

Final Degree Project

**Bachelor's degree in Industrial Technology
Engineering**

**Music quartet based on a 32-bit multicore
microcontroller**

REPORT

Author:
Director:
Submission:

Miquel López Muñoz
Manuel Moreno Eguilaz
June 2018



**Escola Tècnica Superior
d'Enginyeria Industrial de Barcelona**



Abstract

This report describes the migration process of the Quartet C language code from a Microchip PIC24 microcontroller to an Infineon TC275 microcontroller. It is also detailed how to use the multicore development board ShieldBuddy TC275 to divide Quartet code into the three microcontroller cores and take advantage of the great capacity to implement several improvements.

The original Quartet code, developed by Victor Timofeev, synthesized 4 different voices (two guitars, a violin and a bass) coordinated by a conductor task, each of which played from its own music sheet, which was stored in program memory (ROM). The sound samples were generated using amplitude modulation: the characteristic waveform of the instrument (also stored in ROM) and its amplitude envelope (generated through software) were multiplied, and later converted to an analog output through an 8-bit PWM running at 78 kHz and an RC low pass filter. The generated sound could later be played on speakers.

This project explains the basic operation of Quartet Project and step by step how to make the software migration process. It also describes the achieve improvements in the quality sound and efficiency by using a 32-bit powerful microcontroller. Then, it details the restructuration of the whole code in order to use the three cores available and how to share memory among them. Finally, it explains how to set up overdrive effect in the bass and the two guitars.

After completing the project, a program runs by three 32-bit cores generating two 9-bit PWM signals running at 80 kHz is obtained. The capacity of the program has been widely increased, the characteristic waveform and amplitude envelope have been more accurate and realistic and the bass and two guitars can play with overdrive effect.

At the end of the project, some possible improvements are suggested for future projects. A drum kit had to be implemented but it has not been possible because of some errors and limitations also explained in the report. However, it has been proposed a way to add and fit in the code a drum kit with the three basic elements; bass drum, snare drum and hit hat.

All the work done in this project remains as a base for the realization of futures final degree projects.

Summary

| | |
|---------------------------------------|----|
| ABSTRACT | 1 |
| SUMMARY | 3 |
| List of Tables..... | 5 |
| List of Figures | 5 |
| 1. GLOSSARY | 9 |
| 2. PREFACE | 11 |
| 2.1. Origin of the project..... | 11 |
| 2.2. Motivation | 12 |
| 2.3. Previous requirements | 12 |
| 3. INTRODUCTION | 13 |
| 3.1. Objectives of the project..... | 13 |
| 3.2. Scope of the project | 13 |
| 4. QUARTET OPERATION | 15 |
| 4.1. Quartet file structure..... | 16 |
| 4.2. <i>Tempo Part</i> | 17 |
| 4.2.1. Functions | 17 |
| 4.2.2. Tasks..... | 17 |
| 4.3. <i>Sound Part</i> | 18 |
| 4.3.1. Synthesizer | 18 |
| 4.3.2. PWM | 20 |
| 5. SOFTWARE MIGRATION | 21 |
| 5.1. Generic Timer Module (GTM) | 21 |
| 5.1.1. Introducing the GTM_TOM | 21 |
| 5.1.2. GTM_TOM Configuration..... | 23 |
| 5.2. <i>MUL()</i> migration | 26 |
| 5.3. Code migration..... | 27 |
| 5.4. Improvements | 28 |
| 5.4.1. CMD_REPEAT | 28 |
| 5.4.2. Envelope tables | 28 |
| 6. SOUND QUALITY IMPROVEMENT | 29 |

| | |
|---|-----------|
| 6.1. Current project analysis | 29 |
| 6.2. Changing <i>cmd</i> into <i>integer type</i> | 31 |
| 6.3. Amplitude envelope tables..... | 33 |
| 6.4. Characteristic waveform tables..... | 35 |
| 6.5. Adding an extra octave | 39 |
| 6.6. Conclusions about sound quality improvement..... | 39 |
| 7. TRI-CORE MANAGEMENT | 40 |
| 7.1. Project restructuring | 40 |
| 7.2. Sharing Memory..... | 44 |
| 7.2.1. <i>SharedMem</i> tool | 44 |
| 7.2.2. LMU SRAM..... | 46 |
| 7.2.3. Final Solution..... | 47 |
| 7.3. Timers configuration | 52 |
| 8. OVERDRIVE EFFECT | 53 |
| 8.1. Effect description..... | 53 |
| 8.2. Overdrive implementation..... | 54 |
| 9. LIMITATIONS AND PROBLEMS | 55 |
| 10. FUTURE IMPROVEMENTS | 56 |
| 10.1. Inter-core communication | 56 |
| 10.2. Adding new instruments or effects..... | 56 |
| 10.3. Drum kit..... | 57 |
| 10.4. Other ideas..... | 58 |
| 11. BUDGET | 59 |
| 12. ENVIRONMENTAL IMPACT | 60 |
| CONCLUSIONS | 61 |
| THANKS | 63 |
| BIBLIOGRAPHY | 65 |
| Bibliographic references | 65 |
| Complementary bibliography..... | 67 |



List of Tables

| | |
|--|----|
| Table 4.1: Name of the TSound variable of each channel | 15 |
| Table 5.1: Summary of the syntax changes in migration..... | 27 |
| Table 7.1: TSound variables in each main files | 42 |
| Table 7.2: Content of each file in the project..... | 43 |
| Table 11.1: Project Budget..... | 59 |

List of Figures

| | |
|---|----|
| Fig. 4.1: File structure in Pau's Quartet code | 16 |
| Fig. 4.2: Example of characteristic waveform | 18 |
| Fig. 4.3: Example of amplitude envelope | 18 |
| Fig. 4.4: Amplitude modulation synthesis conceptual diagram | 19 |
| Fig. 4.5: Example of information being stored and sent using PWM | 20 |
| Fig. 5.1: TOM block diagram | 22 |
| Fig. 5.2: Connector Pin Allocation | 25 |
| Fig. 5.3: Comparison of CMD_REPEAT before and after | 28 |
| Fig. 5.4: Comparison of forming amplitude envelope before and after | 28 |
| Fig. 6.1: Variable ENUM_NOTES | 29 |
| Fig. 6.2: Data distribution in cmd when a note is assigned | 30 |
| Fig. 6.3: Data distribution in cmd when setbase command is assigned | 30 |
| Fig. 6.4: Command definition..... | 31 |
| Fig. 6.5: New data distribution of cmd as note..... | 32 |

| | |
|---|----|
| Fig. 6.6: New data distribution of cmd as setbase..... | 32 |
| Fig. 6.7: Original BASS amplitude envelope | 33 |
| Fig. 6.8: New BASS amplitude envelope..... | 33 |
| Fig. 6.9: Original VIOLIN amplitude envelope | 34 |
| Fig. 6.10: New VIOLIN amplitude envelope | 34 |
| Fig. 6.11: Original GUITAR amplitude envelope | 34 |
| Fig. 6.12: New GUITAR amplitude envelope..... | 35 |
| Fig. 6.13: Original BASS characteristic waveform..... | 35 |
| Fig. 6.14: New BASS characteristic waveform..... | 36 |
| Fig. 6.15: Original VIOLIN characteristic waveform | 36 |
| Fig. 6.16: New VIOLIN characteristic waveform..... | 37 |
| Fig. 6.17: Original GUITAR characteristic waveform..... | 37 |
| Fig. 6.18: New GUITAR characteristic waveform..... | 38 |
| Fig. 7.1: New file structure in Quartet project | 43 |
| Fig. 7.2: schematic of tri-core behaviour..... | 44 |
| Fig. 7.3: Description of SharedMemSend() function | 45 |
| Fig. 7.4: Description of SharedMemReceive() function..... | 45 |
| Fig. 7.5: Multicore Memory Map | 46 |
| Fig. 7.6: Schematic of how to use LMU SRAM | 48 |
| Fig. 7.7: Address location definitions | 48 |
| Fig. 7.8: Schematic of sending data byte by byte..... | 50 |
| Fig. 7.9: Example of how to store data in LMU SRAM using pointers..... | 50 |
| Fig. 7.10: Example of how to read data from LMU SRAM using pointers | 51 |



Fig. 8.1: Physical modulation of sound wave to get overdrive effect 53

Fig. 10.1: Notework() extension to use drum kit command 57

1. Glossary

CMD: Command

CMU: Clock Management Unit

CPU: Central Processing Unit

GTM: Generic Timer Module

IDE: Integrated Developing Environment

ISR: Interrupt Service Routine

LED: Light-Emitting Diode

LMU SRAM: Local Memory Unit - Static Random Access Memory

OS: Operating System

PFC: Projecte Final de Carrera, translated into English as Final Career Project

PIC: Peripheral Interface Controller

PSoC: Programmable System on a Chip

PWM: Pulse Width Modulation

RC: Resistor-Capacity

RF: Radio Frequency

ROM: Read-Only Memory

RTOS: Real Time Operating System

TFG: Treball Final de Grau, translated into English as Final Degree Project

TGC: Tom Global Channel

TOM: Timer Output Module

2. Preface

2.1. Origin of the project

The origin of this project goes back to Victor Timofeev's usage example of the OSA real time operating system (RTOS) [1]. Timofeev designed a program, known as "Quartet" and written in C language, that was able to synthesize a 4-channel melody using an 8-bit PWM running at 78 kHz so that, with the help of an RC filter, it could be played on speakers or headphones. The original code was written for a PIC16 microcontroller and used the OSA RTOS as an operating system.

In 2015, Pere Domenech, a student at ETSEIB (Escola Tècnica Superior d'Enginyeria Industrial de Barcelona), implemented a new project (PFC) migrating for the first time the PIC16 original code to a PIC18 microcontroller also using OSA RTOS [2].

In February 2017, Pau Mendieta, a student at the same university, developed a project (TFG) by migrating the PIC18 and OSA RTOS code to a PIC24 microcontroller and FreeRTOS [3], changing from an 8-bit CPU to a 16-bit CPU and using this more complete operating system [4] for the first time. This project was a great advance on the CPU capacity but did not get a significant sound quality improvement because of PIC24 only could provide a 9-bit PWM running at 31.25 kHz.

In January 2017, Antonio Martínez [5], another student at ETSEIB, succeeded in improving the sound quality by migrating Pau Mendieta's code so that it could be executed using a PSoC 4 microcontroller with a 16-bit resolution PWM. Also working with a 32-bit CPU and FreeRTOS like Pau, Antonio got a better improvement of sound quality thanks to PSoC 4 using a 16-bit PWM running at 92 kHz.

Finally, it is important to mention other projects (TFGs) related to the Quartet are: Joan Gallostra's *RF Music Festival: orquestra basada en microcontroladores PIC18 y RF* [6] and Joan Calvet's CAN Music Festival: *Orquestra Basada en Microcontroladors PIC18 i un bus CAN* [7]. The former designed a musical orchestra where each instrument was individually synthesized by four different PIC18 microcontrollers and the conductor sent commands or notes to each instrument through RF. The latter pursued the same objective, but using a CAN bus instead of RF as a means of communication. Moreover, it is important to mention Allan Tengg, the author of the FreeRTOS migration for Aurix TC27X [8].

2.2. Motivation

The reasons that give rise to the development of this project are the following:

- Learn how to program one microcontroller in C language.
- Learn how to use a powerful microcontroller, specifically the multicore 32-bit TC275 from Infineon.
- Understand how a RTOS works, specifically FreeRTOS.
- Learn how to work with 3 different cores intercommunicated in the same project.
- Deepen in application design using microcontrollers.
- Apply the knowledge learned in an engineering degree to a practical case.
- Learn a method for synthesizing music.
- Understand and migrate an existing code in order to add personal improvements.
- Provide a starting base for the realization of future Final Degree Projects and Final Master Projects.

2.3. Previous requirements

This is a project based on electronics and computer science. Therefore, it is mandatory to have a certain degree of knowledge about digital electronics and programming. Specifically, it is essential to know about programming in C language [9] because original code is written in this language. At the same time, this project is focused on music so requires a basic knowledge about music.

Regarding to the microcontroller, it is necessary to understand the structure and behaviour of the chosen multi-core microcontroller, the Infineon TC275, in order to configure its ports and peripheral hardware blocks and develop the code. Specifically, it is mandatory to learn about multi-core communication, PWM and Timer modules.

In addition, it is important to understand how FreeRTOS works and manages the different tasks and semaphores, since an important part of the code is based on this point. Respecting on development environment, it is necessary to get familiar with how development tool, the well-known Eclipse IDE [10], works and the way it uses the three cores debugger and manages breakpoints.



3. Introduction

3.1. Objectives of the project

The main objectives of this project can be summarized as follows:

- Migrating the original Quartet code to work in a ShieldBuddy TC275 based on the Infineon TC275 multicore processor. An important step is how to use Infineon Frameworks tools [11] (PWM, Timers, ...).
- Making use of the huge capacity of the Infineon TC275 microcontroller to improve the sound quality without it working in a hurry.
- Working for the first time in one TFG with a multicore microcontroller and learn how to share memory between the three different cores available in TC275 in order to get a Quartet project using the three at the same time.
- Adding to the project the possibility to use overdrive distortion in the two guitar and the bass.
- Designing a way to add a drum kit and implement the three basic elements: bass drum, snare drum and hit hat.

3.2. Scope of the project

This project is related to electronics and computer science. A significant amount of time has been invested on understanding how the original Quartet code and tasks system work, learning about the C language programing, FreeRTOS and Infineon Frameworks and getting use to the Eclipse IDE debugger.

The whole project has been compiled to run in a ShieldBuddy TC275 board written in C language using FreeRTOS [4] and Eclipse IDE [10] from Hightec [12].

The current project has been divided into three cores to increase capacity. It involves some modifications of the original code in order to improve the sound quality, to make possible to share memory between cores and to add the overdrive effect. In the end, two different PWM signals have been created.

During the project, a great deal of errors and unforeseen have appeared on account of the quality of the tools and the development board. In addition, there is very little online information about the mentioned issues. Because of this, it was impossible to implement altogether the drum kit in the time frame of the project so it will not be included in the final version. However, the work that has been done about this topic will be mentioned in future implementations in order to contribute in further projects a way to avoid it.

