        switch(data1){
        case BASS_DRUM_NOTE:
                HAL_GPIO_WritePin(DRUM_KICK_GPIO_Port,
                    DRUM_KICK_Pin, GPIO_PIN_RESET);
                drumTriggers.bassdrumCounter = 1;
                break;
        case RIMSHOT_NOTE:
                HAL_GPIO_WritePin(DRUM_RIM_GPIO_Port,
                    DRUM_RIM_Pin, GPIO_PIN_RESET);
                drumTriggers.rimshotCounter = 1;
                break;
        case SNARE_NOTE:
                HAL_GPIO_WritePin(DRUM_SNARE_GPIO_Port,
                    DRUM_SNARE_Pin, GPIO_PIN_SET);
                drumTriggers.snareCounter = 1;
                break;
        case LOWTOM_NOTE:
                HAL_GPIO_WritePin(DRUM_LOWTOM_GPIO_Port,
                    DRUM_LOWTOM_Pin, GPIO_PIN_RESET);
                drumTriggers.lowtomCounter= 1;
                break;
        case HIGHTOM_NOTE:
                HAL_GPIO_WritePin(DRUM_HIGHTOM_GPIO_Port,
                    DRUM_HIGHTOM_Pin, GPIO_PIN_RESET);
                drumTriggers.hightomCounter = 1;
                break;
        }
}
```

This function will simply activate the right GPIO pin on the STM32 by checking the midi note value contained in data1. Note that all triggers are falling edges, except for the Snare Drum.

updateDrumMachine()

```
/**
 * called by timer to update trigger signal for drums
 */
void updateDrumMachine(){