This project will conclude when all the three cores work correctly, being able to share memory into them, and the two PWM signal playing sound synchronously. Finally, further improvements will be suggested for future projects



4. Quartet operation

To understand the work done in this project, it is mandatory to invest some time to understand the basic Quartet operation. The project starts from the TFG entitled Pau Mendieta's *Audio Application based on FreeRTOS Operating System [3]*, so the explanation will be based on this code.

It can be considered that the Quartet code is divided in two parts. The first one, which could be called *Tempo Part*, takes care of reading the score and supplying data to the other part, which could be called *Sound Part*. *Sound Part* uses data from *Tempo Part* and generates the corresponding PWM signal. Everything is coordinated thanks to the use of tasks run by the well-known FreeRTOS.

This procedure applies to four channels, which represent the four different available instruments: Bass, Violin, Guitar1 and Guitar2. For each instrument, there is a sound variable declared in the original code as S1, S2, S3, S4, for channel 1, 2, 3 and four, respectively.

| | |
|---------------------------|----|
| Channel 1: Bass | S1 |
| Channel 2: Violin | S2 |
| Channel 3: Guitar1 | S3 |
| Channel 4: Guitar2 | S4 |

Table 4.1: Name of the TSound variable of each channel. Source: own.

These sound variables are a structure called *TSound*, which stores data from the sound channel. For example, parameters such as duration or frequency are stored in this structure or if the sound is enabled or disabled.

4.1. Quartet file structure

Before explaining the operation of the original Quartet code, it could be useful to take a look at the most relevant files and their structure.

- *Main.c*: this is the file where the main code is written. It contains the operative of the two parts of the project and all the configuration and initialization.
- *Sinus.h*: this is a header file that contains tables of the waveforms for each instrument; bass, violin and guitar.
- *Elocha.h*: this is a header file that contains the definitions needed to read the score and set the notes and the corresponding frequency of each note.
- *Bach1067.h*: a header file that contains a list of commands and notes for each instrument. This list is called *notelist*.

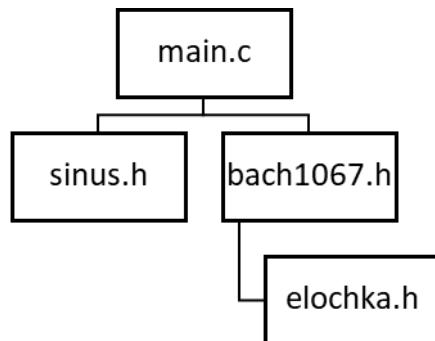


Fig. 4.1: File structure in Pau's Quartet code. Source: own.



4.2. **Tempo Part**

Tempo Part takes care of reading the score and supplies data to *Sound Part*. In this part there are two functions and five tasks that are important to understand.

4.2.1. Functions

- *InitSoundVariable()*: This function set the initial values of *TSound* variables, matches each instrument with its *notelist* and initializes the sound of the corresponding channel.
- *NoteWork()*: This function reads *notelist* and sets the appropriate values of *TSound* variables according to the note that it has to play or the command it has to execute.

4.2.2. Tasks

There are five tasks to know, four of them are similar: one task for each channel. The other is the task called *conductor*.

The *conductor* could be considered the master. By using semaphores, it says which instrument task is going to be run. It sends a semaphore to each instrument for every command or note and then creates a delay that sets the tempo.

The first time after power on reset, each instrument task calls *InitSoundVariable()* and sets the corresponding channel available to play. The following times, it calls *NoteWork()*, reads the score and updates the *TSound* data.

4.3. Sound Part

Sound Part uses data from *Tempo Part* and generates the corresponding PWM signal. In this part, there are two important elements: the synthesizer and the PWM generation.

4.3.1. Synthesizer

To simulate the sound of the different instruments, each of them has a characteristic waveform (stored in *sinus.h*) and a sound envelope. To get the wanted sound, it has to combine both.

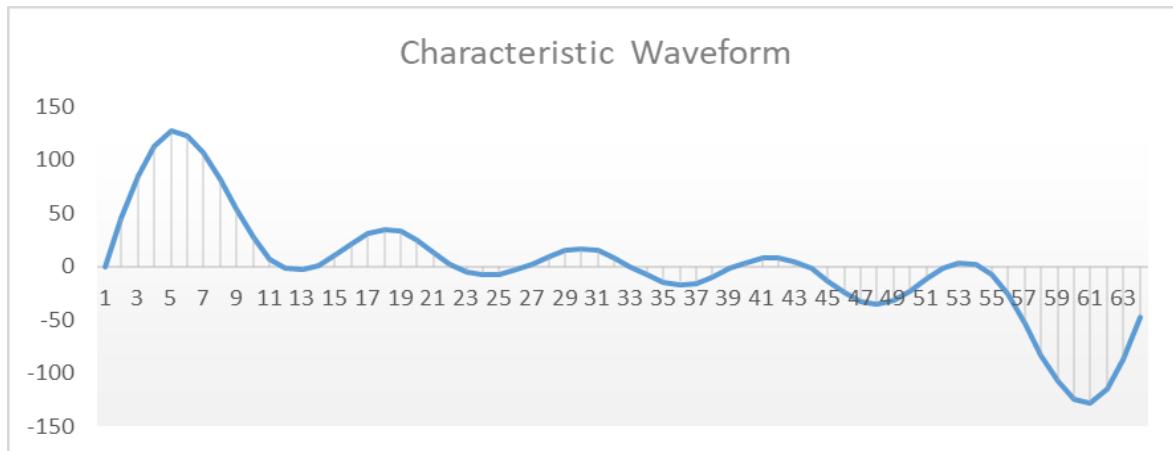


Fig. 4.2: Example of characteristic waveform. Source: Victor Timofeev's original Quartet code.

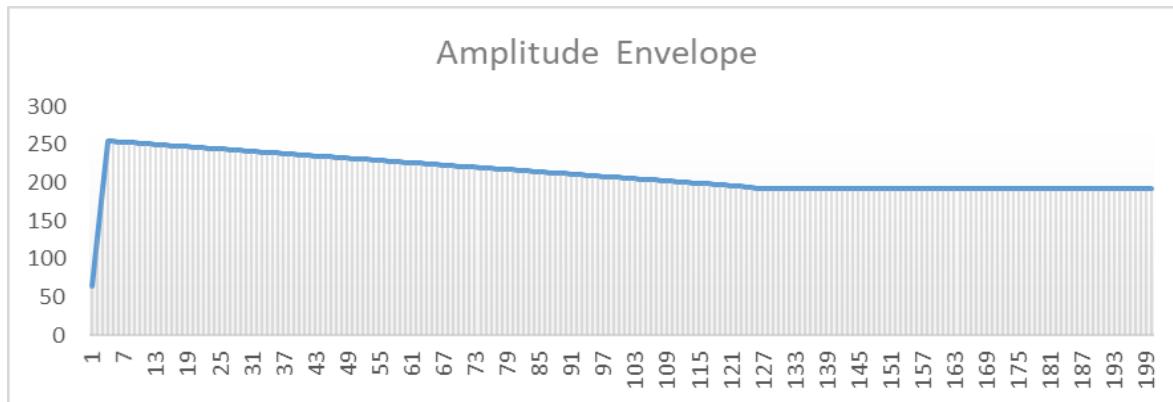


Fig. 4.3: Example of amplitude envelope. Source: Victor Timofeev's original Quartet code.



The sound synthesizing is done during processor interrupts using a technique known as amplitude modulation synthesis [2]. It consists in multiplying two signals: the modulator signal and the carrier signal. In this case, the carrier signal is a periodical repetition of the instrument characteristic waveform (Fig. 4.2) and the modulator signal is the instrument envelope (Fig. 4.3). This is done to simulate the natural evolution of sound.

In the original code, this multiplication was done through a macro called *MUL()*, which was written in assembly code in order to speed up calculations.

Once every sample for each voice is generated, they are added and the resulting value is divided by 4.

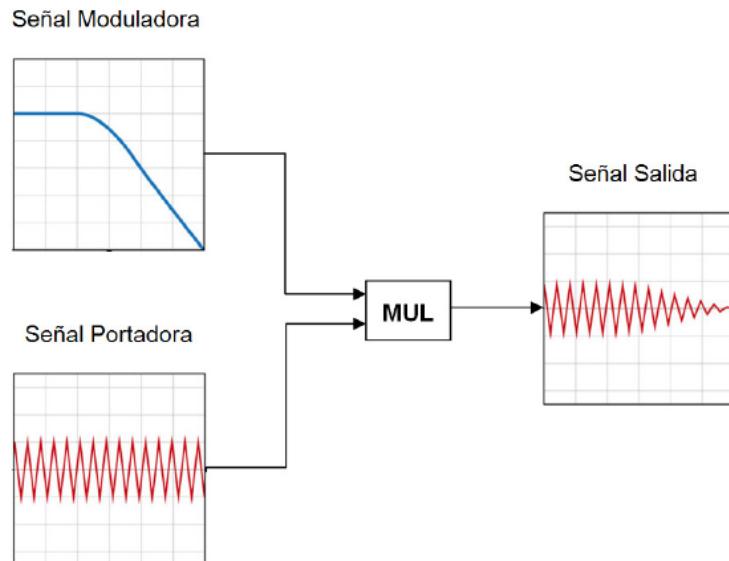


Fig. 4.4: Amplitude modulation synthesis conceptual diagram. Source: [2].

4.3.2. PWM

Once the synthesizer has generated the output signal, this information is stored and sent within the width of a pulse. This is known as pulse width modulation. Finally, the resulting PWM signal passes through a low-pass filter, which eliminates the low voltage fraction of the signal, and thus the waveform is recovered and reproduced through speakers.

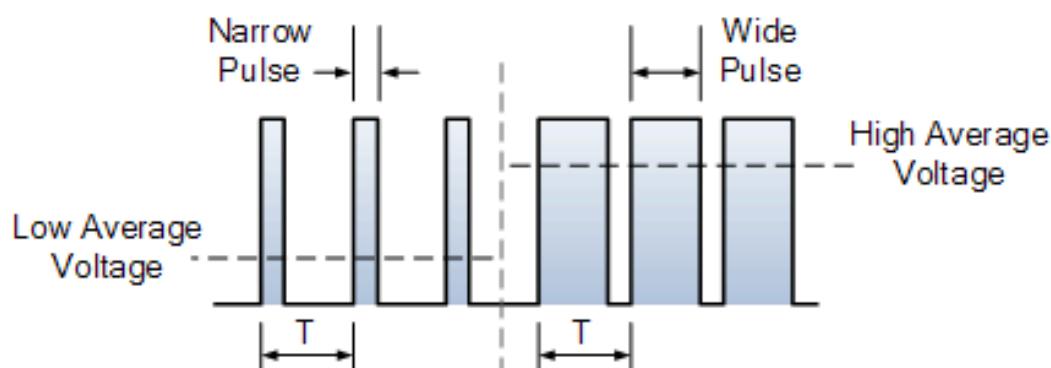


Fig. 4.5: Example of information being stored and sent using PWM. Source: [13].



5. Software migration

5.1. Generic Timer Module (GTM)

5.1.1. Introducing the GTM_TOM

Before starting the migration, it is important to know about Infineon TC275 multicore microcontroller behaviour and structure. It is a three core microcontroller; core 0, 1 and 2. To do the first migration and simplify the test only the core 0 will be used to store the whole code.

To manage timers and outputs, it is mandatory to learn about Timer Output Module (TOM). It offers 16 independent channels to generate simple PWM signals at each output pin. There exist two global channel control units (TGC0 and TGC1) to drive a number of individual TOM channels synchronously by external or internal events. Each TGC[y] can drive up to eight TOM channels where TGC0 controls TOM channels 0 to 7 and TGC1 controls TOM channels 8 to 15.

The architecture of each TOM submodule is depicted in Fig. 5.1.

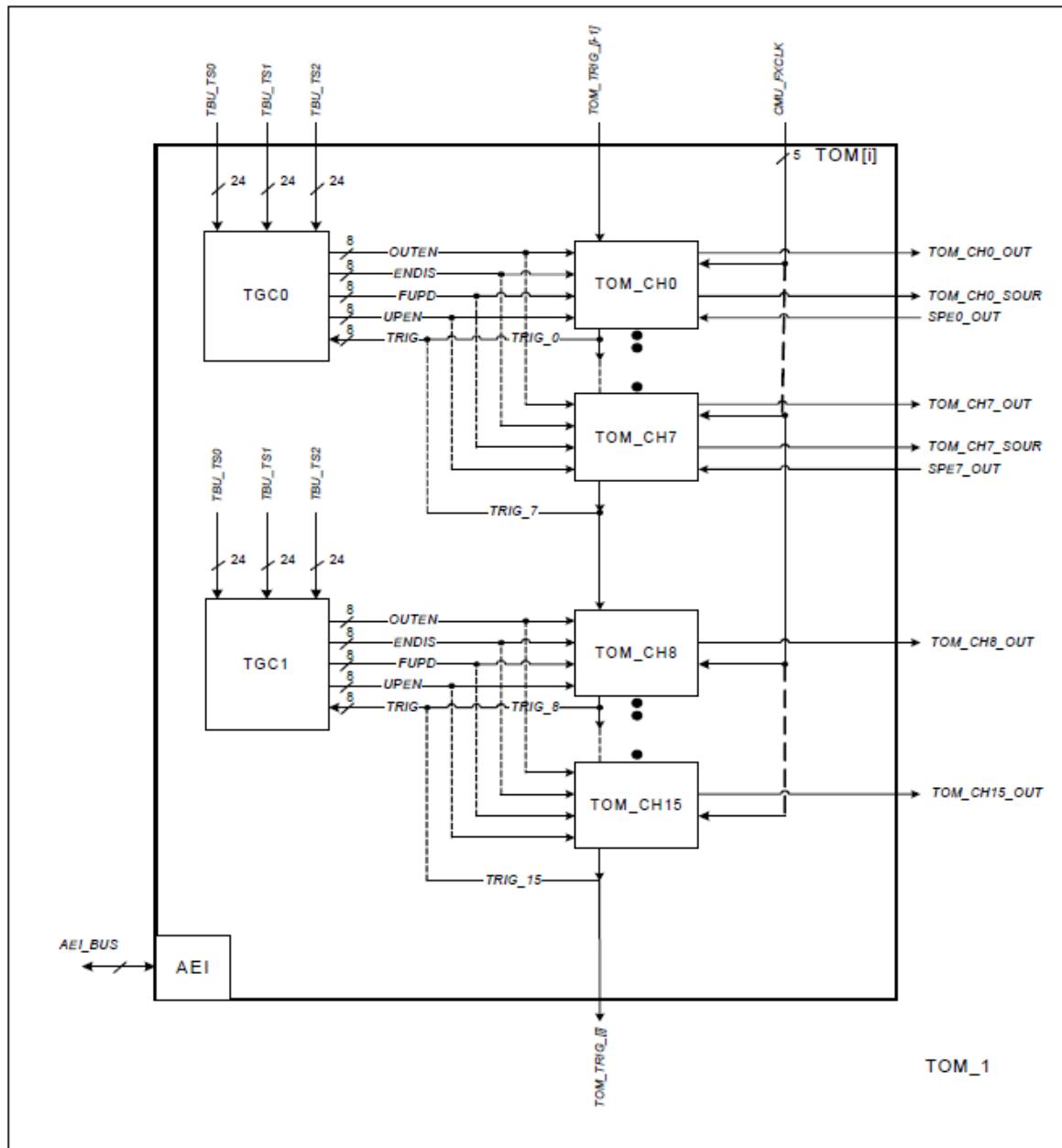


Fig. 5.1: TOM block diagram. Source: [14].