        if (drumTriggers.bassdrumCounter >0) {
                drumTriggers.bassdrumCounter --;
                if(drumTriggers.bassdrumCounter ==0) {
                    HAL_GPIO_WritePin(DRUM_KICK_GPIO_Port,
                        DRUM_KICK_Pin, GPIO_PIN_SET);
                }
        }
        if (drumTriggers.rimshotCounter >0) {
                drumTriggers.rimshotCounter --;
                if(drumTriggers.rimshotCounter ==0) {
                    HAL_GPIO_WritePin(DRUM_RIM_GPIO_Port,
                        DRUM_RIM_Pin, GPIO_PIN_SET);
                }
        }
        if (drumTriggers.snareCounter >0) {
                drumTriggers.snareCounter --;
                if(drumTriggers.snareCounter ==0) {
                    HAL_GPIO_WritePin(DRUM_SNARE_GPIO_Port,
                        DRUM_SNARE_Pin, GPIO_PIN_RESET);
                }
        }
        if (drumTriggers.lowtomCounter>0) {
                drumTriggers.lowtomCounter--;
                if(drumTriggers.lowtomCounter==0) {
                    HAL_GPIO_WritePin(DRUM_LOWTOM_GPIO_Port,
                        DRUM_LOWTOM_Pin, GPIO_PIN_SET);
                }
        }
        if (drumTriggers.hightomCounter >0) {
                drumTriggers.hightomCounter --;
                if(drumTriggers.hightomCounter ==0) {
                    HAL_GPIO_WritePin(DRUM_HIGHTOM_GPIO_Port,
                        DRUM_HIGHTOM_Pin, GPIO_PIN_SET);
                }
        }
}
```

This function is called on every TIM1 interruption. Since the Drum Machine works better when giving short inputs, we will have to count only one short interruption since the Drum Machine trigger before resetting the signal. We decrease the counter that has previously been set up by one, and if we reach zero, we reset the GPIO pin to its original state.

7.5 Summary of the STM32 processing flow

The following chart summarizes the processing flow of the STM32F767ZI in its current state :

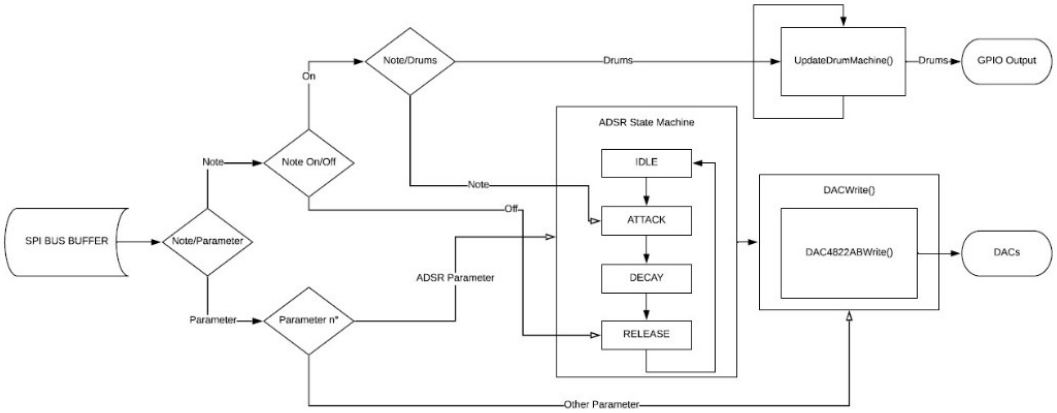


Figure 7.11: Processing Flow Chart

7.6 Bibliography

https://www.midi.org/specifications-old/item/table-1-summary-of-midi-message

https://www.st.com/en/microcontrollers-microprocessors/stm32f767zi.html

Chapter 8

Analog waveform mixers with V2164D Quad VCA (by Thomas Mazella)

8.1 V2164D IC

The V2164D IC is a quad-VCA circuit manufactured by CoolAudio Semiconductors. Each VCA is independant and offers a high efficiency and accuracy.
For the Themis synthesizer, we used two V2164D, summing up to 8 input signals:

- LM13700 Square wave
- LM13700 Triangle wave
- MCP23017 Output (one among the 3 outputs of the CEM3340)
- Sub-bass
- Ring Modulator
- Sample and Hold
- Drum Machine
- CEM3320 output, for envelope generation

8.2 Input signals

The V2164D IC can take up to four different signals through its input pins. These signals are current signals, as often used in audio applications. However, each VCO, as well as the subbass generator, output voltage signals. The conversion from voltage to current is achieved with a resistor in series with the input pin, as shown on **Figure 9.1**. The value of this resistor determines the constant gain of the conversion, according to Ohm's law.

8.3 Gain control

In order to control the gain of each VCA, it is required to apply a DC voltage to the corresponding Control Voltage (CV) input on the IC. The gain is logarithmic and follows this rule :

$$G = \frac{Output}{Input} = 30.3 * V_{input} \tag{8.1}$$

To cut it short, large positive values of control voltage will result in a high dampening whereas large negative voltages applied to the CV pin will amplify the signal. The gain range of the VCA is [-100dB ; +20dB], that is to say a voltage within the range [-0.6V ; 3.3V] applied to the CV terminal.
A 0V value on a CV pin will result with a gain of 0dB.

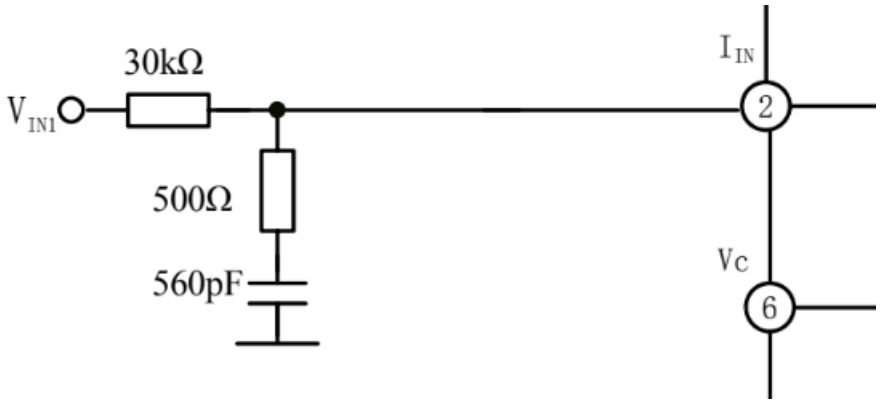


Figure 8.1: Signal conversion prior to IC input

8.4 Output signal

As the input signal, the output signal is under the form of a current flow. This means you can have multiple V2164D chained or operating in parallel to offer more inputs.
Every channel on a V2164D has its own output, meaning that output signals are not necessarily mixed altogether, as we used one of the channels to generate the envelopes on the mixed signal.

Chapter 9

CEM3340 VCO design (by Alexis Poumeyrol)

9.1 Features and characteristics

The CEM3340 is a voltage-controlled analog musical oscillator. It is expected to operate over a range of up to 8 octaves or more at relatively low (audio) frequencies and must be able to remain accurately tuned to this range in order to reproduce the chromatic scale sufficiently to meet the highly sensitive nature of human ear height.
Since a virtual synthesizer (VCO) must cover such a wide frequency range, it cannot be a natural resonant oscillator based on a crystal similar to that of radio circuits. It is generally a relaxation type oscillator whose frequency must be controlled by a precisely controlled current a control voltage from a keyboard.
CEM3340 generates sawtooth, triangle, square and adjustable pulse waveforms. It can be synchronized in hard or soft and can accept linear and exponential control voltages. We will have here a hard synchronization with the LM13700. A few external components are required to create a stable and sophisticated virtual synthesizer. It also replaces the wave shaping circuitry that other synths need after their purely sawtooth VCOs to create triangular and pulsed waveforms.
It is fully compensated to avoid temperature drift, no need to add a compensation resistance. Indeed, its integrated circuit has most of the VCO circuits. This contributes to the temperature stability and consistency of waveforms, an essential element in our polyphonic synthesizer. The VCO can be frequency modulated both linearly (carrier displacement to change notes) and exponentially (use to make effects).

9.2 requirements specification

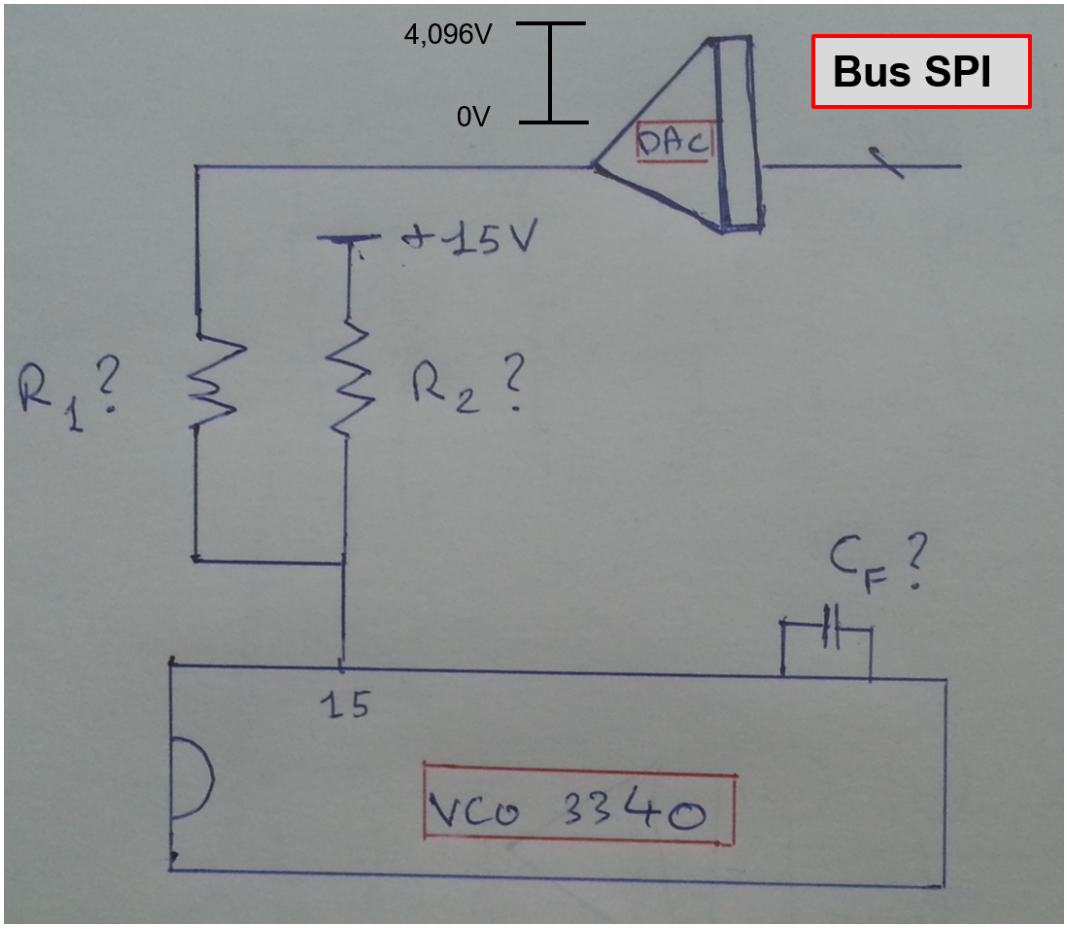


Figure 9.1: illustration of the requirements specification

My objective is to cover 6 octaves with an accuracy of one hundredth of a tone, and it is also necessary to dimension the resistors R1 and R2 connected to pin 15 of the VCO to support a DAC output amplitude of 0V to 4.096V with an accuracy of 1 mV. This will allow us to calculate the value and have the characteristics of the CF capacitor that determines the frequency of the circuit.

9.3 process

The first thing to do is to look in the frequency table to define our frequency range to cover the 6 octaves.

Note/octave	0	1	2	3	4	5
do ou *si*♯	32,70	65,41	130,81	261,63	523,25	1046,50
do♯ ou *ré*♭	34,65	69,30	138,59	277,18	554,37	1108,73
ré	36,71	73,42	146,83	293,66	587,33	1174,66
ré♯ ou *mi*♭	38,89	77,78	155,56	311,13	622,25	1244,51
mi ou *fa*♭	41,20	82,41	164,81	329,63	659,26	1318,51
fa ou *mi*♯	43,65	87,31	174,61	349,23	698,46	1396,91
fa♯ ou *sol*♭	46,25	92,50	185,00	369,99	739,99	1479,98
sol	49,00	98,00	196,00	392,00	783,99	1567,98
sol♯ ou *la*♭	51,91	103,83	207,65	415,30	830,61	1661,22
la	55,00	110,00	220,00	440,00	880,00	1760,00
la♯ ou *si*♭	58,27	116,54	233,08	466,16	932,33	1864,66
si ou *do*♭	61,74	123,47	246,94	493,88	987,77	1975,53

Figure 9.2: frequency table

Then the circuit is tested with the datasheet components on the test board: all you need is a +15V/-15V power supply. We put a control input (Vdac) with an arbitrary resistance (a little less than 100k) at the level of pocket 15 to determine our minimum and maximum currents. We place ourselves at a little more than 32.7Hz and a little less than 1.9kHz to have our minimum and maximum control voltage.

Output on pin 10 :

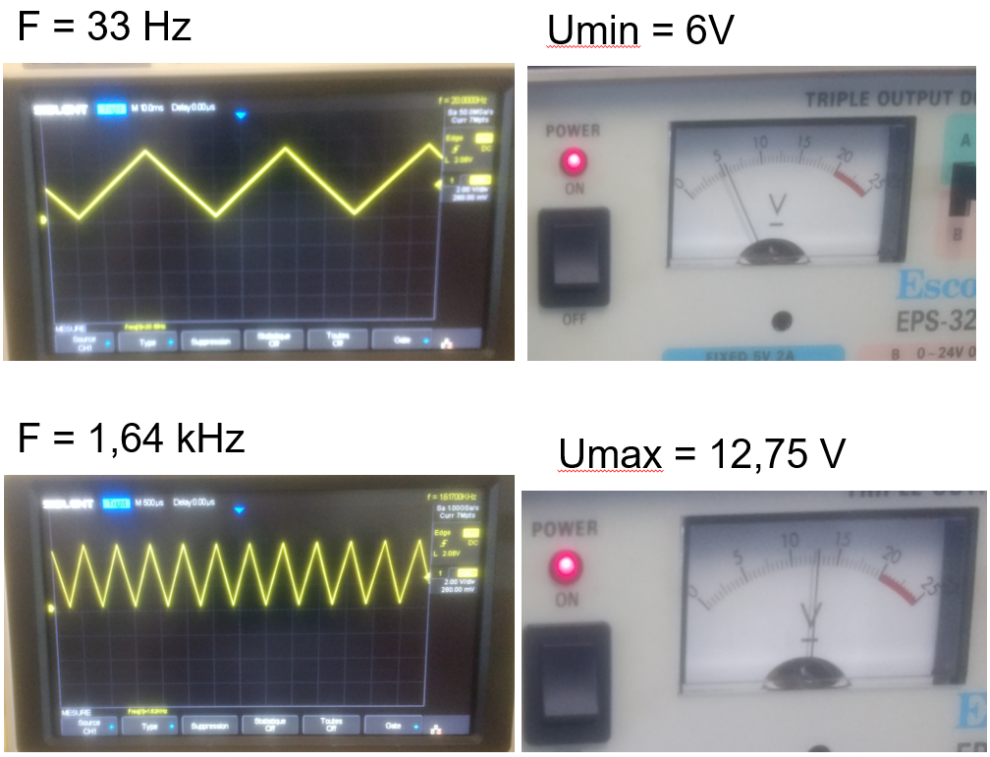


Figure 9.3: illustration of the min and max arbitrary voltages obtained

$$Imin = \frac{Umin}{R} = 60,9uA \qquad Imax = \frac{Umax}{R} = 128,8uA \tag{9.1}$$

The value of CF can then be determined with the formula :

$$f = \frac{3 * I}{2 * Vcc * CF} \tag{9.2}$$

If we take fmax and Imax, CF is 6.5nF. In our frequency range and to have a good temperature stability, we choose a ceramic capacitor.

Let's determine our resistances with Umin at 0V, Umax at 4.096V.

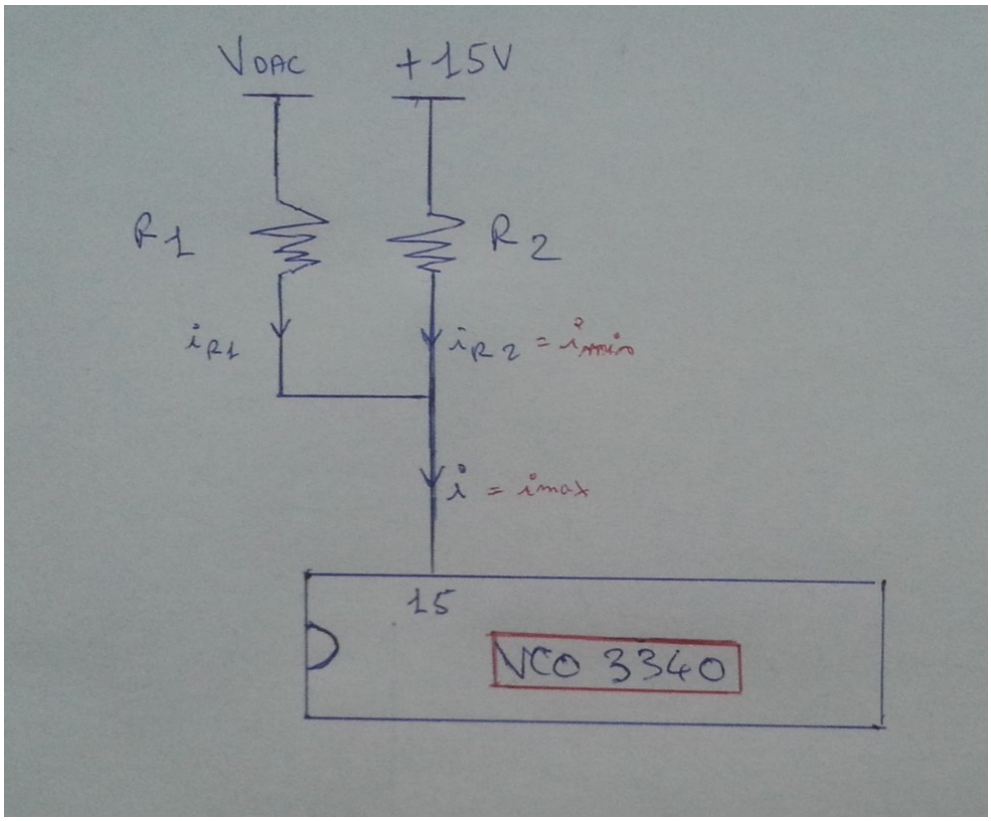


Figure 9.4: Diagram of currents pin 15

The Kirchhoff's law gives :

$$I = IR1 + IR2 = \frac{Vdac}{R1} + \frac{15}{R2} \tag{9.3}$$

Hence

$$Imin = \frac{15}{R2} \quad R2 = \frac{15}{Imin} \quad R2 = 60,36k \tag{9.4}$$

$$Imax = \frac{4,096}{R1} + Imin \quad R1 = \frac{4,096}{Imax - Imin} \quad R1 = 246,2k \tag{9.5}$$

9.4 Design

Here is the Eagle Design of CEM3340 with the 3 switch. I added a voltage divider at the square output to have an amplitude of 3.3V maximum so that the STM32 can connect.

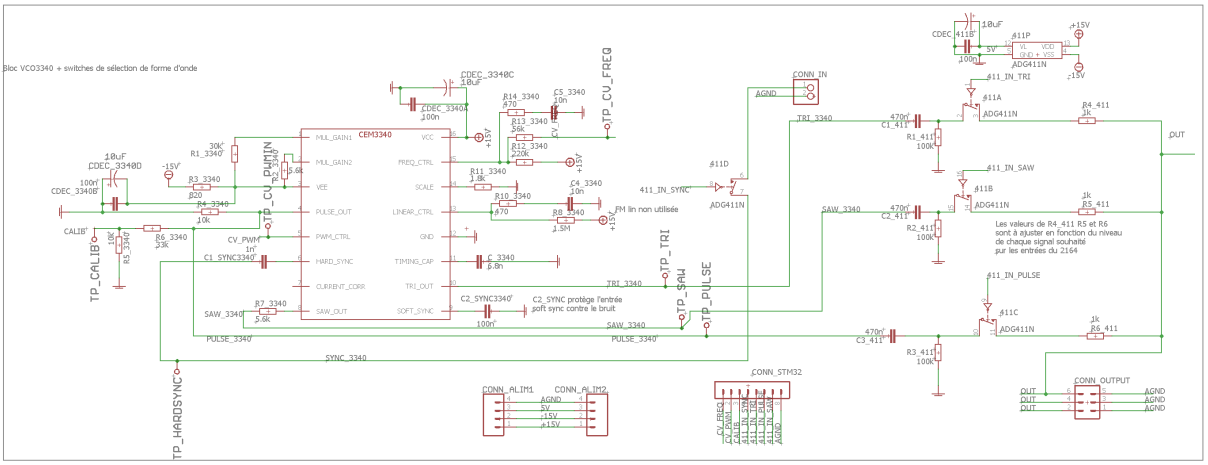


Figure 9.5: Eagle Schematic

Here is one way to connect the board.

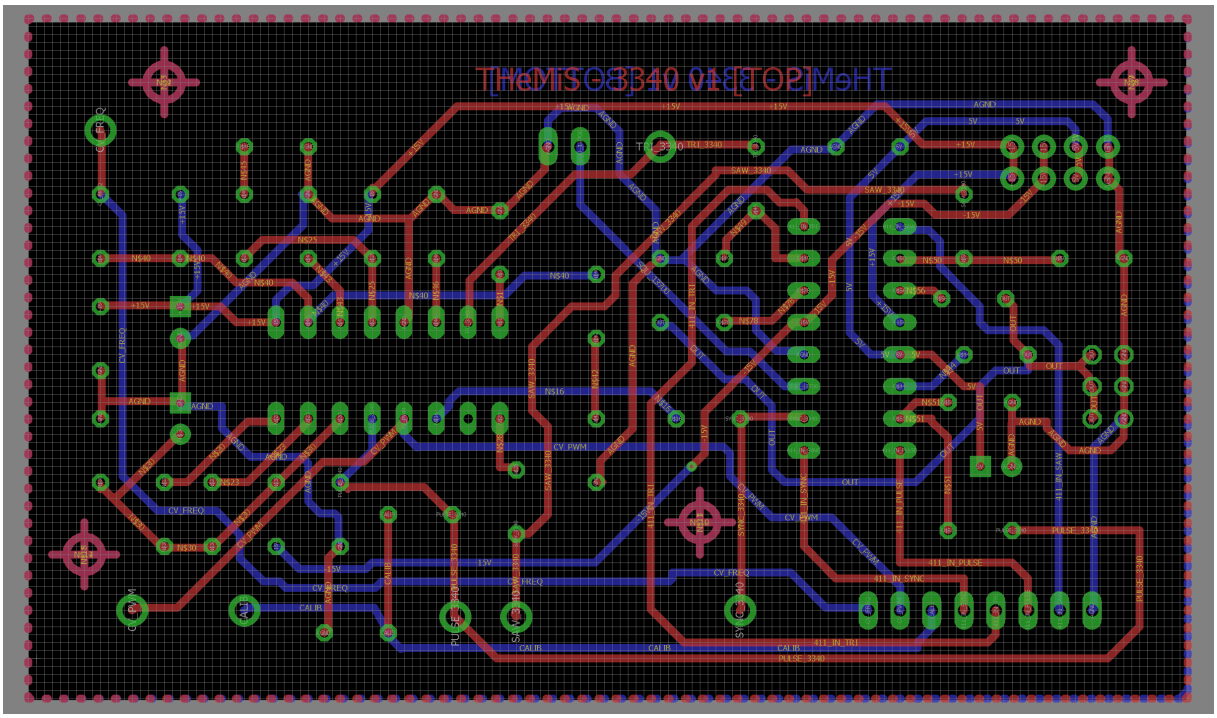


Figure 9.6: Realization of the board

9.5 The CEM3340 inside the THemis

Here is the place that the CEM3340 takes in the overall architecture of the synthetiser.

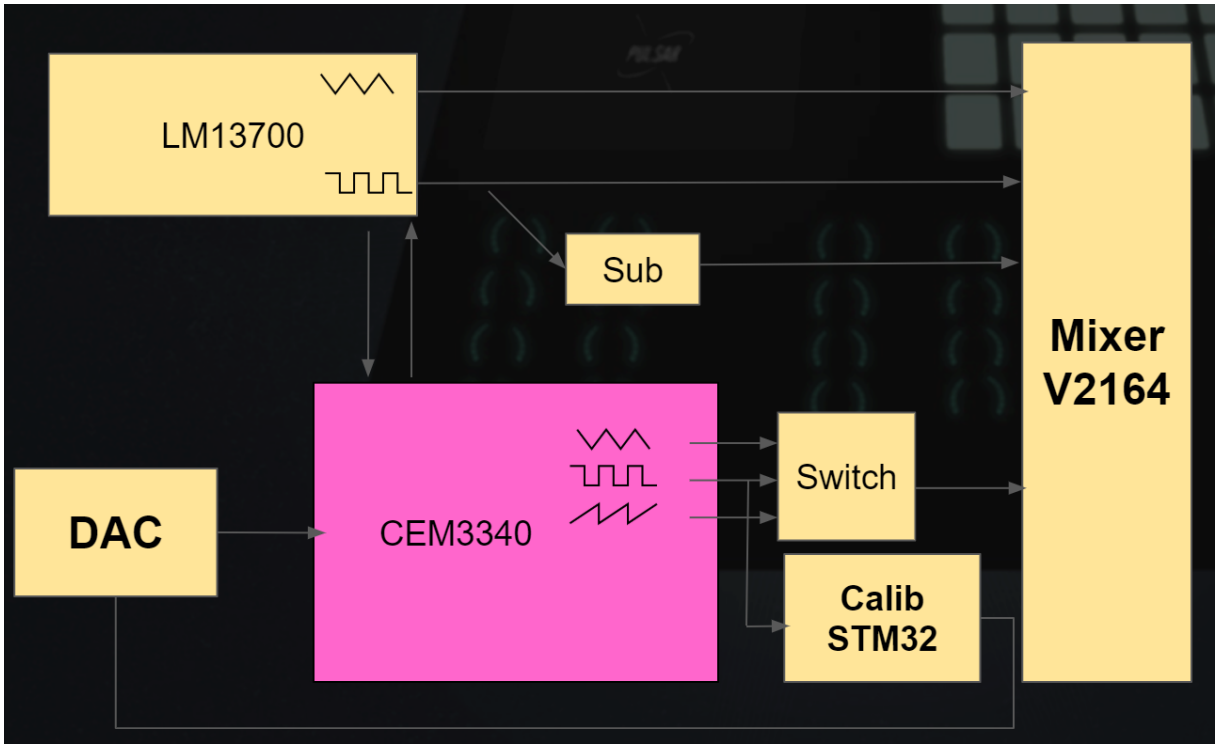


Figure 9.7: The CEM3340 inside the THemis

9.6 Pin Details

Pin 1 and 2 :
This is the temperature compensation circuit. This works by generating a temperature-dependent voltage which is then multiplied on-chip by the incoming CV. Since the VCO and the temperature-compensation circuit are all on the same die, they're all the same temperature, and this gives a highly accurate result.
Pin 3 :
Typical supplies for synth circuits would be +/-15V or +/-12V, but the CEM3340 can't stand more than 24V between its supply pins, so even +/-12V is right at the limit. Curtis dealt with this by adding a zener diode to limit the negative supply connected to pin 3 to -6.5V, so this gives 21.5V across the chip with +15V positive supply. The zener diode needs to be protected from over-current, so you typically see a current limiting resistor hanging off pin 3 (820R with -15V in the datasheet circuit). Alternatively, you can run the chip from +15/-5V supplies, in which case pin 3 can be directly connected.
Pin 4 :
It is the Pulse out. This pin can be used for the calibration of the VCO.
Initial Amplitude : 0V - 12V
Here : 0V - 3,3V
Pin 5 :
It is the PWM control. It serves as attenuator for PWM input or a pulse width control when no PWM is connected.
Pin 6 :
It is the Hard Synchronization input. This pin permits to synchronize the square of the LM13700 with the other wave forms of the CEM3340.
Pin 7 :
It is the High Frequency Tracking. This pin produces a current which can be used to compensate for the way that the time taken for the internal comparator to switch becomes significant at higher frequencies (5KHz or so). The lower half of the preset resistor serves as a resistor to ground which converts this current to a voltage, and the upper half and the series resistor feed a small portion back into the Frequency CV input at Pin 15. We don't use this pin here because we are not in High frequence.
Pin 8 :
It is the Sawtooth out. Amplitude : 0V - 10V
Pin 9 :
It is the Soft Synchronization input.
Pin 10 :
It is the Triangle out. Amplitude : 0V - 5V

Pin 11 :
It is the VCO's timing capacitor. Use a good quality capacitor here, with low leakage and low temperature. We use a ceramic one here. Mica is mentioned in the datasheet and there are other options now (C0G, for example).
Pin 12 :
It's the Ground of the circuit.
Pin 13 :
This pin is the Linear FM input, with an associated bias network.
Pin 14 :
This pin is used to adjust the exponential and linear control scales.
Pin 15 :
This pin is the Frequency CV input. This is a virtual ground summing node, and you usually see a bias network as shown (220K, 470R, 10n), although often with the values tweaked, followed by the summing resistors coming from the various CV sources.
Pin 16 :
This pin is the +15V.

Chapter 10

Power Supply (by Roland Giraud)

10.1 Context

Our synth as any electronic device needs a reliable and compact power supply. All the circuits and electronic cards of our synth are designed to work with +15/-15V or 5V that's why we need to design a power supply using the main supply 230V/50Hz and generating these three constant voltage output.

10.2 General design

We chose a very common yet effective design for this purpose

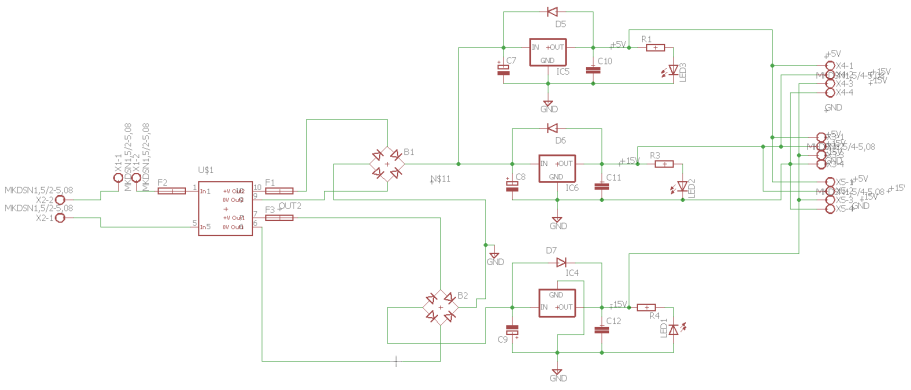


Figure 10.1: General design of the power supply

The main supply goes through the transformer; the tension is lowered then straighten by the diod bridge before getting into the voltage regulator. The LED

at the output of the regulators are meant to witness visually the proper functioning of our circuit, each regulated tension can be found on three different output for practical reasons.

10.3 Component choice

The Transformer is chosen according to the current consumption of our circuit, in our case we chose a transformer Myrra 230/2*18V with 30VA, it's better to use a molded transformer to avoid electromagnetic compatibility problems. Two secondary output are necessary because we want to create both positive and negative tension.
The output voltage at the secondary of the transformer is a 18V amplitude sinusoid, it is going through a diode bridge to straighten it then it is applied on the voltage regulators. The capacitors at the input of the regulators (C7, C8, C9) are meant to reduce the ripple of the straightened tension and must have a low impedance, we chose 2200uF.
The voltage regulators are standard lm7805,7815 and 7915 with 10nF decoupling capacitors (C10 C11, C12) and diodes (D5, D6, D7) to avoid return current.
The LED are chosen red, orange and green for aesthetic reasons and a standard 1kohms value is taken for the resistors (R1, R3, R4).
We chose to have three different output because lots of our circuits use the same power supply so it's more convenient for the wiring.

10.4 PCB design

Here is one possible way to connect your board

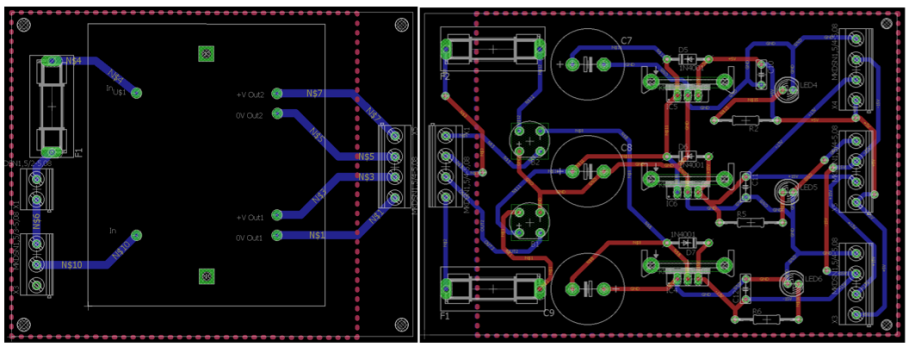


Figure 10.2: Realization of the board

The free version of eagle limits the size of your board so you might need to divide your design and reconnect it during the conception. Please notice that the alim output is not exactly the same as the alimentation input on the other analog board, this is because the alim was designed before the conception of these analog boards, no you might want to modify the order of the output pins.

Chapter 11

VCO LM13700 logarithmic converter (by Nathan Tricot)

11.1 Logarithmic converter

11.1.1 General view

The LM13700 provides a frequency signal proportional to its input current. But, the human ear is logarithmic: increasing a note by one octave is equivalent to doubling its frequency. Thus, if a note is obtained at a frequency corresponding to 50uA, the next octave will be 100uA, then 200, 400, 800, etc. However, the input current will be adjusted through a potentiometer, which will therefore have the same accuracy between 0 and 400 uA and between 400 and 800 uA. But while in the first case we travel up to 5 octaves, in the second, we travel only one. There is therefore a serious lack of precision in the bass.