5.1.2. GTM_TOM Configuration

To get a PWM signal, it has been necessary to analyse an example project called *BaseFramework_TC27xC_iLLD_GtmTomPwmHIDemo* [15] and the microcontroller document called *AURIX unleashed. Getting started with AURIX* [16] and it has been concluded that the best thing is to generate a Simple Centre-Aligned PWM and an interrupt to modify the value. So, the following basic steps to get a generic PWM have been considered:

- Connect the system clock to the GTM to enable it.

```
Ifx_GTM *gtm = &MODULE_GTM;
IfxGtm_enable(gtm);
```

- Get the current GTM clock frequency (System Peripheral Bus).

```
g_GtmTomTimer.info.gtmFreq = IfxGtm_Cmu_getModuleFrequency(gtm);
```

- Set up the GTM's Clock Management Unit (CMU).

```
IfxGtm_Cmu_setGclkFrequency(gtm, g_GtmTomTimer.info.gtmFreq);
g_GtmTomTimer.info.gtmGclkFreq = IfxGtm_Cmu_getGclkFrequency(gtm);
```

- Set the required frequency of the interrupt to be generated.

```
timerConfig.base.frequency = 10000;
```

- Set the priority of the interrupt to be generated.

```
timerConfig.base_isrPriority = ISR_PRIORITY(INTERRUPT_TIMER_1MS);
```

- Set the source of the interrupt

```
timerConfig.base_isrProvider = ISR_PROVIDER(INTERRUPT_TIMER_1MS);
```

- Make a check to see whether the requested interrupt frequency is feasible.

```
timerConfig.base_minResolution = (1.0/timerConfig.base.frequency)/1000;
```

- Disable the TOM output trigger as there are no other TOMs sending triggers to this TOM.

```
timerConfig.base_trigger.enabled = FALSE;
```

- Set which TOM to use (0, 1 and 2) and which channel. Here it is TOM1, channel 1.

```
timerConfig.tom = IfxGtm_Tom_1;
timerConfig.timerChannel = IfxGtm_Tom_Ch_0;
```

- Set which clock source the TOM should use.

```
timerConfig.clock = IfxGtm_Tom_Ch_ClkSrc_cmuFxclk1;
```

- Write the configuration into the GTM:

```
IfxGtm_Tom_Timer_init(  
&g_GtmTomTimer.drivers.timerOneMs, &timerConfig);
```

- Set up Two Complementary PWM Outputs. Nominate the two port pins to be used for the PWM outputs.

```
IfxGtm_Tom_ToutMapP ccx[1] = {&IfxGtm_TOM1_1_TOUT10_P00_1_OUT};  
IfxGtm_Tom_ToutMapP coutx[1] = {&IfxGtm_TOM1_2_TOUT12_P00_3_OUT};  
.....  
pwmH1Config.ccx = ccx;  
pwmH1Config.coutx = coutx;
```

- Indicate the base timer to be used. Here it is the TOM1 channel 0 from the previous steps above.

```
pwmH1Config.timer = &g_GtmTomPwmH1.drivers.timer;  
pwmH1Config.tom = timerConfig.tom;
```

- Set the number of channels of PWM to generate, plus the deadtime offset between them and the minimum permitted pulselength. The units for these are expressed in seconds.

```
pwmH1Config.base.deadtime = 2e-6;  
pwmH1Config.base.minPulse = 1e-6;  
pwmH1Config.base.channelCount = 1;
```

- Set up the output driver to be used by the PWM pins.

```
pwmH1Config.base.outputMode = IfxPort_OutputMode_pushPull;  
pwmH1Config.base.outputDriver = IfxPort_PadDriver_cmosAutomotiveSpeed1;
```

- Set the active states of the two pins.

```
pwmH1Config.base.ccxActiveState = Ifx_ActiveState_high;  
pwmH1Config.base.coutxActiveState = Ifx_ActiveState_high;
```

- Write the configuration into the GTM.

```
IfxGtm_Tom_PwmH1_init(&g_GtmTomPwmH1.drivers.pwm, &pwmH1Config);
```

- Start the TOM1_0.

```
IfxGtm_Tom_Timer_run(&g_GtmTomPwmH1.drivers.timer);
```



All the mentioned steps are needed to get one PWM signal, but it is necessary to get PWM a signal running at 78 kHz and an interrupt running at 10 kHz. Hence, it is needed two timers, one for the PWM signal and the other for the interrupt service.

Although only TOM0 and TOM1 are capable to generate PWM signals, there exists also another TOM, called TOM2, which can manage a timer for the interrupt service. So, it will be TOM2 in charge of managing the interrupt service routine at 10 kHz and TOM1 used to generate PWM signal.

To configure the output pins, it has been used the pin map of the Fig. 5.2 and a header file called *IfxGtm_PinMap.h*, that matches the output pins with the TOM channels.

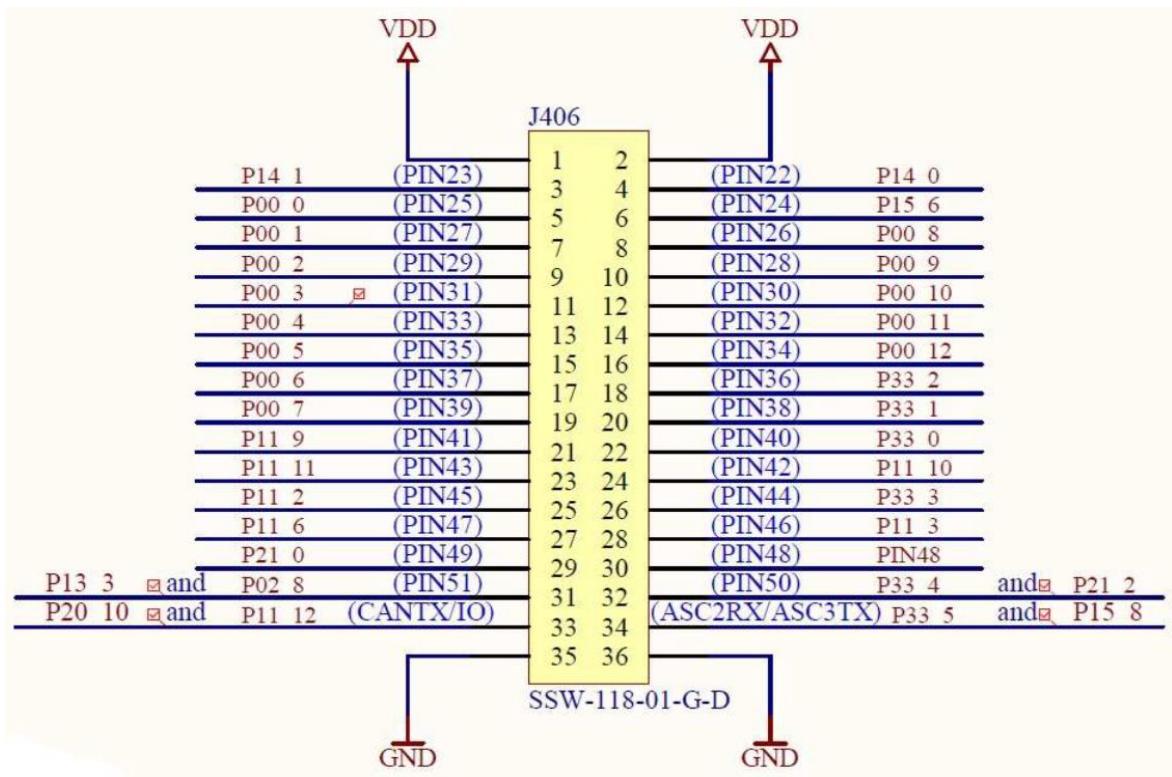


Fig. 5.2: Connector Pin Allocation. Source: [20].

Finally, two functions (*GtmTomPwmHl_initTimer()* and *GtmTomPwmHlDemo_init()*) have been implemented to initialize and configure the timers and interrupt service called *ISR_Timer()*, in addition to the necessary variables to store data extracted from the example project.

5.2. *MUL()* migration

In Pau's project [3] the sound synthesizing is done using the amplitude modulation synthesis. This operation done through a macro called *MUL()* is written in assembler in order to speed up calculations. Pau had a 16-bit CPU and he needed to beware of the resource usage but TC275 is a 32-bit CPU and it is powerful enough to make very fast arithmetic operations such as an arithmetic product between two variables in C language.

To do that, three variables: *temp1*, *temp2* and *temp_dac* have been used:

- *temp1*: characteristic waveform value.
- *temp2*: amplitude envelope value.
- $\text{temp_dac} = \text{temp_dac} + \text{temp1} * \text{temp2}$

temp_dac will be accumulating the values of all instruments and finally it needs to be adjusted.



5.3. Code migration

With the detailed changes in the previous chapters, it is time to make the code migration. Now, only it is necessary to make some syntax changes:

| Microchip PIC 24 | Infineon TC275 |
|---|---|
| <pre>EventGroupHandle_t flag_Playing #define FLAG_BASS_PLAYING 0x01</pre> | <pre>Struct { unsigned BASS_PLAYING:1 } Flags</pre> |
| <code>flag_Playing = xEventGroupCreate()</code> | <code>Flags.BASS_PLAYING = 0</code> |
| <code>xEventGroupClearBits(flag_Playing, FLAG_BASS_PLAYING)</code> | <code>Flags.BASS_PLAYING = 0</code> |
| <code>xEventGroupSetBits(flag_Playing, FLAG_BASS_PLAYING)</code> | <code>Flags.BASS_PLAYING = 1</code> |
| <code>static TaskHandle_t xTaskBass = NULL</code> | <code>static xTaskHandle xTaskBass = NULL</code> |
| <code>SemaphoreHandle_t BS_START_MUSIC</code> | <code>xSemaphoreHandle BS_START_MUSIC</code> |
| <code>BS_START_MUSIC = xSemaphoreCreateBinary()</code> | <code>vSemaphoreCreateBinary(BS_START_MUSIC)</code> |
| - | <code>xSemaphoreHandle BS_BASS</code> |
| - | <code>vSemaphoreCreateBinary(BS_BASS)</code> |
| <code>ulTaskNotifyTake(pdTRUE, portMAX_DELAY)</code> | <code>xSemaphoreTake(BS_BASS, portMAX_DELAY)</code> |
| <code>xTaskNotifyGive(xTaskBass)</code> | <code>xSemaphoreGive(BS_BASS)</code> |

Table 5.1: Summary of the syntax changes in migration. Source: own.

To success in the code migration, the following documents have been consulted: *FreeRTOS Reference Manual [17]*, *Mastering the FreeRTOS Real Time Kernel - A Hands-On Tutorial Guide [18]* and *Using the FreeRTOS Real Time Kernel - A Practical Guide [19]*.

5.4. Improvements

5.4.1. CMD_REPEAT

In *NoteWork()*, the function developed to read the score and update *TSound* variables data, the repeat command is only able to repeat a fragment once. In case that was necessary to repeat two or more times, it will repeat one time and reset data related to the repetition. Hence, a little adjust has to be done to fix this.

```
case CMD_REPEAT: // Repeat fragment
    if (S->bRepeat || !(cmd & 0x1F)) {
        S->bRepeat = 0;
        S->cRepeatPosition = S->cCurNote;
    } else {
        S->bRepeat = 1;
        S->cCurNote = S->cRepeatPosition;
    }
    break;
```

```
case CMD_REPEAT: // Repeat fragment
    if (S->bRepeat == 1 || !(cmd & 0x1F)) {
        S->bRepeat = 0;
        S->cRepeatPosition = S->cCurNote;
    }
    else if (S->bRepeat > 1) {
        S->bRepeat--;
        S->cCurNote = S->cRepeatPosition;
    }
    else { // 
        S->bRepeat = (cmd & 0x1F);
        S->cCurNote = S->cRepeatPosition;
    }
    break;
```

Fig. 5.3: Comparison of CMD_REPEAT before and after. Source: own.

5.4.2. Envelope tables

In Pau's code [3], the envelope values are calculated by operating in the interrupt service. It is easier to make a table of values like characteristic waveforms, so the tables have been including in *sinus.h* file and the code of the interrupt service have been modified for each channel as shown in the following.

```
// FORMING SIGNAL_GUITAR //Amplitude Envelope
if (S3.t > 64) {
    temp2 = S3.t >> 2;
    temp2 = 64 - temp2;
} else if (S3.t >= 4) {
    temp2 = 128 - S3.t;
} else {
    if (!S3.t) {
        temp2 = prs << 2;
    } else {
        temp2 = 255;
        if (S3.t & 2) temp2 &= ~0x40;
        if (S3.t & 1) temp2 &= ~0x20;
    }
}
```

```
// FORMING SIGNAL_BASS
temp2 = bassEnv[S1.t & 0x40];
```

Fig. 5.4: Comparison of forming amplitude envelope before and after. Source: own.



6. Sound quality improvement

In this chapter the improvements in the sound quality of the Quartet Project will be explained.

6.1. Current project analysis

To know how the project can be improved it is mandatory to make an exhausting analysis of the main code and the header files.

After the first look, it shows that it is nothing to do with the tasks system. It is complex but it is well optimized. The same happens on the GTM_TOM configuration. It was expected but you have to discard.