The purpose of the logarithmic converter is as follows: if the human ear is logarithmic, the solution to the above problem is to create a circuit to convert a linear input voltage into a logarithmic output current for the LM13700. Thus, by varying the input voltage by a fixed value, we will increase linearly in the ranges, and therefore maintain the same accuracy in the bass as in the treble.
A second objective of this project is to find a solution to the previous problem while minimizing the influence of temperature on the circuit.

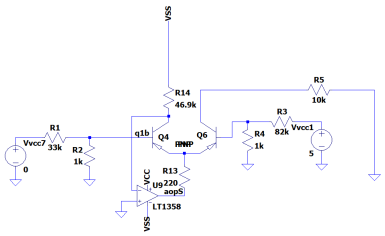


Figure 11.1: Logarithmic Converter Schematic

11.1.2 How does it work?

The logarithmic converter uses the exponential relationship between the base-emitter voltage of a transistor and the current flowing through it, according to the following relationship :

$$Ic = Is.[exp(\frac{q.Vbe}{k.T}) - 1]$$

While this relationship could provide us with a direct solution to the problem, we will seek to reduce temperature dependence. Indeed, we can see the presence of temperature in the above-mentioned relationship. In addition, in a transistor, the value of the coefficient in front of the exponential of the formula can double for a temperature difference of only 10 degrees. However, a transistor can heat much more, up to a maximum of 100 degrees. A solution must therefore be found to reduce these variations.
To do this, a transistor current mirror coupled to an amplifier is used to fix the current of one of the two transistors. This eliminates first-order temperature dependence. For optimization purposes, an offset has also been added on one of the transistors to reduce the gaps between theory and practice. Finally, for an optimal adjustment of the VCO bandwidth, the current entering the first transistor was set to -33mA. The global structure of the logarithmic converter is in figure 11.1.

11.1.3 Inserting the logarithmic converter into the board

The logarithmic converter has 6 pins: one input, one output, one ground, -15V, +15V, and +5V. The log. conv. connects between the VCO control potentiometer and the VCO (which has the effect of short-circuiting the operational amplifier present, but it is always possible to add it afterwards). The circuit, once made, is not very big (figure 11.2)

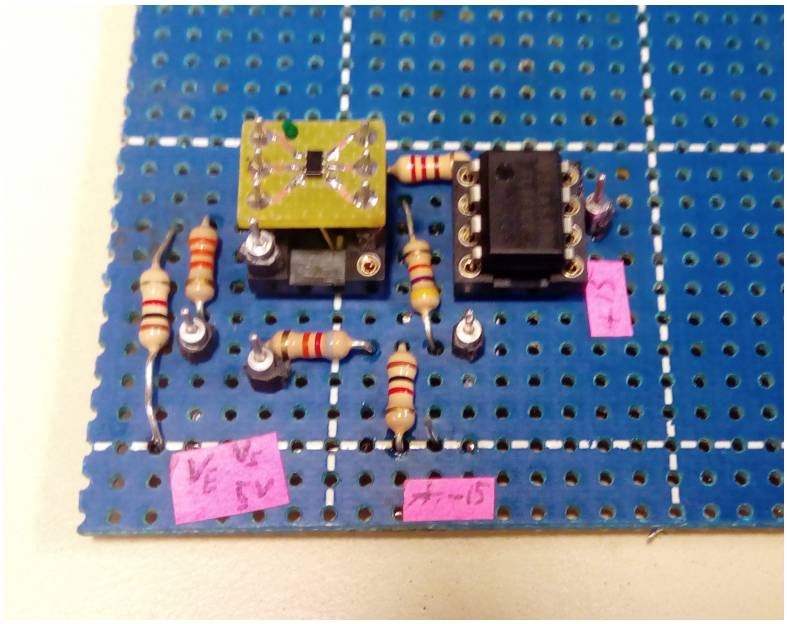


Figure 11.2: Logarithmic Converter

11.2 2. Creating a PCB for the logarithmic converter, the LM13700 and the subbass

The conception team decided to put on the same board the logarithmic converter, the LM13700 and the subbass. Indeed, as these three circuits work together, grouping them will save space inside the synthesizer.
The conception of the PCB led us to put some coupling capacitors. We also needed to put several test pins, supplying pins and input/output pins.
Another constraint we faced was that the differential pair of transistors is on a CMS integrated circuit. We decided to directly weld it directly on the board, but it imposed some constraints. Indeed, at ENSEA, there is a limitation to the minimum track size of a PCB, and this minimum size was greater than the gap between the legs of the CMS. It was therefore necessary to override this minimum size, to finally obtain a fairly satisfactory result. Another problem with the CMS component is that if the integrated circuit malfunctions, it must be desoldered directly on the board (which we have had to do several times).
Obviously, even if theoretically everything was supposed to work, at first nothing worked. But the creation of test bench on each floor made it possible to very quickly determine that the origin of the problem was the logarithmic converter.
The end of the project consisted in debugging the logarithmic converter, in particular by testing all welds and changing the resistance values. In the end, we

realized that the printed circuit board was not the right one, so the transistors were not connected correctly. After replacing this circuit with the right transistors (BC857BS), the circuit was functional. It was then simply a matter of changing the appropriate resistors to obtain the desired frequency range.

Chapter 12

Subbass oscillator (by Tristan Poul)

12.1 Context of use

The Subbass oscillator is a voltage-controlled analog musical oscillator. It is expected to generate bass frequencies. You can see below a global schematic which introduce this module.

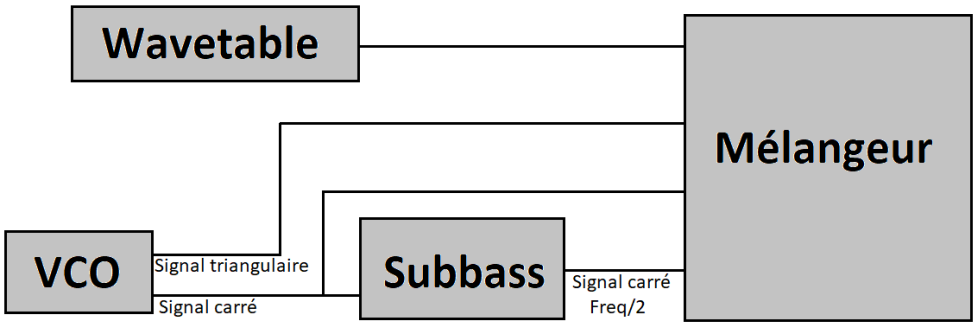


Figure 12.1: Global schematic

Subbass output is connected to one of the four inputs of the analog waveform mixers and subbass input is connected to the square output from VCO13700. The difference between Subbass in and out is the signal frequency, divided by 2. This is the method used to generate bass frequencies on the audio signal. One wonders why divided by 2 the signal (and not 3, 4 or 5 for example). When you divided (or multiplicated) by 2 (or a power of 2) the frequency of an audio signal you obtain a signal at the same tone but a different octave. This module allows the synthesizer to modulate huge bass sound, often used in electronic music.

12.2 How to build a Subbass oscillator

As you can well imagine, this module is simply a frequency divider. To build this function we use a simple D flipflop or JK flipflop. You can see below the electronic schematic of the circuit, and it chronogram.

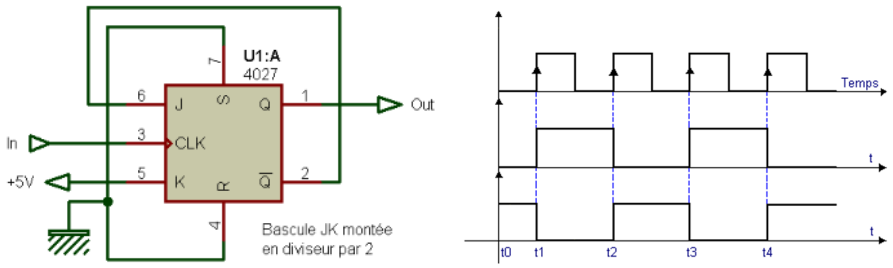


Figure 12.2: Theoretical Subbass circuit and chronogram

In practical, to build this circuit we use a HEF4027B integrated circuit. This circuit contain two JK flipflop (we use just one of this). You can see below the electronic schematic of the circuit.

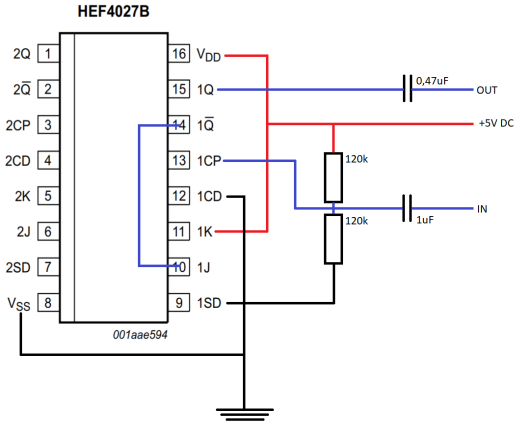
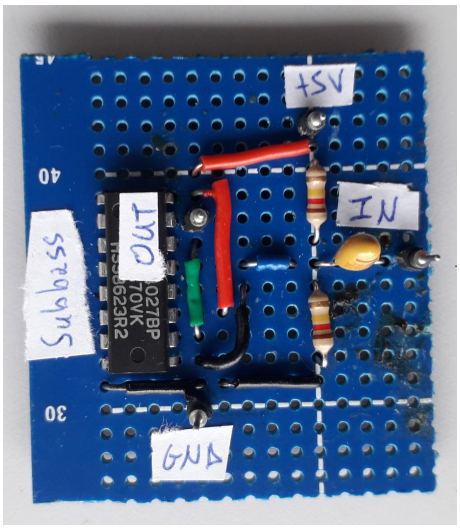


Figure 12.3: Practical Subbass circuit

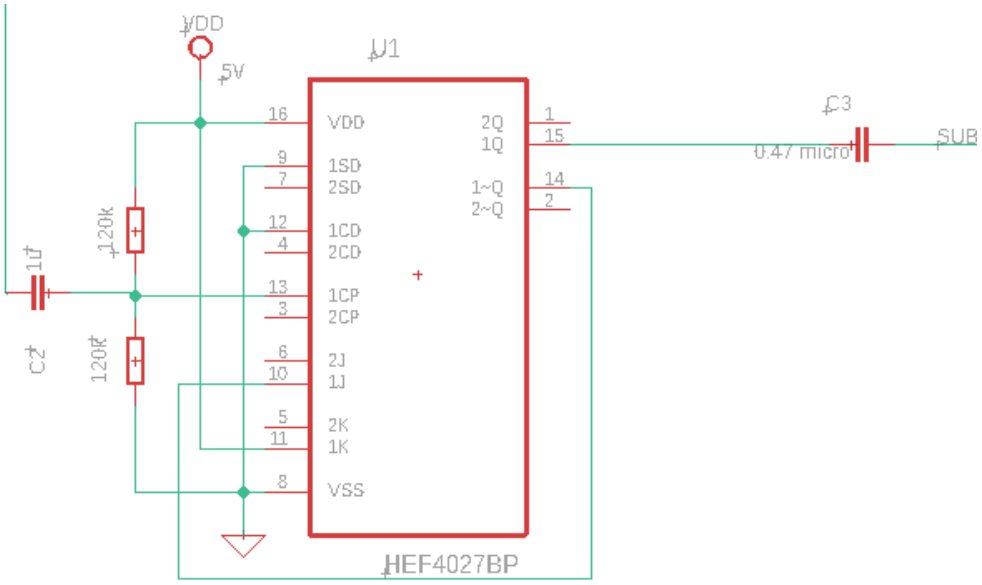
It can be noted some differences between the theoretical and the practical schematic. The components in input allow to create an offset on the signal and the components in output allow to recenter the signal on zero.
We understand the module interest. With a very small and affordable circuit, we allow the user to modulate his synthesizer sound with a new parameter.

12.3 Practical implementation

First step Prototype



Second step EAGLE schematic



Chapter 13

Drum Machine (by Tristan Poul)

13.1 Introduction

The drum machine is an analog module which allows to create percussive sounds with electronic components. This percussive sounds imitate the acoustic sounds create by a drum. This part talks about the process used to generate five elements of a drum, a kick, a low tom, a high tom, a rim shot and a snare. The electronic schematics presented in this report are all inspired by the TR808 from Roland, a very famous drum machine whose service manual is completely available on internet.
`https://archive.org/details/synthmanual-roland-tr-808-service-notes`

The choice to imitate the sound generated by the TR808 was did for different reasons. First, the documentation of this device is full available on internet, so it allow us to quickly understand the operating principle of the basic electronic circuits, and secondly the sounds generated by the TR808 are really references in terms of electronic percussive sounds (big fat sound, not like acoustic drum sounds, but really useful in electronic music, like techno for example).