Checking the *NoteWork()* function and *elochka.h* header file, some interesting conclusions can be made. The variable that is used to know if there is a note or a command, and which type of command, in the score (*cmd*) is an *unsigned char*. And this generates some limitations:

- A. The maximum *note* that score can set is 31.

In this project, notes are represented in an *enum* variable *ENUM_NOTES* ordered in octaves. Each note is represented with a number. There are four octaves and 57 notes in total.

```
enum ENUM_NOTES {
    C0, C0_, D0, D0_, E0, F0, F0_, G0, G0_, A0, A0_, B0,
    C1, C1_, D1, D1_, E1, F1, F1_, G1, G1_, A1, A1_, B1,
    C2, C2_, D2, D2_, E2, F2, F2_, G2, G2_, A2, A2_, B2,
    C3, C3_, D3, D3_, E3, F3, F3_, G3, G3_, A3, A3_, B3,
    C4, C4_, D4, D4_, E4, F4, F4_, G4, G4_
};
```

Fig. 6.1: Variable *ENUM_NOTES*. Source: own.

When there is a note in the score, a command called *play(note, duration)* is used and the definition of this command is $((duration-1) \ll 5) | note$. Since *cmd* is an *unsigned char*, only 8 bits are available. So the distribution of the data in *cmd* variable is the following:

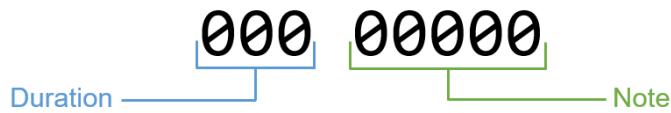


Fig. 6.2: Data distribution in cmd when a note is assigned. Source: own.

So looking at Fig. 6.2, it can see that the maximum *note* will be $2^5 - 1 = 31$.

- B. *setbase* can only take the following values: 0, 1 or 2.

In the score, at the beginning on the song every instrument set its octave using the command *setbase(baseoctave)*. The definition of this command is $(CMD_SET_BASE | baseoctave * 12)$ and *CMD_SET_BASE* is defined as 0xA0.

Every octave consists of 12 notes, so multiplying the *baseoctave* by 12 results the first note of the desired octave.

The distribution of data in *cmd*, after applying the definition, is the following:



Fig. 6.3: Data distribution in cmd when setbase command is assigned. Source: own.

The maximum value of *setbase* is $2^5 - 1 = 31$. With this value it is possible to set base 0, 1 and 2, but 3 would be $3*12 = 36$ and it will ruin the command system.



If limitation A and B are combined it becomes a more serious bug. The way that *NoteWork()* sets the note in *TSound* variable is by adding the *note* to *setbase*. Trying to do this with the maximum values results in 55. It means that there is one note that it is impossible to play (there are 57 notes, 0-56).

These three limitations can be resolved by changing *cmd* variable type from *unsigned char* to *unsigned int*.

By analysing the characteristic waveform and amplitude envelope tables, it is easy to observe that better sound quality can be obtained by increasing the resolution from 64 points to 256 points. TC275 microcontroller is powerful enough to support this change.

Reading Antonio's Project [5], it can see that he gets a good result by adding a new octave in the lowest frequencies to get a more realistic sound.

6.2. Changing *cmd* into *integer type*

To fix the limitations mentioned in previous section, *cmd* has been changed from *unsigned char* to *unsigned int*. By doing this, it is mandatory to make some adjusts in the code. The first and most important adjust is to do a re-definition of the command types as follows:

```
#define CMD_PAUSE      0x100 // 0001 0000 0000
#define CMD_PLAY_MORE   0x120 // 0001 0010 0000
#define CMD_SET_BASE    0x140 // 0001 0100 0000
#define CMD_REPEAT      0x180 // 0001 1000 0000
#define CMD_STOP        0x1C0 // 0001 1100 0000
```

Fig. 6.4: Command definition. Source: own.

In this new system, the bit that differentiates between notes and commands is now the bit 8. Also the bits that differentiate command type (repeat, stop, setbase...) changes from bits 6, 5 and 4 to bits 7, 6 and 5.

By doing these simple movements, the limitations of *cmd* have been fixed.

Now, the definition of *play(note, duration)* is (((*duration*-1) << 6) | *note*) and the distribution of data in *cmd* results the following:



Fig. 6.5: New data distribution of cmd as note. Source: own.

The maximum value of *note* is now $2^6 - 1 = 63$.

In the same way, the distribution of data in *setbase*:



Fig. 6.6: New data distribution of cmd as setbase. Source: own.

The maximum value of *setbase* is now $2^6 - 1 = 63$ so, the maximum value of *baseoctave* is 5.

To fit this change in the code, it is necessary to make some syntax changes. All variables which store commands have to be changed into an *integer type*.

In regard to *NoteWork()* operation, the following changes must be done:

- Bit selection:

```
if (cmd & 0x100) {           // this is command
```

- Command type:

```
n = cmd & 0x1C0;
```

- Set base:

```
S->cBaseNote = (char)(cmd & 0x3F);
```

- Note Frequency:

```
f = Freq[S->cBaseNote + (cmd & 0x3F)]; // Set note frequency
```

- Command and note duration:

```
n = cmd & 0x1F;
n = cmd >> 6;
```



6.3. Amplitude envelope tables

The current amplitude envelope consists of straight lines stored in a 64-point ROM table. This is an approximation but it is not as realistic as it can be since 64 points are not enough. Antonio Martínez made a great job designing a new table with a more realistic envelope and based on 256 points [5]. Hence, this approach has been used in this project.

To fix the new tables to the rest of the code it has been necessary to make a little change in the *temp2* value assignation for each channel and change it to an *integer type*.

```
temp2 = bassEnv[S1.t];
```

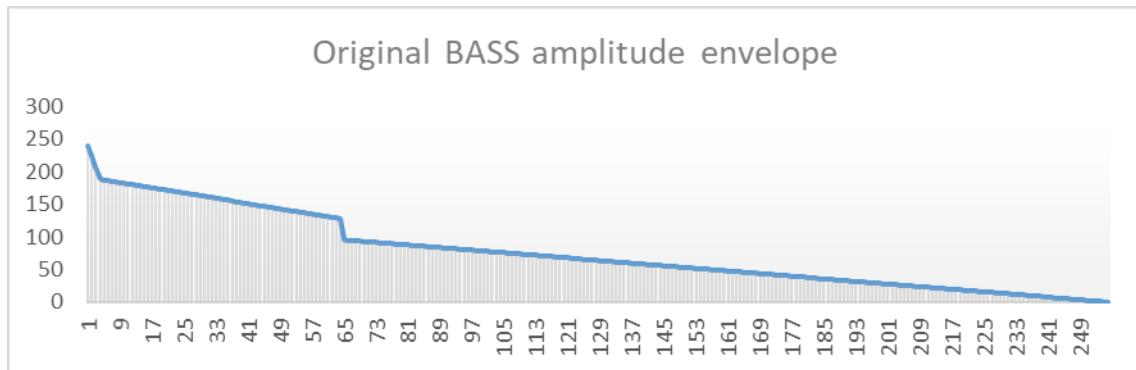


Fig. 6.7: Original BASS amplitude envelope. Source: Victor Timofeev's original Quartet code.

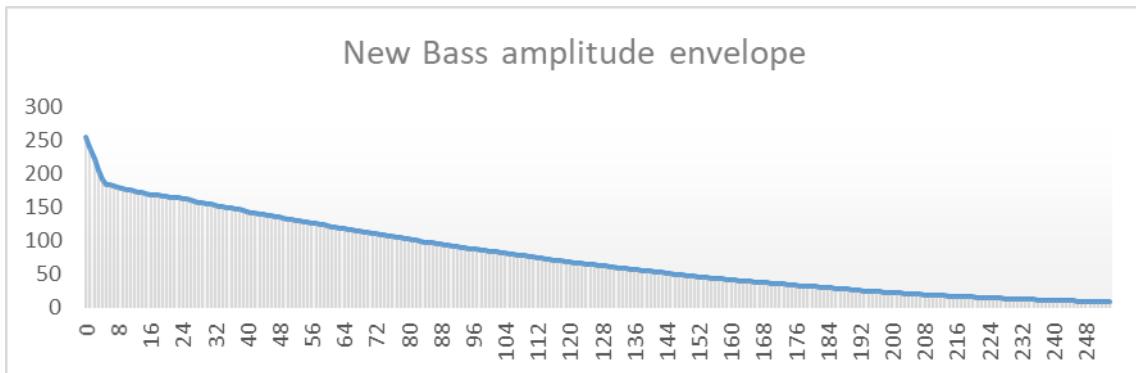


Fig. 6.8: New BASS amplitude envelope. Source: Antonio Martinez's Quartet code.

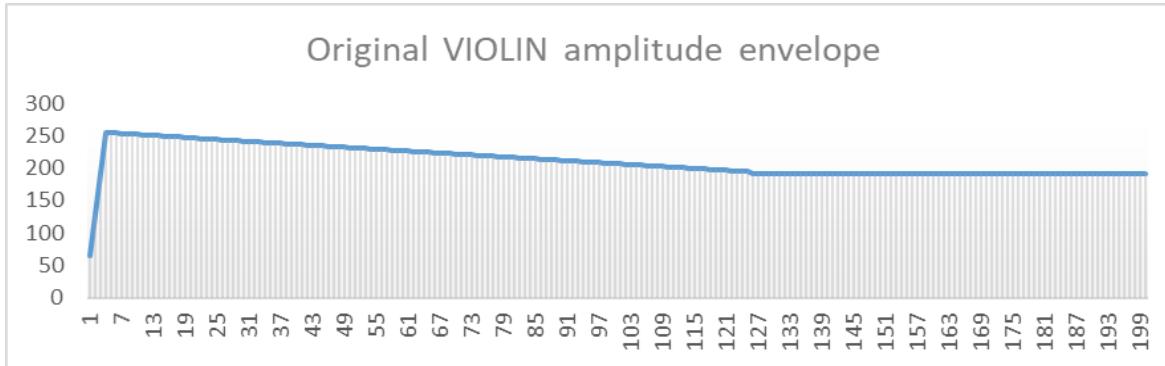


Fig. 6.9: Original VIOLIN amplitude envelope. Source: Victor Timofeev's original Quartet code.

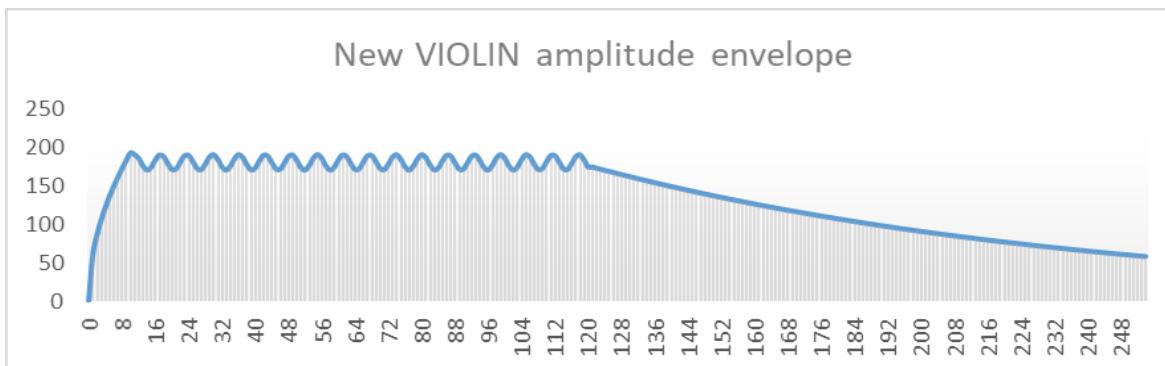


Fig. 6.10: New VIOLIN amplitude envelope. Source: Antonio Martinez's Quartet code.

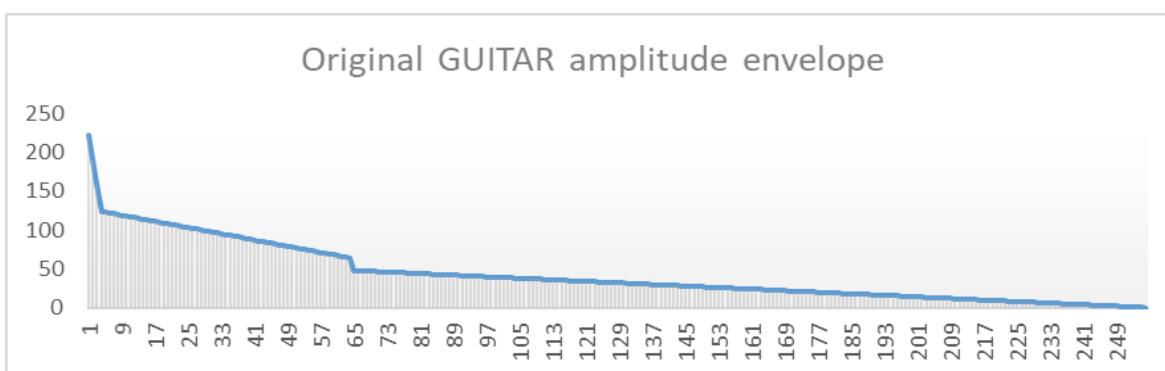


Fig. 6.11: Original GUITAR amplitude envelope. Source: Victor Timofeev's original Quartet code.



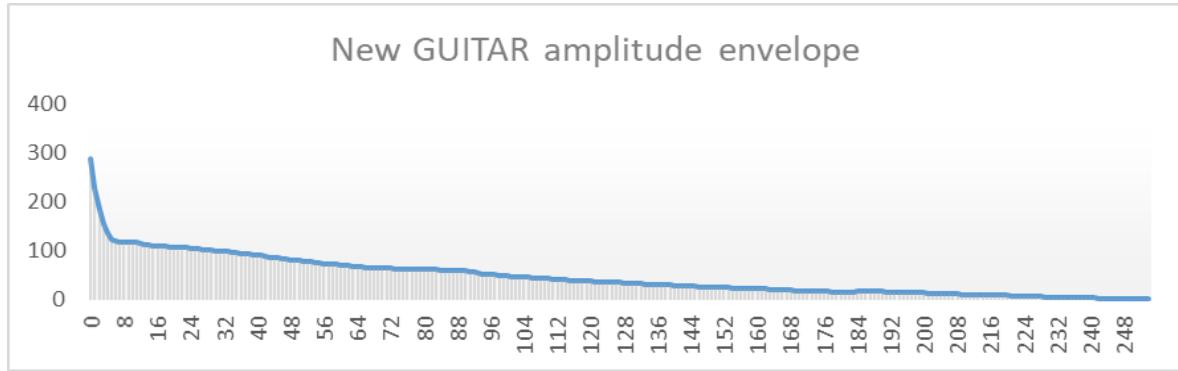


Fig. 6.12: New GUITAR amplitude envelope. Source: Antonio Martinez's Quartet code.