13.2 Main percussive circuit

Page 5 of the TR-808 Service Manual gives the schematic and formula for the bridged t-network. This bridged allow us to create a waveform of a percussive sound, it's was the main circuit used in the conception of the drum machine.

You can see below a schematic of this circuit.

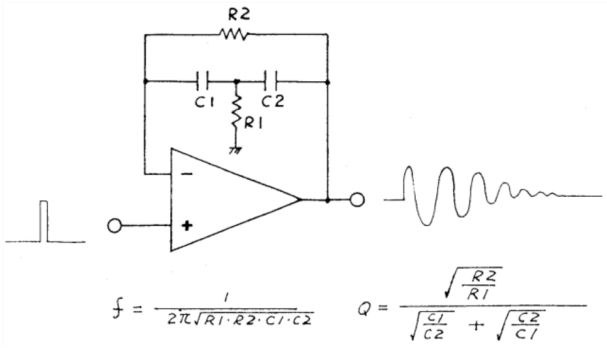


Figure 13.1: Representative Bridged T-network (TR-808 Service Manual)

As you can see, the circuit consists of two resistors and two capacitors, all of which combine to determine the properties of the sound: the pitch, resonance, and decay. Changing any one of the values effects all three of the properties. Some rudimentary tuning of the pitch can be obtained by replacing R1 with a trimmer. In the TR-808, there are quite a few additional components in the circuitry of each of the instruments that enable their pitch and/or decay to be tuned.

With just this circuit, we had created four elements of a drum, the kick, the low tom, the high tom and the rim shot.

13.3 Snare circuit

A snare drum is really two sounds: the sound of the drum head being hit and the sound of the snare wires vibrating against the bottom resonant head of the drum.You can see below the schematic circuit used by the TR808.

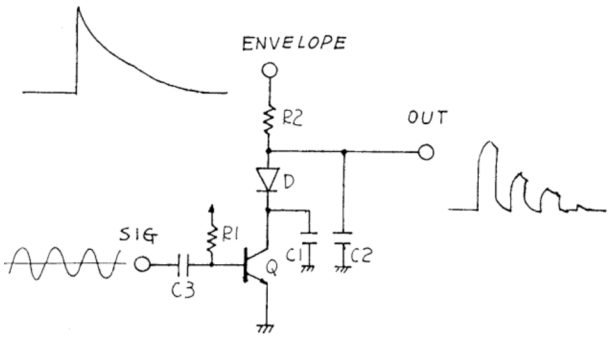


Figure 13.2: Representative Swing type VCA (TR-808 Service Manual)

The circuit for the snare drum contains the same bridged t-network used by the bass and toms, but instead of triggering it with a simple pulse, the snare drum is triggered with a burst of white noise. The white noise is first passively low pass filtered by C3 and R1. Then it is sent through a single transistor VCA (voltage controlled amplifier), a circuit referred to by the Roland engineers as a ”Swing Type VCA.” The VCA envelope is provided by a pulse passing through a very simple RC circuit.

The method of exciting the bridged t-network with white noise saves a lot of components when compared with the alternative method of separating the circuitry for the drum head and snare wires, triggering them both, then mixing their outputs.

13.4 Practice implementation

Five different elements was made with the circuits shown before. We made a kick, a low tom, a high tom and a rim-shot just with the T-network by change the value of the resistor and capacitor. You can see bellow the four different EAGLE schematic created.

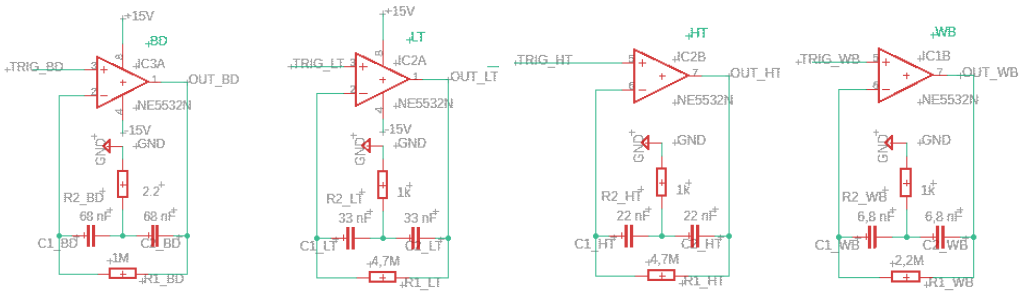


Figure 13.3: EAGLE Schematic : Kick module, Low Tom module, High Tom module and Rim-Shot module

The only difference between this modules is the values of the resistor and capacitor, for example the value of the kick module allows us to create a kick sound to a frequency of 80 Hz. All this module are triggered by an inverse pulse of 1ms (+3.3V to 0V) from the STM32.

The EAGLE schematic you can see bellow is the module used to generate snare sound :

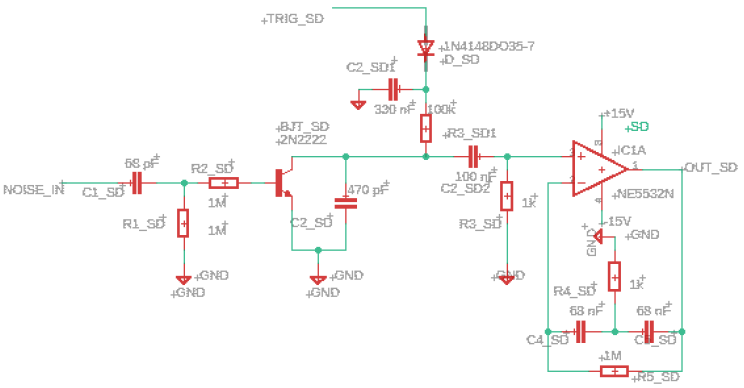


Figure 13.4: EAGLE Schematic : Snare module

The snare module is a combination of swing type VCA, triggered by an impulse of 1ms (0V to 3.3V) at the diode input (to create an envelop) and a white noise at the filter input, the twice signals are combined to the input of a Bridged T-network. Like with the others elements the input and triggered signals are created by the STM32.

All the output of the sub-modules presented before go in the circuit you can see bellow.

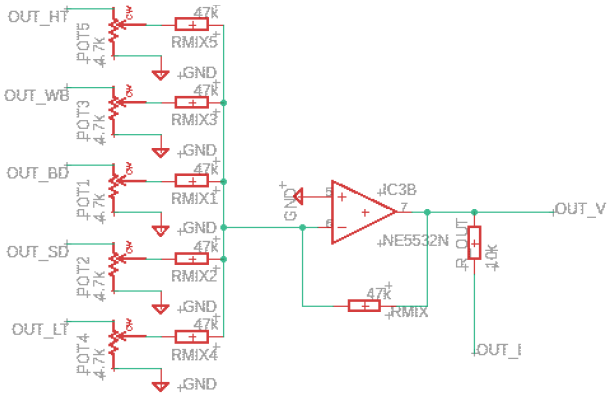


Figure 13.5: EAGLE Schematic : output stage

The output stage allows different things. First it allows us to have two different types of audio output, current output or voltage output. Secondly it allows us to control the amplitude of each sub-modules by a trimmer, in practice this trimmers are buttons on the board of Themis synthetizer.

Chapter 14

Sample and Hold (by Roland Giraud)

14.1 Description

A sample and hold circuit is an analog device that samples the voltage of a continuously varying analog signal and holds its value at a constant level for a specified minimum period of time. Using this device we can create a very specific sound modulation

14.2 operation mode

To realize it's modulation the sample and hold device needs two different inputs signals, one is the signal which will be modulated and the other is the synchronous signal which give the modulation rate.

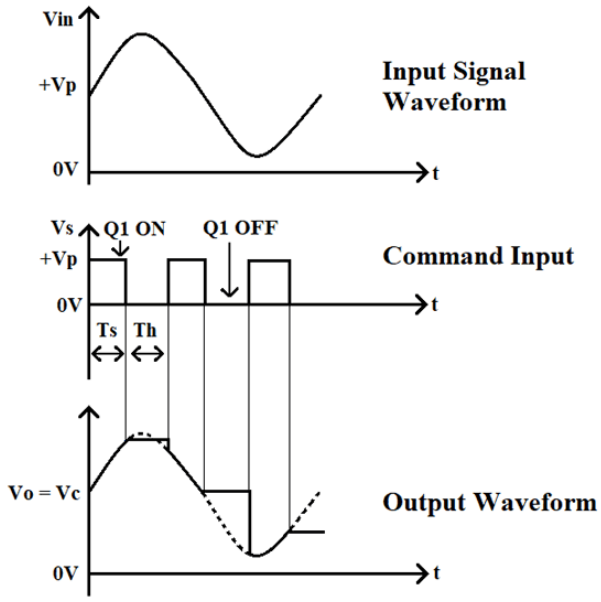


Figure 14.1: Working principle S and H

14.3 Implementation

We use the LF398 device it features supply voltage of +/- 5 to 18V, a low output noise and a differential logic threshold of 1.4V which fits our requirements.

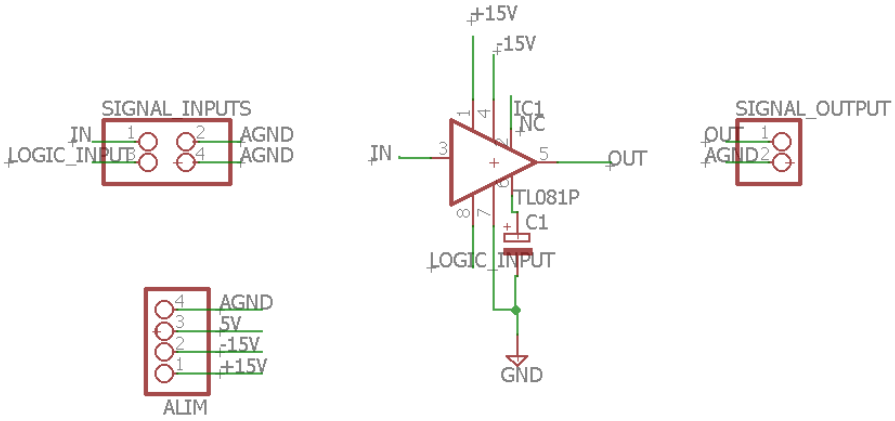


Figure 14.2: EAGLE design

A standard 10nF capacitor is taken for Ch

14.4 Use and observations

- The inputs signals can be taken from the VCO outputs or the STM
- the input signal waveform can be either a square, a sinus or a triangle
- the waveform of the logic input doesn't matter as long as it's periodic and has a maximum of amplitude superior than the threshold voltage, modify the waveform is equivalent to modify the duty cycle of the signal
- the most interesting sounds are noticeable when the frequency of the Logic signal is slightly different of the frequency of the analog signal (few Hertz)
- You can also set the frequency of one of the signal as a multiple of the other and do the same operation.
- LF398 is quite robust and can resist input signals of 10V amplitudes

Chapter 15

Ring modulator (by Mathieu Tissandier)

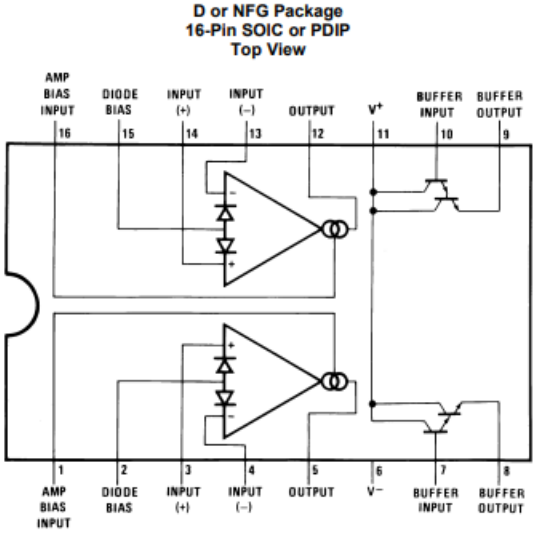
15.1 Ring modulator characteristics

The Ring modulator is an electronic device very used in music synthesizers to produce a sound with richer harmonics. It requires an amplitude modulation, between a sin wave signal and a simple waveform like square waveform or saw smooth signal thus the circuit executes a multiplication operation between these two signals. The word ”Ring” comes from the original ring shape of diode multipliers. This electronic device could also be used as effect unit.
the multiplication of two signals will produce signals with the sum and difference of the input frequencies. When the frequencies are linked together, the sound still adheres to the partial harmonics. But if the frequencies are not linked together it will produce inharmonics which will give a metallic sound. This circuit allows you to create really rich sounds.

15.2 The LM13700 as a multiplier

15.2.1 The LM13700 circuit

The LM13700 series consist in two current-controlled transconductance amplifiers, each with differential inputs and a push pull output. The two amplifiers share common supplies but otherwise operate independently. High impedance are provided which are especially designed to complement the dynamic range of amplifiers. The out put buffer’s input bias current are independent of Iabc.Here are the main characteristics of the LM13700.



Pin Functions

PIN		I/O	DESCRIPTION
NAME	NO.		
Amp bias input	1, 16	A	Current bias input
Buffer input	7, 10	A	Buffer amplifier input
Buffer output	8, 9	A	Buffer amplifier output
Diode bias	2, 15	A	Linearizing diode bias input
Input+	3, 14	A	Positive input
Input–	4, 13	A	Negative input
Output	5, 12	A	Unbuffered output
V^+	11	P	Positive power supply
V^-	6	P	Negative power supply

Figure 15.1: Main characteristics

Note here the presence of the two output buffers that allow to isolate the low output impedance circuit with the potentially following circuits. The circuit also needs to be alimented with a -15V/+15V alimentation.

15.2.2 From the LM13700 to the multiplier

As mentionned above, the LM13700 uses 2 differentials OTA(operational transconductance amplifiers), that's to say, the output current is an image of the difference of the inputs voltages. The output current and the input voltages are linked with the transconductance factor gm. Moreover the gm factor directly include the Iabc (amplifier bias current), thus this one will change the value of gm, this particular property of the OTA will make us able to create a amplitude

modulation.Here are the equations of the output and the gm transductance factor :

$$Io = gm(V + -V-) \tag{15.1}$$

VT=25mV

$$gm = \frac{Iabc}{2VT} \tag{15.2}$$

So, in theory with the rights values of V+,V- and Iabc we can obtain the multiplication of the 2 signals, we just have to convert the voltage in Iabc current.It's also interesting to notice the circuit's bandwidth which is 2MHz, so it's very convenient to use for audio frequencies. The following schematic illustrates how the OTA works :

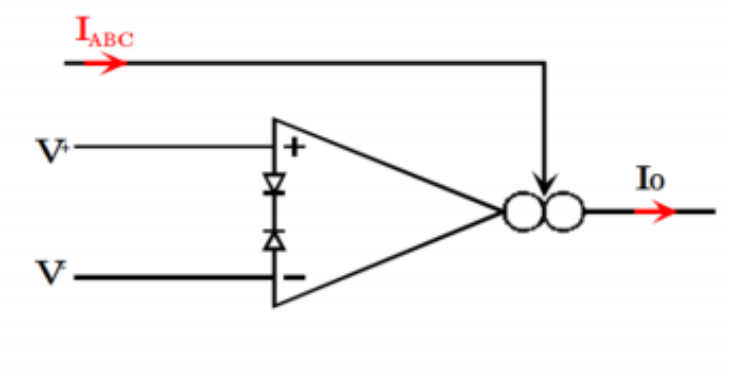


Figure 4.1: Symbole/Modèle d'un OTA

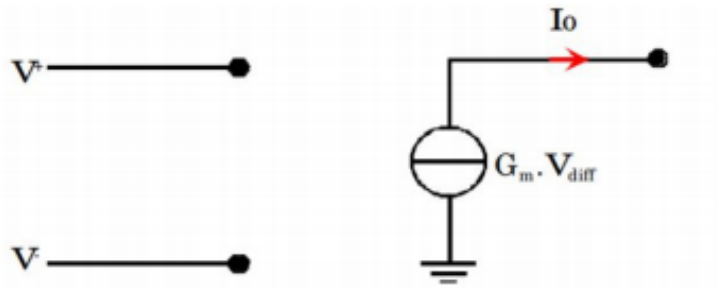


Figure 15.2: Operating principle of the OTA

15.2.3 Design of the multiplier based on LM13700

In order to design the multiplier circuit, we have chosen the following basic's schematic and adapt the value of each parameters to obtain the AM modulation to finally have the ring modulator effect :

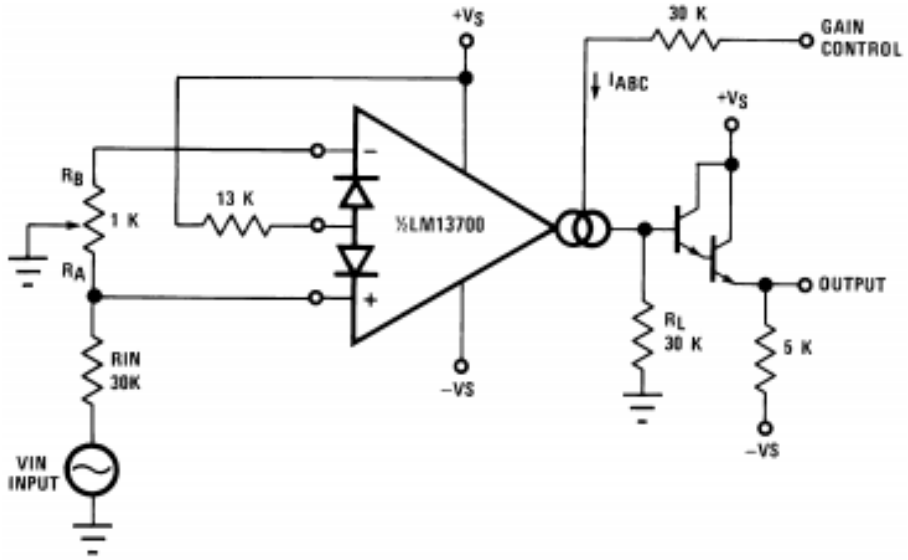


Figure 15.3: Basic schematic

During the first time, we have choosen closed values of resistor and then we added capacitors to cut the DC component, between the VCOs and the ring modulator (100nF we tested this value and it works well to isolate the circuit). Then we designed the circuit on Eagle :

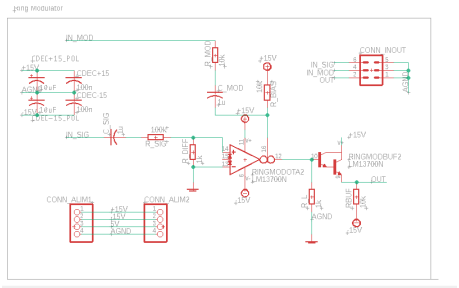


Figure 15.4: Eagle's schematics of the Ring modulator

Then we had to design the PCB of the Ring modulator,according to the requirements of Eagle free version and the specifications of the school about the design of PCB. In order to have a common power supply basis we added the same power supply connectors than the other cards, it would be more convenient to link power supplies with each others, here is a way to design the PCB :

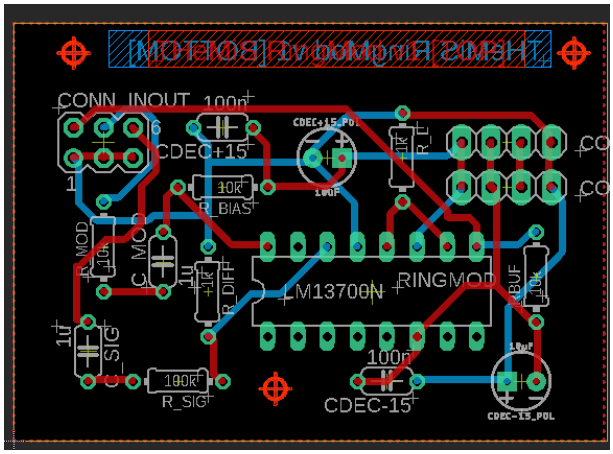


Figure 15.5: Eagle's PCB of the Ring modulator

We added decoupling capacitor's between the circuit and the power supply thus the circuit can be well isolated and protected against high harmonics.

Here we have the circuit once finished :

Figure 15.6: Final prototype of the Ring modulator

We will describe the Pin's detail more precisely in the next section untitled section "PIN's details" in order to precise more practical details for the commissioning of the Ring modulator card.

15.3 The Ring modulator inside the Themis

As we said previously, the Ring modulator needs 2 signals in order to multiply them with each others, thereby we will use the output of the LM13700 VCO, whose output are two signals : square and triangle. The output of the Ring modulator is the mixed signal, it will be linked with one of the input of the mixer. The following schematic sums up the Ring modulator's role inside the global Themis's architecture :

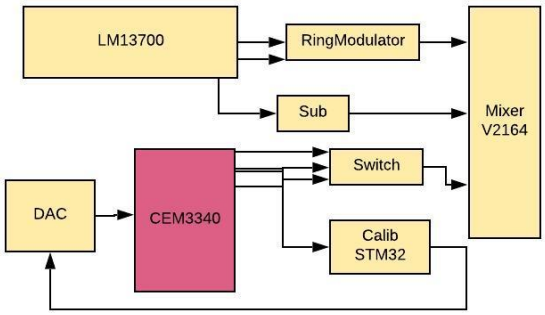


Figure 15.7: Ring modulator in the Themis's architecture

15.4 PIN's details

In this last section we will describe the PINs of the Ring modulator :
Pin 1 :
OUTPUT : on this pin we can get the output signal of the Ring modulator, this is this output we have to connect with one of the mixer input.
Pin 2 Pin 3 :
INPUTs : these two pins have to be connected with the LM13700 VCO's, respectively square and triangle signal. The Ring modulator has been designed to suit to this VCO (voltage and current level). The frequency range cover all the audible frequencies.
Pin 4 Pin 5 Pin 6 Pin 7 :
Power supply's Pins: These PINs, respectively +15V -15V 5V and GND (we

have chosen this standard for all analogical cards) Here we don't use the 5V Pin, thereby don't need to connect it to anything. The others 4 Pins below are connected to the PINs 4 5 6 7. If the Ring modulator is used apart of the Themis, don't need to use these 4 others PINs.

other Pins :

The other Pins are connected to the ground (3 others pins below I/O connectors) These don't have any utility.

Here is the PCB's PINdetails to sum up :

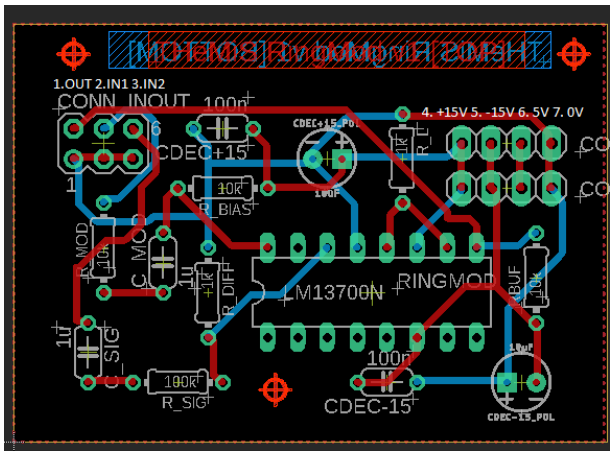


Figure 15.8: PCB TOP PIN's details

Here were all the needed informations to use the Ring modulator inside and outside the Themis.:

Part II

Etudiants 3SyM

Chapter 16

Karplus-Strong synthesis: Matlab prototyping and STM32H7 implementation (by Ahmed W. Moutawakil)

Physical modelling synthesis refers to sound synthesis methods in which the waveform of the sound to be generated is computed using a mathematical model. In the 90s physical modeling seemed like the next big thing, but only a few synthesizers ended up ever using it. The Yamaha VL and the Korg Z1 MOSS are two of the most quoted examples.
Let's bring back physical modelling synthesis with something better than a lousy VST, shall we ?

16.1 Goal

This project's goal is to synthesize an acoustic guitar using the Karplus-Strong algorithm on an STM32H7 development board.
The quality of the synthesis is first going to be tested on Matlab. The written code is then going to be translated to C++ and implemented on the STM32 board.

16.2 Karplus-Strong Synthesis : Overview

Karplus–Strong string synthesis is a method of physical modelling synthesis that loops a short waveform through a filtered delay line to simulate the sound of a

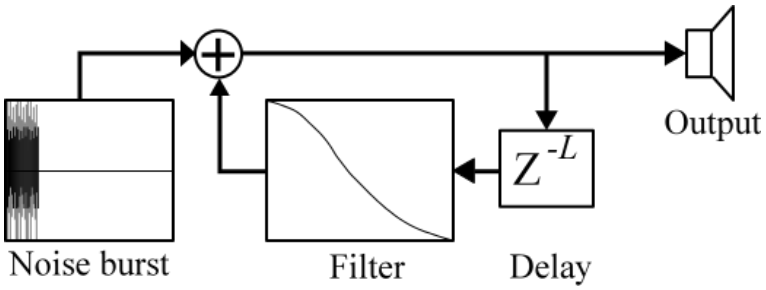


Figure 16.1: Block diagram of a Karplus Strong synthesis

hammered or plucked string or some types of percussion.
At first glance, this technique can be viewed as subtractive synthesis based on a feedback loop similar to that of a comb filter for z-transform analysis. However, it can also be viewed as the simplest class of wavetable-modification algorithms now known as digital waveguide synthesis, because the delay line acts to store one period of the signal.
Alexander Strong invented the algorithm, and Kevin Karplus did the first analysis of how it worked. Together they developed software and hardware implementations of the algorithm, including a custom VLSI chip. They named the algorithm ”Digitar” synthesis, as a portmanteau for ”digital guitar”

16.2.1 How it works

1. A short excitation waveform (of length L samples) is generated. We are going to experiment using a burst of white noise.

2. This excitation is output and simultaneously fed back into a delay line L samples long. The delay line is here to simulate the length of the string.

3. The output of the delay line is fed through a filter. The gain of the filter must be less than 1 at all frequencies, to maintain a stable positive feedback loop.

4. The filtered output is simultaneously mixed back into the output and fed back into the delay line.

16.3 Matlab prototyping

16.3.1 Setting everything up

Two parameters are going to be crucial to our implementation.

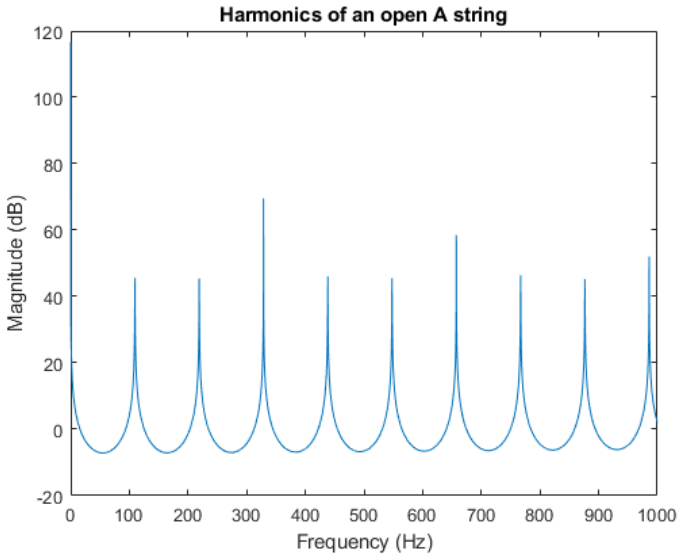


Figure 16.2: Harmonics of an open A string

First, the sample rate is set at 44100Hz. Second is the A note frequency, 110Hz for an A1, we also define a frequency offset for all the other notes.