6.4. Characteristic waveform tables

The situation of the current characteristic waveform is similar to amplitude envelope. 64-points are not enough to get an accurate waveform. In this case, Antonio's waveform has not been used and new interpolation has been done. After doing the necessary interpolation, by splitting the waveform into different sections, a good result has been achieved.

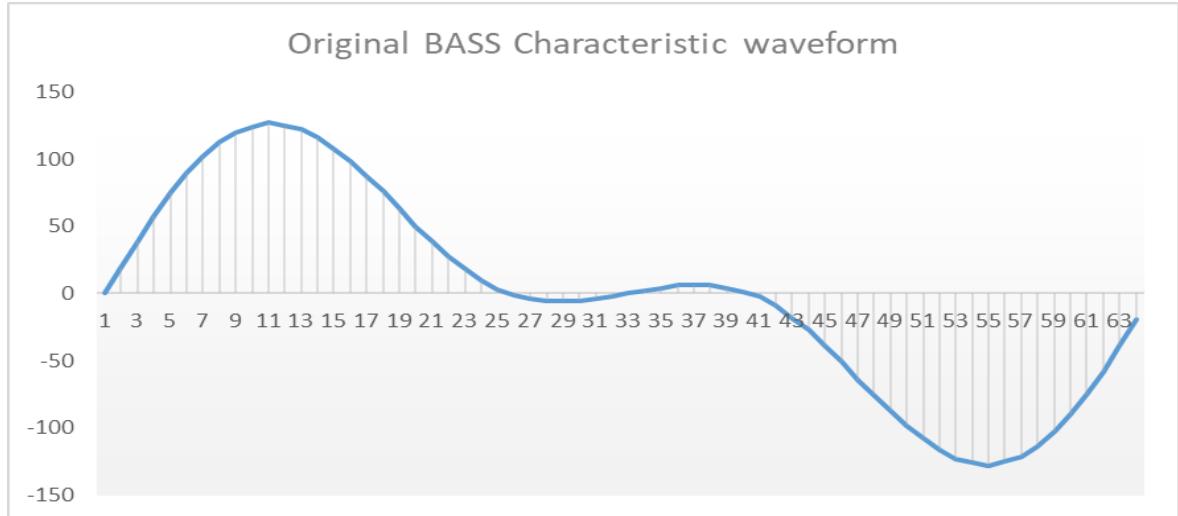


Fig. 6.13: Original BASS characteristic waveform. Source: Victor Timofeev's original Quartet code.

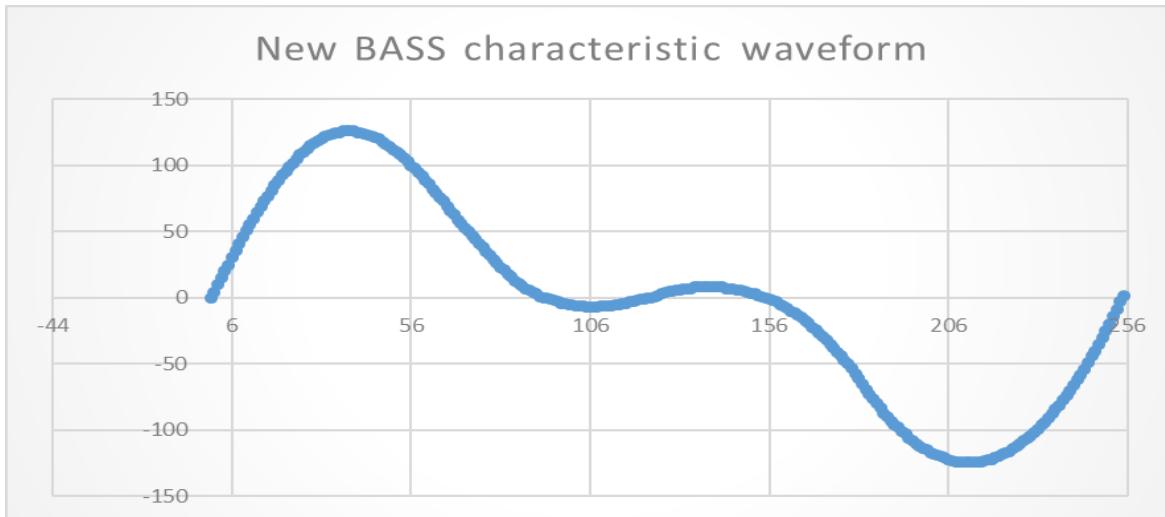


Fig. 6.14: New BASS characteristic waveform. Source: own.

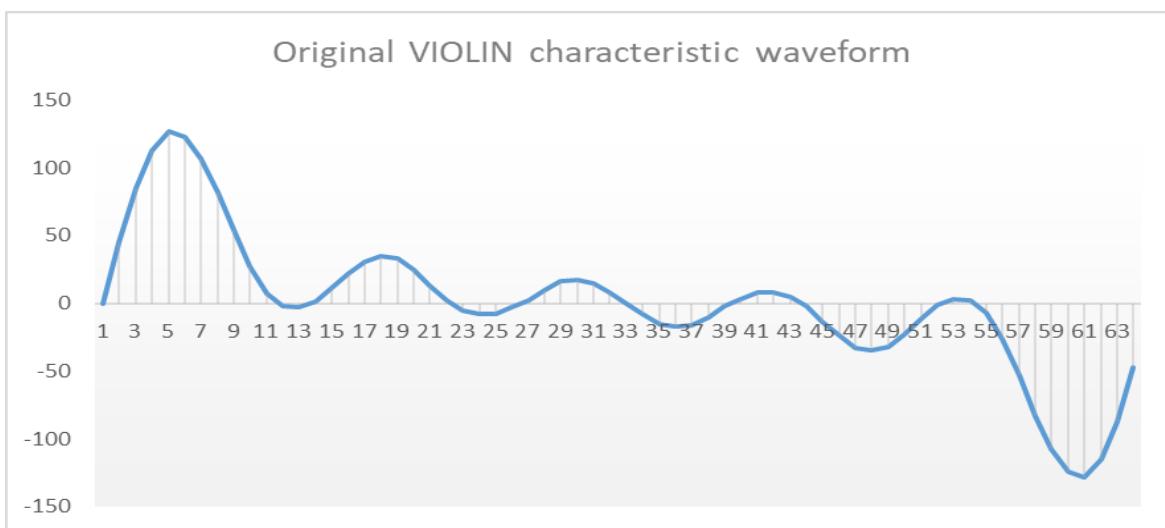


Fig. 6.15: Original VIOLIN characteristic waveform. Source: Victor Timofeev's original Quartet code.



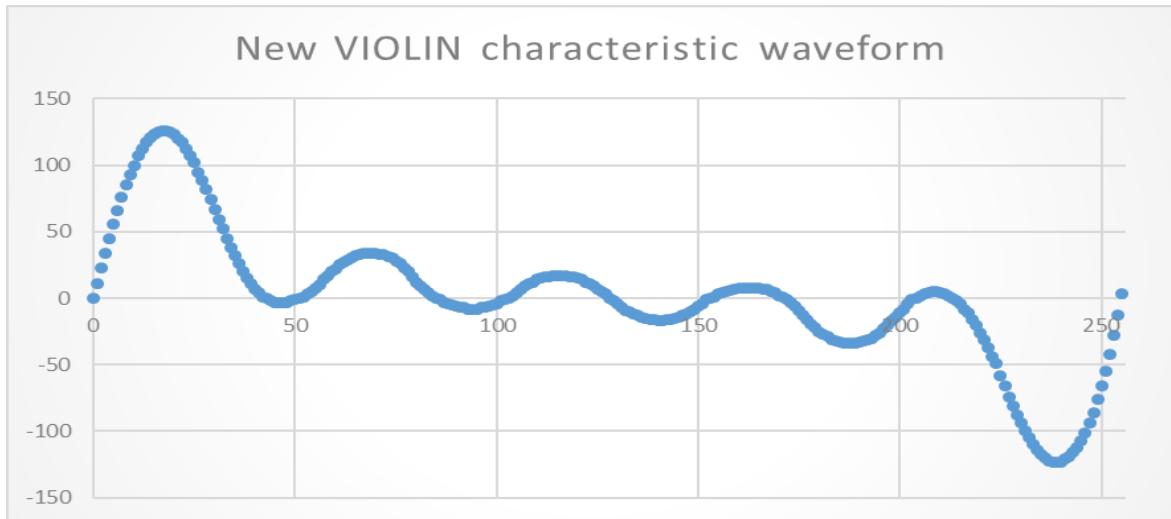


Fig. 6.16: New VIOLIN characteristic waveform. Source: own.

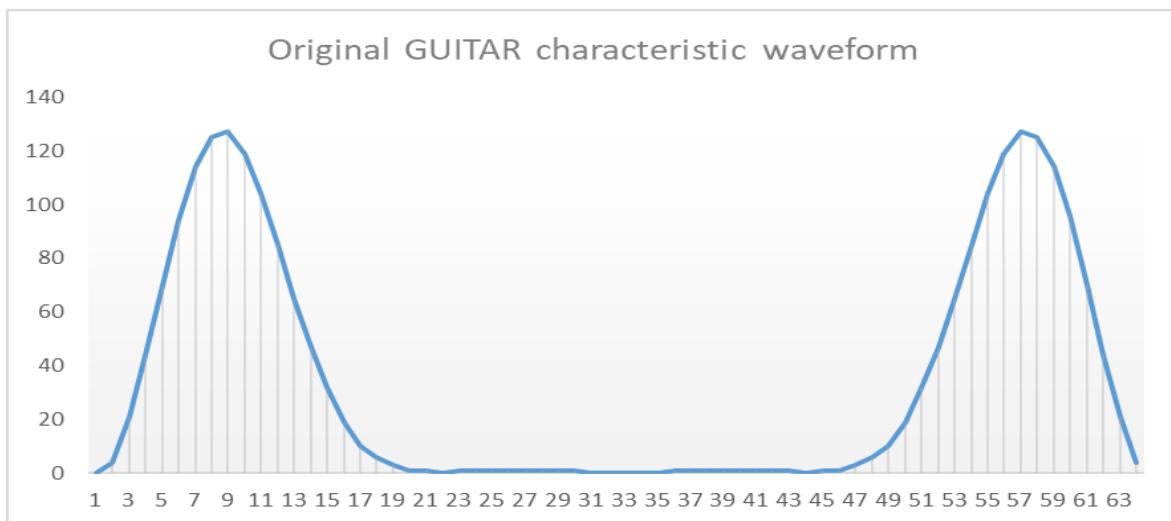


Fig. 6.17: Original GUITAR characteristic waveform. Source: Victor Timofeev's original Quartet code.

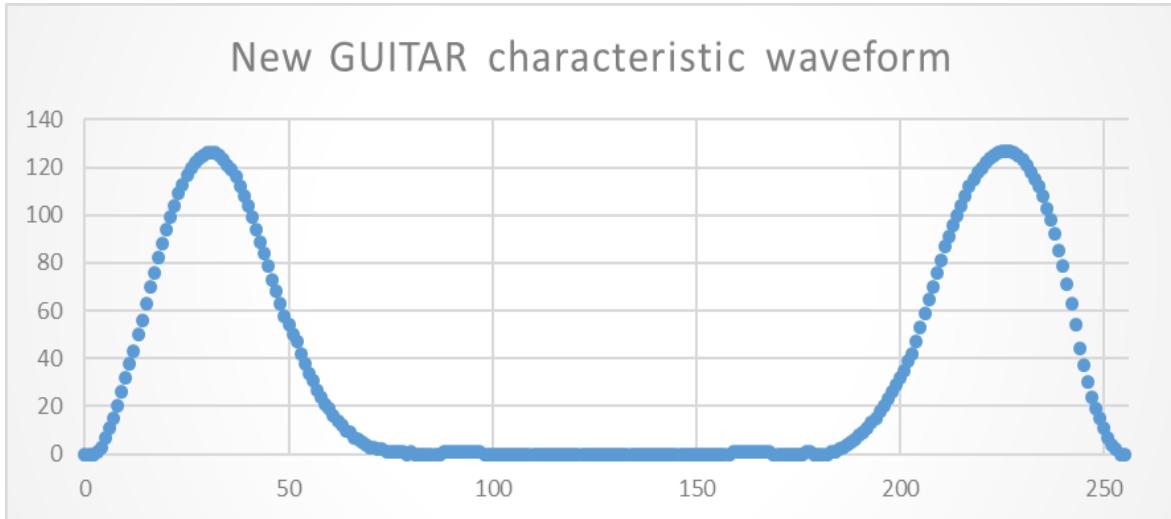


Fig. 6.18: New GUITAR characteristic waveform. Source: own.

To use these new waveform tables, it is necessary to make some changes in the code.

First of all, the frequency of the interrupt must be quadrupled. Data in the tables represents one period of the characteristic waveform. The number of points of this period has been quadruplicate so, to keep the same tempo it is necessary to read the new number of points in the same time. To get this, the frequency has to be quadruple. The interrupt service is running at 10 kHz and hence, the new one has to run at 40 kHz.

To fix the new tables to the rest of the code it has been necessary to make a small change in the *temp1* value assignation for each channel and change it to an *integer type*.

```
temp1 = bassSin[*((char*)&S1.f+1) & 0xFF];
```

There is a counter that checks that you do not skip to the next point of the envelope without passing to all waveform points. It has to be changed from 64-points to 256-points.

```
if (prs & 0x100) {
```

Finally, it is necessary to adjust *temp_dac* value to the PWM range before update it. To do that it is needed to know the maximum and minimum value of *temp_dac*, and make sure that *temp_dac* range is inside PMW range by dividing and adding an offset.

```
temp_dac = (temp_dac >>8)+200;
```



6.5. Adding an extra octave

To add an extra octave, it is only necessary to know the new frequencies to add. Antonio added a new octave in his project so the corresponding frequencies have been taken from his code [5].

Thanks to the improvements already done, there is no problem to add a new octave because *setbase* can reach up to five octaves. Therefore, the only modification to do is to add the new frequencies to the *Freq[]* array as follows:

```
const int Freq[] = {
    (int)(32.703 Hz),           //C0
    (int)(34.648 Hz),
    (int)(36.708 Hz),
    (int)(38.891 Hz),
    (int)(41.203 Hz),
    (int)(43.654 Hz),
    (int)(46.249 Hz),
    (int)(48.999 Hz),
    (int)(51.913 Hz),
    (int)(55.000 Hz),
    (int)(58.270 Hz),
    (int)(61.735 Hz),
    (int)(65.40639 Hz),        //C1
    (int)(69.29566 Hz),
    (int)(73.41619 Hz),
    (int)(77.781745 Hz),
    (int)(82.40689 Hz),
    (int)(87.30706 Hz),
```

6.6. Conclusions about sound quality improvement

A great improvement has been achieved in this section. The resolution of the characteristic waveform and the amplitude envelope have been increased and there is a more realistic representation. By changing *cmd* to an *integer type* the detected errors have been fixed and a large capacity has been generated in this variable, providing many possibilities. Moreover, by adding a lowest octave the instrument sound becomes more realistic. However, this great code improvement is not reflected at all in the sound reproduced by the headphones because the resolution of the TC275 PWM signal is only 9-bit running at 78 kHz.

7. Tri-core management

In this section it will be explained the new structure of the developed project by working with the three cores of the ShieldBuddy TC275 microcontroller. It is mandatory to get memory sharing between cores in order to develop the desired structure.

With regard to this objective, a lot of problems have been happened during the process and a workaround has had to be used. There are different ways to share memory but because of the continued appearance of errors together with the lack of complete examples and information from the manufacturer they have had to be discarded.

7.1. Project restructuring

This project is made with the aim of taking advantage of the whole capacity of ShieldBuddy TC275 microcontroller. By dividing the code into three 32-bit CPU cores you get a huge capacity to add as many features as you want.

Thinking of the new improvements can be made for the instruments, it has been decided to use one core to execute the *Tempo Part*, specifically core 0, and the *Sound Part* will be splitted into the remaining two. Beforehand, it is known that drum kit and overdrive effect will be implemented. Hence, it must be taken into account that drum kit will need some space as a new instrument and overdrive effect will be implemented in guitars and bass. After consider this, a logical distribution of the code could be:

- Core 0: *Tempo Part*.
- Core 1: Bass, Guitar1 and Guitar2 with overdrive effect.
- Core 2: Violin and drum kit.



With regard to the file structure, it is important to know that each core has its own main file (*main0.c*, *main1.c* and *main2.c*) and there is a global main (*main.c*), executes by core 0, which enables/disables cores 1 and 2. Therefore, the code has to be distributed in the three main files as follows:

- *main0.c*: *InitSoundVariable()*, *NoteWork()* and Tasks system.
- *main1.c*: *GtmTomPwmHIDemo_init1()*, *GtmTomPwmHI_initTimer1()* and *ISR_Timer1()*.
- *main2.c*: *GtmTomPwmHIDemo_init2()*, *GtmTomPwmHI_initTimer2()* and *ISR_Timer2()*.

The header files *bach1067.h* and *sinus.h* causes an error if they are included in more than one main file because they contain array variable that acquires multiple definition.

To fix this, it is necessary to understand the quartet behaviour and the header files content. The content of *bach1067.h* and *elochka.h* (which is included in *bach1067.h*) only refers to score, command definitions and variables used by *NoteWork()* so, it can only be included in *main0.c*. In the other hand, *sinus.h* contains the characteristic waveform and amplitude envelope tables of each instrument, so it is not necessary to include in *main0.c* but it has to be splitted into two files (*sinus1.h* and *sinus2.h*), each one including the corresponding instrument tables.

The element which relates *Tempo Part* and *Sound Part* are *TSound* variables. These variables are structures that contain all necessary data related to each instrument. They are modified by *NoteWork()* and read by *ISR_Timer()*, so these variables must be shared between cores. Initially *TSound* structure was defined in the main file. Since it will be used in all main projects it has been decided to create a header file called *TSound.h*, that contains the structure definition and it is included in the three main projects (it can be included in the three without causing errors because it includes only definitions, not variables). Therefore, a lot of *TSound* variables are created in the project as it can be seen in Table 7.1:

| | BASS | VIOLIN | GUITAR1 | GUITAR2 |
|-------------------|------|--------|---------|---------|
| original variable | S1 | S2 | S3 | S4 |
| main0.c | S01 | S02 | S03 | S04 |
| main1.c | S11 | - | S13 | S14 |
| main2.c | - | S22 | - | - |

Table 7.1: TSound variables in each main files. Source: own.

In addition, it is necessary to add some lines of code in each main file to share *TSound* variables among them. In *main0.c*, it will be included in the instrument tasks so any other function has to be added. In *main1.c* and *main2.c* files, after the PWM and ISR initialization and configuration, only the interrupt service is running so, it will be added a new task called *vShareMemTestTask()*, that will recipe the data from *main0.c* and update its *TSound* variables.

The following figures will be useful to sum up all these changes in header and main files and the new structure:

| FILE NAME | CONTENT |
|----------------|--|
| <i>main.c</i> | Enable/disable cores. |
| <i>main0.c</i> | <i>InitSoundVariable()</i> , <i>NoteWork()</i> and Tasks system. S01, S02, S03, S04. |
| <i>main1.c</i> | <i>GtmTomPwmHIDemo_init1()</i> , <i>GtmTomPwmHI_initTimer1()</i> , <i>ISR_Timer1()</i> and <i>vShareMemTestTask1()</i> . S11, S13, S14. |
| <i>main2.c</i> | <i>GtmTomPwmHIDemo_init2()</i> , <i>GtmTomPwmHI_initTimer2()</i> , <i>ISR_Timer2()</i> and <i>vShareMemTestTask2()</i> . S22. |



| | |
|--------------------------|---|
| <i>elochka.h</i> | Frequencies table and command definitions. Included in <i>bach1067.h</i> . |
| <i>bach1067.h</i> | Commands and notes list for each instrument. Included in <i>main0.c</i> . |
| <i>sinus1.h</i> | Characteristic waveform and amplitude envelope of BASS, GUITAR1 and GUITAR2. Included in <i>main1.c</i> . |
| <i>sinus2.h</i> | Characteristic waveform and amplitude envelope of VIOLIN. Included in <i>main2.c</i> . |
| <i>tsound.h</i> | TSound structure and memory directions (explained in next sections) definitions. Included in <i>main0.c</i> , <i>main1.c</i> , <i>main2.c</i> . |

Table 7.2: Content of each file in the project. Source: own.

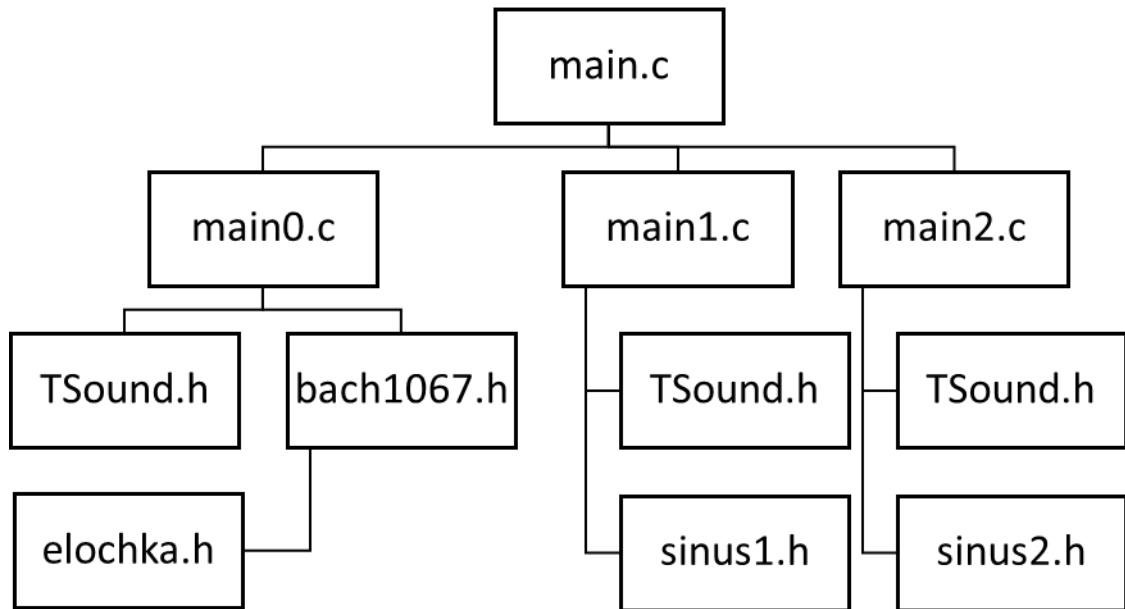


Fig. 7.1: New file structure in Quartet project. Source: own.

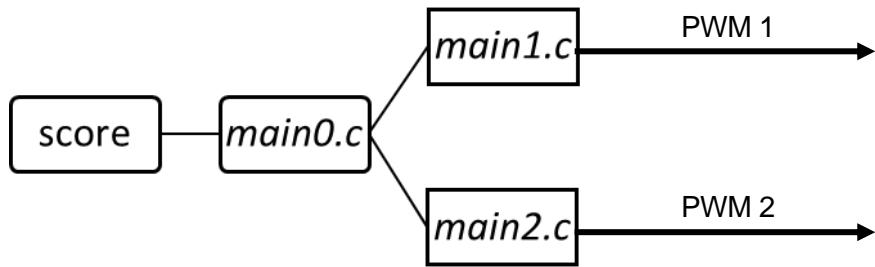


Fig. 7.2: schematic of tri-core behaviour. Source: own.

7.2. Sharing Memory

Each core has its own memory block. It is independent and other cores cannot read or write in it. Therefore, it is easy to conclude that a way to share memory between cores has to be found. Otherwise, the project will have no sense. It is not useful to have three independent cores working in parallel without communication between them.

Before starting to look for a complex way to share memory, it has been tried with the declaration of extern variables. It works with the initial value but the variable is not updated when *main0.c* changes its value. It makes sense: when extern variable is declared, the core in which is declared keeps it in his memory block. When *main0.c* changes the value, it is changing the value in core 0 memory block but it has no access to the other core memory so it cannot update the variable value.

As expected, a more complex way to share memory has to be found.

7.2.1. *SharedMem* tool

In the migrated version of FreeRTOS for Aurix TC27X, there is a tool called *SharedMem*, designed by Allan Tengg [8], the author of this version, to share memory between cores. It is the first option because it is an existing tool that belongs to the operating system FreeRTOS so it is already integrated in the project.



The way to use it is based on an example project called *TC277_FreeRTOS_Pub_original* [21] and it is simple:

First of all, it is mandatory to include in all main files the *SharedMem* header file *sharedmem.h*:

```
#include "FreeRTOS_Source/sharedmem.h"
```

To send and receive data between cores there are two simple functions that are explained in the following figures:

```
/*
 * SharedMemSend' initiates a shared memory inter core message transfer by writing the
 * given message ('data', 'len') into the shared FIFO memory and triggering an interrupt in the
 * receiving core ('dest_cpu'id').
 * A negative return value indicates a problem. A positive return value tells the number of
 * data words written in the receiving core's FIFO.
 */
short int SharedMemSend(unsigned short dest_cpu_id, unsigned int *data, unsigned short len)
```

Fig. 7.3: Description of SharedMemSend() function. Source: [8].

```
/*
 * 'SharedMemReceive' waits for an incoming shared memory message. If there is no message within
 * the specified timeout (in OS ticks), a negative value is returned. Otherwise the number of
 * words successfully transferred from the FIFO to the 'data' pointer is reported.
 */
short int SharedMemReceive(unsigned int *data, unsigned short len, unsigned int timeout)
```

Fig. 7.4: Description of SharedMemReceive() function. Source: [8].

Since these functions are really pretty simple, it has been tested directly in Quartet Project and it does not work. Lots of errors have been appeared. It was so difficult to look for the error because the debugger is extremely delicate and unstable and it is not possible to use breakpoints in core 1 and core 2 (there is a section, **9. Limitations and problems**, dedicated to explain the limitations of the development board and all tools). After a lot of time testing different ways to find what the problem was without succeed, something else was tried.

Due to the difficulties to find what is wrong in the project, it has been decided to test the example project mentioned [21] before in order to compare it with Quartet Project and check the bug. Nevertheless, *SharedMem* function does not work in the example project either. It is not possible to know for sure the reason because no error is generated. The compiler and the debugger do not find anything wrong but it is not possible to share memory among cores.

It has been looking for some information of *SharedMem* but no document was found. Finally, some help has been searched:

- It has been contacted Allan Tengg, author of the migrated version of FreeRTOS for Aurix TC27X [8] and *SharedMem* tool and he suggested that probably the linker files were wrong but after checking this, *SharedMem* function still does not work.
- A query has been published in the internet Hitex ShieldBuddy forum [22]. The administrator suggests that *lfxMemMap.h* header file was included in the project but still does not work. It was not found any solution to still working with *SharedMem* but it was said that there is a memory block that is shared by all cores.

So, it has been resigned to use *SharedMem* and it will try to use this shared memory location.

7.2.2. LMU SRAM

It has been found some information of the shared memory block location of ShieldBuddy TC275 in the document *ShieldBuddy TC275 User Manual* [20]. This memory location is called LMU SRAM and all cores have access to it.