```
Fe          = 44100;
A           = 110;
```

We then create a frequency vector that is going to be used for our analysis and 4 seconds of zeros to generate the guitar notes.

```
F = linspace(1/Fs, 1000, 2^12);
x = zeros(Fs*4, 1);
```

16.3.2 Synthesizing a note

We determine the feedback delay based on the first harmonic frequency. Then we generate an IIR filter whose poles approximate the harmonics of the A string. The zeros are added for subtle frequency domain shaping.

```
delay = round(Fs/A);
b   = firls(42, [0 1/delay 2/delay 1], [0 0 1 1]);
a   = [1 zeros(1, delay) -0.5 -0.5];
[H,W] = freqz(b, a, F, Fs);
plot(W, 20*log10(abs(H)));
title('Harmonics of an open A string');
xlabel('Frequency (Hz)');
ylabel('Magnitude (dB)');
```

To generate a 4 second synthetic note first we create a vector of states with random numbers. Then we filter zeros using these initial states. This forces the

random states to exit the filter shaped into the harmonics.

```
zi = rand(max(length(b),length(a))-1,1);
note = filter(b, a, x, zi);

note = note-mean(note);
note = note/max(abs(note));
hplayer = audioplayer(note, Fs); play(hplayer)
```

16.4 Hardware implementation

The hardware implementation is straight to the point, the previously coded algorithm is translated to C language. Here is the code :

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <arm_math.h>
#include "dsp.h"
#include "mbed.h"

#define pi 3.14159

#define NB_SAMPLES 1024
#define Fs 48000

#define deuxpinu 2.0 * pi * 0.01

#define A 110 // frequence du A

// filtre RII :
#define TAILLE_NUMER 439
#define TAILLE_DENOM 43

AnalogOut DA(PA_4);
Serial pc(USBTX, USBRX);

float da;
float Amp = 1.0;
float w = 10.0;
float deltaT = 0.01;


/* ------------------------------

Noise -> + -> Retard de L -> Z(n)
          |                   |
          \--- W -- filter ----/

------------------------------ */

//float X[NB_SAMPLES];
float Y[NB_SAMPLES];
float Z[NB_SAMPLES];
float W[NB_SAMPLES];

int retard=5; // ligne a retard
int i,j,n;
```

```
float coeff_a[TAILLE_NUMER]; // numerateur
float coeff_b[TAILLE_DENOM]; // denominateur

//////////////////////////////////////
int main(void) {

    // init des coeffs du filtre
    coeff_a[0]=1;
    for (j=1; j<TAILLE_NUMER-2; j++){
        coeff_a[j]=0;
    }
    coeff_a[TAILLE_NUMER-2]=-0.5;
    coeff_a[TAILLE_NUMER-1]=-0.5;

    for (j=0; j<TAILLE_DENOM; j++){
        coeff_b[j]=-0.0025;
    }
    coeff_b[(TAILLE_DENOM-1)/2]=0.9975;

    srand(12345);

    // boucle principale :
    n=0;
    j=0;


    W[0]=1;
    while((j++)<20000){
    //while(1){

            // 1) Y[n] = Random + W[n]
            //Y[n]=((float)rand()/RAND_MAX)-0.5 + W[n];
            Y[n] = W[n];


            // 2) Z[n] = Y[n-L]
            if (n < retard){
                Z[n]=Y[n-retard+NB_SAMPLES];
            }
            else
                Z[n]=Y[n-retard];

            // 3) W[n] = filtre(Z[n])
            // a) num rateur :
            for (i=0; i<TAILLE_NUMER ; i++){
                if (n < i)
                    W[n] +=  coeff_a[i] * Z[n-i+NB_SAMPLES];
                else
                    W[n] +=  coeff_a[i] * Z[n-i];
            }
            // b) d nominateur :
            for (i=0; i<TAILLE_DENOM ; i++){
                if (n<i)
                    W[n] -=  coeff_b[i] * W[n-i+NB_SAMPLES];
                else
                    W[n] -=  coeff_b[i] * W[n-i];
            }
            pc.printf("%f ",Y[n]);


            DA.write(0.01 * Y[n]+0.5);
```

```
                pc.printf("Y[%d]=%f ; ",n,Y[n]);

                n++;
                if (n==NB_SAMPLES) n=0;

    }


}
```