To this document it can extract the following information:

- Multicore Memory Map

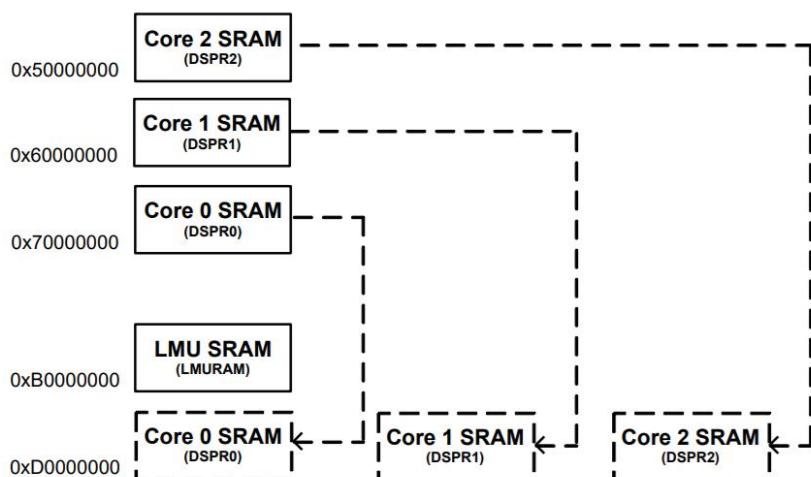


Fig. 7.5: Multicore Memory Map. Source: [20]



- How to store variables in LMU SRAM

```
/* LMU uninitialized data */
StartOfUninitialised_LMURam_Variables
/* Put your LMU RAM fast access variables that have no initial values here e.g.
uint32 LMU_var; */
EndOfUninitialised_LMURam_Variables

/* LMU uninitialized data */
StartOfInitialised_LMURam_Variables
/* Put your LMU RAM fast access variables that have an initial value here e.g.
uint32 LMU_var_init = 1; */
EndOfInitialised_LMURam_Variables
```

After trying to use this approach for global variable initialization, it has been proven that it does not work; the variable is never updated and again with no errors generated by the compiler. This manual [20] is probably thanked to use ShieldBuddy TC275 with the Arduino IDE but there is none to be used in Eclipse IDE.

By making a review of Allan's suggestion and the Multicore Memory Map, it has been seen that there is a mistake in the LMU SRAM address written in *iROM.Id* linker file. The address suggested by Allan and which is already written in *iROM.Id* is 0x90000000 but the direction indicated in the Multicore Memory Map (Fig. 7.5) is 0xB0000000.

To fix this, it is mandatory to change the address in *iROM.Im* model file. Otherwise, the next compilation another wrong *iROM.Id* linker file will be created.

After fix this configuration error, it has been thinking that this could be the problem by using the *SharedMem* tool. Nevertheless, it still does not work. In addition, using LMU SRAM with this modification does not work either.

7.2.3. Final Solution

Having the direction of a memory location that all cores have access, there is one possibility. It is not the best option but it works. It consists of making a pointer to this memory location and store data there. It is not possible to send data from one core to other. But if you want to share data from core 0 to core 1, you can create a pointer in core 0 to one address, sharing data there using the pointer, create other pointer in core 1 to the same address and use this pointer to read the value stored in there (it is the same way to share data between core 0 and core 2). The following figure will illustrate that method:

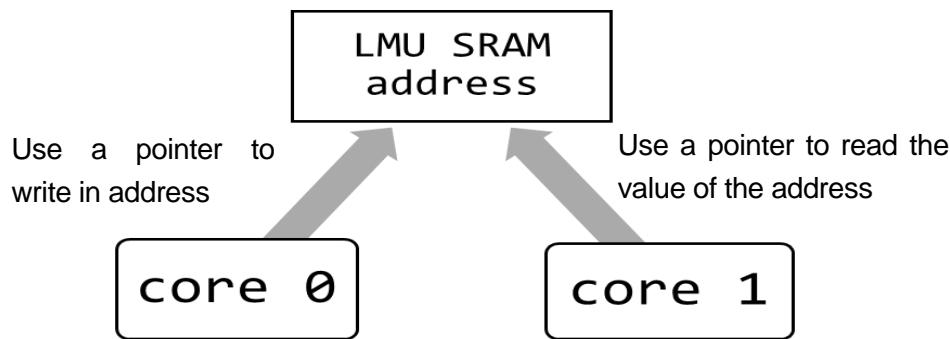


Fig. 7.6: Schematic of how to use LMU SRAM. Source: own.

It is necessary to make sure that the two cores do not access the data at the same time. To do this, it has been used a kind of semaphore built with pointers. These pointers will be stored in LMU SRAM too.

So, in total we need to store four *TSound* variables and four semaphores. One for each instrument. To make it easy, all address will be defined in *TSound.h* header file. It is mandatory to know which the size of every variable is, one byte has been left between variables as a safety margin.

```
#define SemBass      0xB0000000
#define BaseBass     0xB0000002
#define SemViolin    0xB000001B
#define BaseViolin   0xB000001D
#define SemGuit1     0xB0000036
#define BaseGuit1    0xB0000038
#define SemGuit2     0xB0000051
#define BaseGuit2    0xB0000053
```

Fig. 7.7: Address location definitions. Source: own.



The procedure to send BASS data from core 0 to core 1 (the same from core 0 to core 2 and from other instruments) using the semaphores is as follows:

- *main0.c* initializes BASS semaphore pointer and BASS *TSound* pointer.
- *main1.c* initializes BASS semaphore pointer and BASS *TSound* pointer.
- *main0.c* sets semaphore pointer value as 0.
- if semaphore pointer value is 0, *main0.c* sets *TSound* data in *TSound* pointer address and sets semaphore pointer value as 1.
- If semaphore pointer value is 1, *main1.c* reads *TSound* data stored in *TSound* pointer address and sets semaphore pointer value as 0.

For an unknown reason, if you try to apply this method with a pointer to a *TSound* variable, it does not work. In fact, it only works with a pointer to a *volatile unsigned char* type. Therefore, a new solution has to be found.

What has been done is to create an auxiliary pointer to a *volatile unsigned char* but with the address of the *TSound* variable. Now, you have a *volatile unsigned char* pointer to the address of the first byte of *TSound* variable and a *volatile unsigned char* pointer to the address of LMU SRAM. What has to be done is to go over *TSound* variable copying byte by byte into LMU SRAM. To read the value it is exactly the same way.

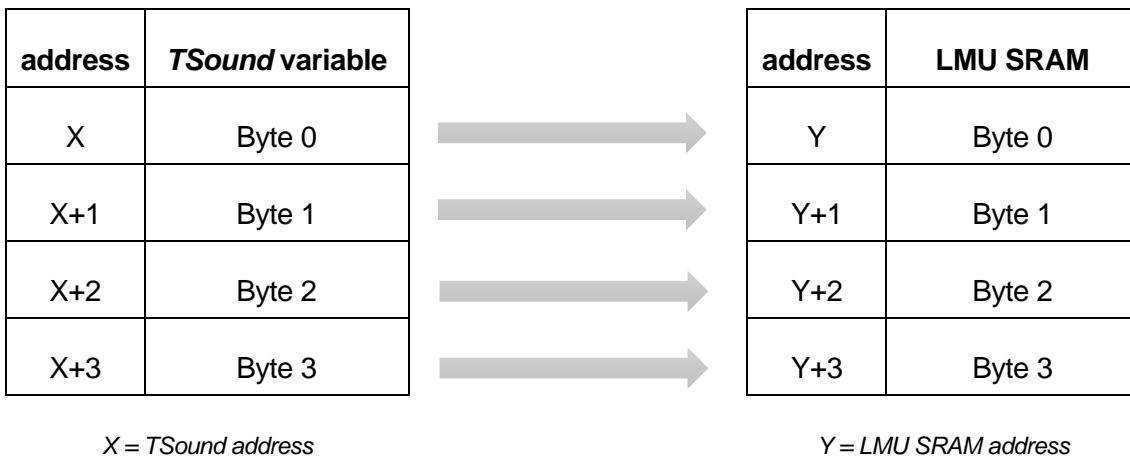


Fig. 7.8: Schematic of sending data byte by byte. Source: own.

To make easy the pointer definition, it has been made the following macro definition stored in *TSound.h*:

```
#define MemPointer(x) ((volatile unsigned char*) x)
```

The following figure is an example about how to store data in LMU SRAM:

```
unsigned char i;
volatile unsigned char *CoreSem2 = MemPointer(SemViolin);
volatile unsigned char *S2 = MemPointer(BaseViolin);
volatile unsigned char *point2;

point2 = &S02;
*CoreSem2 = 0x0;

if (!(*CoreSem2)){
    for(i = 0 ; i<24 ; i = i+1){
        *(S2+i) = *(point2+i);
    }
    S02.t = 1;
    *CoreSem2 = 1;
}
```

Fig. 7.9: Example of how to store data in LMU SRAM using pointers. Source: own.



The following figure is an example about how to read data from LMU SRAM:

```

unsigned char i;
volatile unsigned char *CoreSem1 = MemPointer(SemBass);
volatile unsigned char *S1 = MemPointer(BaseBass);
volatile unsigned char *point;
if (*CoreSem1){

    point = &S11;
    for (i = 0 ; i<24 ; i=i+1){
        if ((i==20) && (*(S1+i)==1)){}
        else{
            *(point+i) = *(S1+i);}
    }
    *CoreSem1 = 0;
}

```

Fig. 7.10: Example of how to read data from LMU SRAM using pointers. Source: own.

Finally, It is necessary to make a little modification. In the original Quartet project, the interrupt service use *S.t* (being *S* a *TSound* variable) as an index for the amplitude envelope table. *S.t* is initially set as 0, the interrupt service modifies it and when a new note has to be played, *NoteWork()* sets *S.t* as 0 to start the table.

The problem is that now, the interrupt service is in core 1 (or core 2) and *NoteWork()* is in core 0. If *main0.c* initializes *S.t* as 0, it will never be modified because interrupt service is in core 1. The project is constantly sending and receiving data so core 1 will always be receiving a 0 and it does not play.

To solve this, core 0 will initialize *S.t* as 1. It will be sending 1 until a new note has to be played. Then, it sends a 0. To avoid that *S.t* remains as 0, it is set as 1 immediately after data transfer. Thus, it only sends a 0 once in each new note.

On the other hand, if core 1 receives *S.t* equal 1, it will be ignored. So, the interrupt service routine will be the one which modifies *S.t* until it receives a 0.

7.3. Timers configuration

To generate a second PWM signal, it is necessary to generate another interrupt and a PWM. Therefore, two timers are needed. Taking into account the two cores configuration it is necessary to have four timers. It is a problem because there are three available timers in ShieldBuddy TC275. A TOM can set two channels at different frequency or in different cores. So, trying out different possibilities, the following conclusions have been drawn:

- TOM2 cannot generate a PWM signal. There is no output channels available.
- TOM0 cannot generate PWM signals and manage interrupt service at the same time.
- TOM1 can generate PWM signal and manage interrupt service at the same time, with the same timer.
- Any TOM can manage two timers at different frequency.
- Any TOM can manage two timers at different cores.

At the end, it has been used TOM2 and TOM0 in core 0; Tom2 manages the interrupt timer and TOM0 generates PWM signal. And TOM1 has been used to manage a timer who is used at the same time for interrupt service and for PWM.

PWM is running at 78 kHz and the interrupt has to be generated at 40 kHz. In fact, ISR can be set at different frequency, the point is that it has to update PWM value at 40 kHz. So the only thing to do is to set the timer at 78 kHz and use a counter to adjust the frequency of PWM update. To make it easier, the PWM frequency of the two cores has been changed from 78 kHz at 80 kHz.

The last thing to do is to configure the output pins exactly in the same way explained in section **5. Software Migration** and adding another headphone with the RC filter.



8. Overdrive effect

Thinking of future projects, it is possible that someone wants to leave classic music and made something more modern. A little step in this direction is to add a distortion effect in the guitars and the bass, specifically overdrive distortion.

8.1. Effect description

In order to change the sound of a guitar or a bass, it exist some effects achieved by modifying the waveform of the sound. There are a lot of effects: distortion, compression, “wah” effect, booster effect.... One of them, the distortion is the most well-known so it is the one that will be implemented.

There are a lot of different types of distortion: fuzz, heavy, overdrive... Of all, overdrive is a good choice because it is easy to achieve and it is not a heavy effect that can be a bit annoying.

Physically, overdrive effect consists of cutting the waveform of the sound by the upper and lower parts. There exist two types of overdrive: hard overdrive and soft overdrive. The difference is seen in Fig. 8.1.

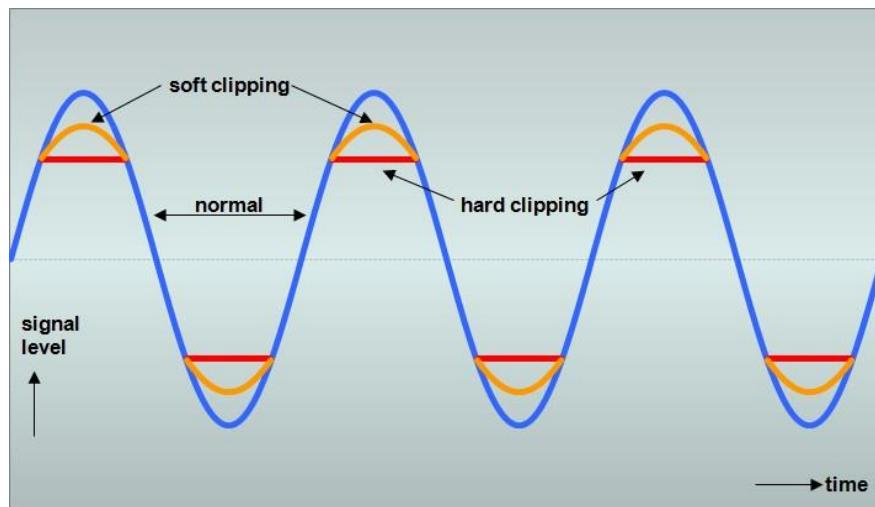


Fig. 8.1: Physical modulation of sound wave to get overdrive effect. Source: [23]

In this project, only hard overdrive will be implemented.

8.2. Overdrive implementation

The way to get the overdrive effect is by modulating the characteristic waveform of each instrument. It will be implemented in GUITAR1, GUITAR2 and BASS.

To get the overdrive effect in the code the following steps must be followed:

- It is necessary to define a new command to enable and disable effect in the score. Since *cmd* is now an integer, there are capacity to introduce a new command. In this case, it will be the following.

```
#define CMD_OVERDRIVE 0x1E0 // 0001 1110 0000

#define overdrive() (CMD_OVERDRIVE)
```

In the score *overdrive()* will appear and *cmd* will be set as 0x1E0.