Chapter 17

An analog-like sequencer for the Themis synthetizer (by Alex Niger)

Because of its various features, Themis deserves a proper sequencer. This one will give the user a chance to program sound changes and parameters over time.

17.1 Generalities and objectives

17.1.1 Sequencers and MIDI

Sequencer

A music sequencer is a device or application software that can record, edit, or play back music, by handling note and performance information in several forms (analog, digital), and possibly audio and automation data. With Themis, a sequencer can change some parameters or play polyphonic material over our different oscillators. It can also show sequences on the screen and the user can make and modify his own sequences. The easiest way to make it is with code. And as a non-audio device, it seemed better to make it part of our raspberry environment. Programmed in JAVA, this sequencer treats with MIDI messages.

MIDI

Short for Musical Instrument Digital Interface, MIDI is a technical standard that describes a communications protocol, digital interface, and electrical connectors that connect a wide variety of electronic musical instruments, computers, and related audio devices. A single MIDI link can carry up to sixteen channels of

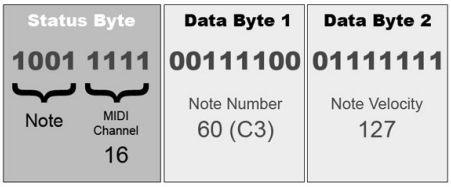


Figure 17.1: MIDI message format

information, each of which can be routed to a separate device. MIDI carries event messages that specify notation, pitch, velocity, vibrato, panning, and clock signals (which set tempo). A message is made of three bytes. To play a note, you need a MIDI ON message followed by a NOTE OFF.

17.1.2 Objectives

Project specifications

- MIDI oriented
- Programmed in JAVA for RaspBerry
- Built for Themis
- Selection/edition fonctionnalities
- Use of java.midi and java.sound libraries
- Test on a virtual synthesizer

17.2 Java libraries takeover

17.2.1 Interfaces

Sequencer and Synthesizer are subinterfaces of MidiDevice interface. They connect each other or they connect to external devices with receivers and transmitters to root MIDI events.

Sequencer

Sequencer interface contains all the top level methods the user needs to play and edit one sequence. The sequencer can be used as a slave or master tempo device. I will only use it as master and leave the default tempo (120 BPM).

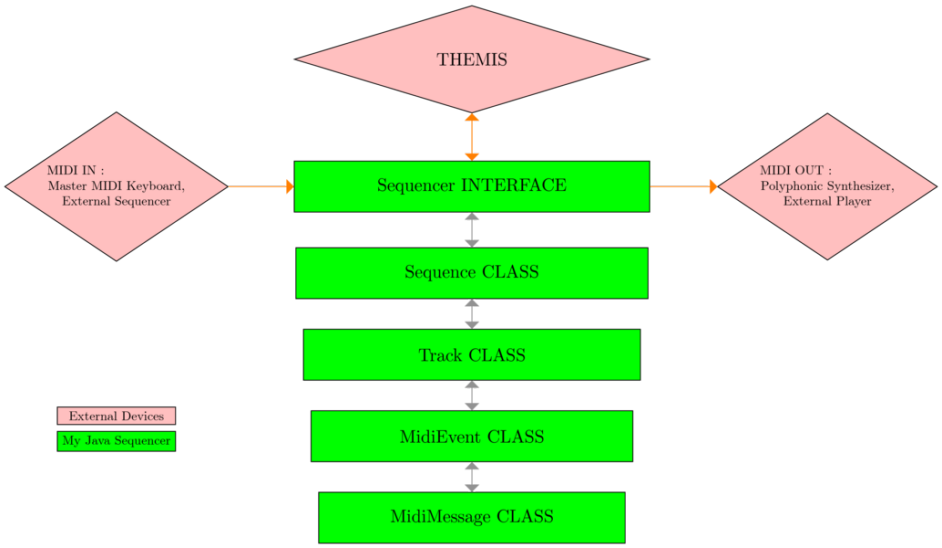


Figure 17.2: Java Sequencer Synoptic

Synthesizer

It is essential to test the sequences with sound. Themis being in process, I had to test it on another synthesizer. Fortunately, the java.sound library contain a midi synthesizer. I will not set it up much and keep the default piano sound.

17.2.2 Classes

Classes design our sequences from MIDI messages.

MidiMessage

This class builds the three bytes messages. It is not mandatory to know how to code them because this class directly takes arguments type, data1 and data2 in.

MidiEvent

A MidiEvent object contains a MidiMessage associated with a time stamp. This time stamp is expressed in ticks.

Track

Track objects add or delete MidiEvent objects at their specific time stamp. It is important to understand there are different ways to build tracks depending on different MIDI messages. Firstly, A NOTE ON message needs a NOTE OFF message to end the note. This means issues can apply if one track is used for chords, especially if the exact same note has to be played twice at the same time. If the same NOTE ON message has been sent x time in the past, this note has to be shut after x NOTE OFF message. So it become essential to count them. Secondly, A CC message sends a parameter value until another one is sent. They can be sent on the same tracks as notes but it might be more difficult to root the signal correctly afterwards.

Sequence

A Sequence object is made of Track objects stored in a Track vector. Every track of the sequence is then identified by its vector index. It could be good to rank them in a logical order. A simple solution remains to initialize a track per note A Sequence constructor takes two parameters to set the resolution: the tempo-based timing type and its proper resolution.

17.3 New Features

Some new functionalities can give this basic sequencer a little more soul.

17.3.1 Time-Shifting Effects

The EffectFactory class contain new time-shifting effects. There are two very useful effects: Quantizer and Randomizer.

Quantizer/Randomizer

Beginners often have difficulties to play precisely tempo. A quantizer will correct it after recording. It just need to put the wanted messages on some specific ticks according to sequence's resolution and sequencer's tempo. On the opposite, a randomizer will be used to get a little distance from the exact tempo. It is a way to sound less digital.

New Class

These new functions need to apply on parts of tracks and on every wanted tracks. I can either add new methods in the sequence class either create a new class for effects. One problem is that Track objects are protected so I need to create a new Track class. But I used an adapter design pattern to get all previous methods from Track.

17.3.2 Real-Time

Copying a sequence to change it before loading it is not very hard but changing in real-time is way more difficult. I found a way to change a sequence and reload it at a precise tick but it is not accurate because, depending how complex the changes are, it takes an unknown and variable process time between the two sequences.

17.4 Biliography

1. https://docs.oracle.com/javase/7/docs/api/javax/sound/midi/Sequencer.html
2. https://en.wikipedia.org/wiki/Sequencer