- To let the instruments know if overdrive effect is enabled or disabled, a new element of *TSound* structure will be added in *TSound.h* and initialized in *InitSoundVariable()*.

```
char overdrive:1; // Overdrive enable or disable

S->overdrive = 0;
```

- Now, *NoteWork()* has to recognize the command. The way that recognizes the different types of commands is by doing *n* = *cmd* & 0x1C0 and a *switch* case with the variable *n*. In this case, *n* will be the same that in *CMD_STOP*, so inside *CMD_STOP* case it will be an *if* statement to recognize that it is an overdrive command. When it is recognized, it will complement its value. The idea is to use the same command in the score to enable and disable effect.

- ```
if (cmd & 0x20) S->overdrive = ~S->overdrive;
```
- Finally, the synthesizer has to know if it is enabled or disabled and modulates waveforms in case that it is enabled.

```
// READING SINUS
temp11 = bassSin[*((char*)&S11.f+1) & 0xFF];
if (S11.overdrive){
 if (temp11 > 100) temp11 = 100;
 else if (temp11 < -100) temp11 = -100;
}
```



## 9. Limitations and problems

It is important to mention that this project has been full of significant problems and conditioned by some important limitations.

It has to be noted that the ShieldBuddy development board used in this project is very cheap and the tools are free and limited, with no support at all from the manufacturer.

The free version of Eclipse IDE is also hard to use. It is not an intuitive environment to configure the project and work. It is extremely delicate. Sometimes it breaks without sense and the error does not disappear even if you undo it and you have to delete the whole project and continue from the last backup copy if you have.

Often, the debug configurations change without notice and it does not work. The debugger does not allow you to use breakpoints in core 1 and core 2, so if you have to analyse a variable evolution or something, you have to use your imagination and play with LEDs to know what is happening. In core 0, there are code lines that inexplicably you cannot use breakpoints. Often, you cannot see local variables because the window of views/locals is blocked or in blank. Similar error occurs with global variables. Sometimes it does not update its value in views/watch window until you close and reopen the debugger.

Since there was a bug in FreeRTOS configuration, the memory map was wrong. Despite this, there is no logical reason to explain why *SharedMem* does not work. The example project [21] works but not in this microcontroller.

It has been asked for help to Allan Tengg, author of the migrated version of FreeRTOS for Aurix TC27X [8], and to internet Hitex Shieldbuddy forum [22] and nobody knows what is wrong.

In addition, there is not enough information. The few example projects that use *SharedMem* or Frameworks are too simple and they do not clarify anything. Data sheets and user manual that have been found do not help to configure GTM\_TOM beyond the basics and there is none that explains how to initialize variables stored in LMU SRAM without using pointers.

Due to all these problems and limitations, it has been decided that drum kit implementation will not be included in this project.

## 10. Future Improvements

In this chapter, suggestions for future improvements of the Quartet project will be presented.

### 10.1. Inter-core communication

One of the future improvements may be to improve the communication between cores. In this project this limitation has been resolved with a workaround.

Fix this will be a great improvement because working with pointers to a pre-defined memory address is too complicated. Some ideas to improve this situation are the followings:

- Success in using *SharedMem* or find a way to initialize variables in LMU SRAM directly if the project is made for the ShieldBuddy TC275.
- Using another multi-core microcontroller in order to use its predefined way to share memory between cores.
- Taking advantage of that, the project can be splitted into three parts. Another idea could be to work with different microcontrollers communicated with different communication protocols such as Wi-Fi or Bluetooth, for example.

### 10.2. Adding new instruments or effects

Using a 32-bit three-core microcontroller provides a huge capacity to add a lot of functionalities. There are resources to spare and the new *cmd* as an integer generates a lot of space to devise. Therefore, some possibilities to improve the project are:

- Adding new voices with an existing instrument.
- Adding a new instrument type such as saxophone or trumpet with its waveform.
- Adding a new effect such a fuzz distortion or “wah” effect.



### 10.3. Drum kit

As it is mentioned in the introduction, one of the objectives of this project was the implementation of a basic drum kit. Nevertheless, the significant number of hours spent trying to fix the errors that have been happening when sharing memory between cores have caused that there is no time to develop this point.

However, some work has been done in this direction. It will be explained here as a base for future projects:

The basic elements needed to include in a drum kit are: Bass drum, snare drum and hit hat. It can be more elements but these three are the essential to get a basic rhythm.

It is mandatory to include drum kid as a new voice channel. To do it you have to add in the *main0.c* the next elements and use it exactly in the same way that the other voices: drum kit *TSound* variable, a drum kid flag, a task and semaphore.

It is necessary to write a score to the drum kit. To do it, the following definitions are added in *elochka.h*:

|                                   |                      |
|-----------------------------------|----------------------|
| <code>#define bombo()</code>      | <code>(0x200)</code> |
| <code>#define redoblante()</code> | <code>(0x201)</code> |
| <code>#define hihat()</code>      | <code>(0x202)</code> |

And it has to add in *NotewWork()* the following lines:

```
//-----
else if (cmd & 0x200) { //this is drumkit
 i = (char)(cmd & 0x3); //bit select to identify element type
 f = DrumFreq[i]; //frequency of the element
 S->cBaseNote = i; //it use Basenote to store the element type
 S->bEnable = 1;
 S->F = f;
 S->t = 0;
 n = 1;
 break;
}
```

*Fig. 10.1: Notework() extension to use drum kit command. Source: own.*

To share data, the drum kit address in LMU SRAM will be set and it will be shared with pointers as the other voices:

|                                  |                         |
|----------------------------------|-------------------------|
| <code>#define SemDrumkit</code>  | <code>0xB000006C</code> |
| <code>#define BaseDrumkit</code> | <code>0xB000006E</code> |

Now, there are some elements missing that cannot be investigated due to lack of time. It is necessary to find the different elements frequencies, characteristic waveforms and amplitude envelopes. Although these values are missing, a way to use it will be proposed:

- In *elochka.h* it is necessary to include the frequencies. It has been proposed to declare an array called *DrumFreq[]* and store the frequencies in the same order as the command definition sets (bassdrum, snaredrum, hihat).
- In *sinus2.h*, the amplitude envelope and characteristic waveform tables have to be included. A proposed way to do it is to use an array of arrays. An array called *drumkidSin[]* that contains the three arrays of characteristic waveform (the same in case of amplitude envelope).
- Finally, to synthesize sound in the same way to the other voices. The only difference is that now you have to access to an array or arrays. First, you have to use the index stored in *setbase* to access the desired element and then, the same index than other voices.

## 10.4. Other ideas

There are some small changes that can suppose a significant improvement of the project:

- Use the proposed improvements in this report with a microcontroller that has a wider PWM resolution. 9-bit PWM resolution is not enough to appreciate the new sound at all.
- Widening the range of sound duration. It is not necessary in this project but it would be.
- Changing the concept of the Quartet Project to reproduce a modern song. For example, to reproduce a rock song. There are two guitars and a bass with the possibility to play with overdrive, it can be a drum kit and the singer voice can be simulated by violin.



## 11. Budget

| <b>Hardware</b>          |              |                       |                        |
|--------------------------|--------------|-----------------------|------------------------|
| <b>Concept</b>           | <b>Units</b> | <b>Unit price [€]</b> | <b>Total price [€]</b> |
| <b>Shieldbuddy TC275</b> | 1            | 115,18                | 115,18                 |
| <b>headphones</b>        | 2            | 8                     | 16                     |
| <b>PC (amortization)</b> | 1            | 50                    | 50                     |
|                          |              |                       | 181,18                 |

| <b>Work hours</b> |                         |                        |
|-------------------|-------------------------|------------------------|
| <b>Units [h]</b>  | <b>Unit price [€/h]</b> | <b>Total price [€]</b> |
| <b>300</b>        | 40                      | 12000                  |
|                   |                         | 12000                  |

| <b>Energy consumption</b> |                           |                        |
|---------------------------|---------------------------|------------------------|
| <b>Units [h]</b>          | <b>Unit price [€/kWh]</b> | <b>Total price [€]</b> |
| <b>300</b>                | 0,14                      | 42                     |
|                           |                           | 42                     |

|                                  |                   |
|----------------------------------|-------------------|
| <b>Total cost of the project</b> | <b>12223,18 €</b> |
|----------------------------------|-------------------|

*Table 11.1: Project Budget. Source: own.*

## 12. Environmental impact

The environment impact of this project can be evaluated taking into account three factors; the way generated by the electronic components, the consumption of electrical power used and the produced acoustic impact.

Regarding electronic components, they comply with the RoHS directive [24]. This means that they have no lead and their environmental impact is low.

Electric power consumption rate is almost negligible to run Quartet Project so the impact is insignificant.

Finally, the acoustic impact produced by playing this project is moderated for the human being.

Thus, it can be concluded that this project has a minimal environmental impact.



## Conclusions

Regarding the objectives of the project, we can draw satisfactory conclusions:

- The code has been successfully migrated from Microchip PIC24 to Infineon TC275 and a functional program has been achieved.
- Although only a 9-bit PWM is available, several sound quality modifications have been implemented.
- In spite of the significant errors and limitations, it has managed to share memory between cores.
- The resultant project is a 32-bit three core microcontroller generating two 9-bit PWM signals running at 80 kHz. It means that there is a very powerful board running the program with capacity to spare.
- The overdrive effect has been successfully implemented and clearly appreciable in the bass and two guitars.

Nevertheless, it is a pity that it could not finish the drum kit implementation because of the lack of time. But it has been proposed a way to do it and it would be the natural continuation of this project.

It is important to note that a lot of time has been lost because of the lack of documentation, manufacturer support and useful examples. The ShieldBuddy development board is more than enough powerful to do more and better improvements and there were many ideas about it but it has become so hard due to the lack of information and the limitation of the tools. In fact, it has been considered to change to the Arduino IDE before finding the mentioned solution to share memory between cores.



## Thanks

It is mandatory to thank Pau Mendieta [3], Antonio Martinez [5] and Allan Tengg [8] for this previous works on which this project is based.

It has to thank Allan Tengg also for answer the help request sent in order to understand where the error is and for propose a solution although in the end it did not work.

Finally, it has to thank Manuel Moreno Eguilaz to help with dealing with the errors that have been happening and the tool limitations that could have ruin this project.



# Bibliography

## Bibliographic references

- [1] VICTOR TIMOFEEV, OSA RTOS, [<http://www.picosa.narod.ru/>].
- [2] PERE DOMENECH, *Aplicaciones musicales del sistema operativo en tiempo real OSA RTOS*. Barcelona 2015.
- [3] PAU MENDIETA, *Audio application based on FreeRTOS Operating System*. Barcelona 2017.
- [4] REAL TIME ENGINEERS LTD., *FreeRTOS Official Website*, [<http://www.freertos.org/RTOS.html>]
- [5] ANTONIO MARTINEZ, *Music Quartet based on a PSoC*. Barcelona 2017
- [6] JUAN GALLOSTRA, *RF Music Festival: orquesta basada en microcontroladores PIC18 y RF*. Barcelona 2015.
- [7] JOAN CALVET, *CAN Music Festival: orquesta basada en microcontroladores PIC18 y bus CAN*. Barcelona 2016.
- [8] ALLAN TENGG, *FreeRTOS migration for AURIX TC27x*, [<https://interactive.freertos.org/hc/en-us/community/posts/210026366-FreeRTOS-7-1-Port-for-Aurix-TC27x-using-Free-Entry-Toolchain>, 15/03/2018] .
- [9] TUTORIALS POINT, *C programming*, [<https://www.tutorialspoint.com/cprogramming/>, consulted on February 2018].
- [10] HIGHTEC, *Getting started to Free TrCore Entry Tool Chain.AURIX family and AUDO Future, AUDOMAX*. Germany 1991-2017. V 4.08.00.02
- [11] HITEK, *AURIX Microcontroller Software Development. User Manual SW Framework Tools*. U.K. 2014. V 3.0

- [12] HIGHTEC, [<https://hightec-rt.com/en/company/about-us.html>, consulted on February 2018]
- [13] ELECTRONICS TUTORIALS, *Pulse Width Modulation*, [<https://www.electronics-tutorials.ws/blog/pulse-width-modulation.html>, consulted on June 2018].
- [14] INFINEON TECHNOLOGIES AG, AURIX. *TC275x D-Step 32-Bit Single-Chip Microcontroller. User's Manual*. Germany 2014. V 2.2
- [15] *BaseFramework\_TC27xC\_iLLD\_GtmTomPwmHIDemo*.  
[[www.hitex.co.uk/fileadmin/uk-files/downloads/ShieldBuddy/ExampleCode.zip](http://www.hitex.co.uk/fileadmin/uk-files/downloads/ShieldBuddy/ExampleCode.zip), 30/03/2018].
- [16] MICHAEL BEACH. HITEX, *AURIX unleashed. Getting started with AURIX*. U.K. 2015.
- [17] REAL TIME ENGINEERS LTD., *FreeRTOS Reference Manual*. 2016. V 9.0.0
- [18] RICHARD BARRY, *Mastering the FreeRTOS Real Time Kernel - A Hands-On Tutorial Guide*. 2015.
- [19] RICHARD BARRY, *Using the FreeRTOS Real Time Kernel - A Practical Guide*. 2010.
- [20] HITEX, *ShieldBuddy TC275 User Manual*. U.K. 2015. V 1.5
- [21] *TC277\_FreeRTOS\_Pub\_original*.  
[[https://interactive.freertos.org/attachments/token/S8GhjRxiQYaYdkzJvHefvERRX/?name=TC277\\_FreeRTOS\\_Pub.zip](https://interactive.freertos.org/attachments/token/S8GhjRxiQYaYdkzJvHefvERRX/?name=TC277_FreeRTOS_Pub.zip), 01/03/2018].
- [22] HITEX, *HITEX ShieldBuddy forum*, [<http://arduino.boards.net/forum>, consulted on April 2018].
- [23] GUITAR GEAR, *Blog: On Choosing a Drive Pedal...*  
[<https://guitargear.org/2012/02/17/on-choosing-a-drive-pedal/>, consulted on May 2018].
- [24] EUROPEAN COMMISSION. ENVIRONMENT, *The RoHS Directive*.  
[[http://ec.europa.eu/environment/waste/rohs\\_eee/index\\_en.htm](http://ec.europa.eu/environment/waste/rohs_eee/index_en.htm), consulted on June 2018]



## Complementary bibliography

CUERDABIERTA, *Conoce Distintos Tipos De Distorsiones.*

[<https://www.youtube.com/watch?v=VY3WdNT3dPA>, consulted in April 2018]. [YouTube video].

LADOMICILIO, 1. *Partes de la Batería: Sonidos, Materiales y Tamaños (Curso de Batería).* [<https://www.youtube.com/watch?v=aZkrxP9iFt4&t=20s>, consulted in May 2018]. [YouTube video].