NTNU – Trondheim
Norwegian University of
Science and Technology

TDT4258 Low-Level Programming
Laboratory Report

Exercise 2

Group 5:

Marius C. K. Gulbrandsen
Antoni Climent Muñoz
Andrea Mazzoli

October 18, 2019

Contents

1 Introduction

In this report it will be explained two different methods of performing the same task. The task was to generate sounds using the board EFM32GG by pressing the buttons on the joystick. The code is written in the C language and the final goal will be to use these sounds and melodies in a game. Creating the game is not part of this task.

The two methods use different features and improvements to obtain the same final results. In the first chapter, it is explained two of the general features that are common to both solutions. Other improvements used will be treated in the next chapters.

1.1 Folder Structure and Makefile

Particular attention was paid to the organization and structure of the files of the project. There's one folder for each of the two solutions that are implemented, called "polling" and "interrupt". These folders has a structure of seperating the *build* files, *header* files, *source* files and *linkerscript*. In addition there is a folder called *vscode*, this contains the workspace for Visual Studio Code that makes working with the project files easier. The folder structure is shown in figure 1.1.

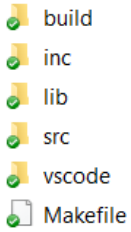

Figure 1.1: Folder structure.

1.2 Sound Wave Generation

In order to generate sound, a square wave is created by writing a positive value to the registers DAC0_CH0DATA and DAC0_CH1DATA for a certain amount of time, and then writing a low value for the same time period. There are three predefined levels of volume that are given. This is realized simply by changing the positive value that correspond to the amplitude of our wave.

The following function was used to generate the sound. It toggles the value of registers between *volume* and 0. The variable volume correspond to the amplitude of the wave.

```
1  void dac_square_wave(uint8_t volume)
2  {
3      static bool state = true;
4
5      if (state){
6          *DAC0_CH0DATA = volume;
7          *DAC0_CH1DATA = volume;
8          state = false;
9      }
10     else{
11         *DAC0_CH0DATA = 0x00;
12         *DAC0_CH1DATA = 0x00;
13         state = true;
14     }
15 }
```

dac.c

2 Solutions

In this section we describe the two solutions that was realized. The first one is based on polling. It uses busy-waiting to generate three different sound effects and one start-up melody. Each one of these sound tracks are realized using two arrays, one containing the different tones to play, while the second array contains the duration of each tone. The implementation of the sound track will be different in the interrupt version of the program. The four different sound tracks are mapped to their own button on the joystick.

The second solution was implemented using the interrupt concept. We decided to have three different sound effects and one start-up melody that starts when the board is turned on and when a specific button is pressed. In this case we used a more complex, but at the same time more compact, way to write the sound track in the program by utilizing multidimensional arrays.

2.1 Baseline Solution

For the Base Line solution, buttons are tracked with a loop by reading from the GPIO_PC_DIN register. Masks are used in order to know which button is pressed. When a button is registered, the desired track is copied into two vectors in the main. In one of them, the notes to be played are stored, and in the other, the duration of notes. The program stops checking the buttons state and goes to the next loop, which plays the desired sound.

A square wave form is used in order to make the notes, and measuring time between changes in the wave and between notes is done by counting the number of clocks. It is reached with the help of the TIMER1_CNT register, that keeps track of the clock. In the loop we will be continuously reading this register and storing the difference with respect to the last check. This way it is known how many cycles has happened since the last time the note, or the note wave was changed.

When the desired time has passed, the only thing left to do, is to go to the next note or to change the waveform. At the end of the track vector, a “0” is stored. That way, it’s known when the track is finished. When it finishes, it goes back to poll the button states. The connection between all the states in the program is shown in figure 2.1.

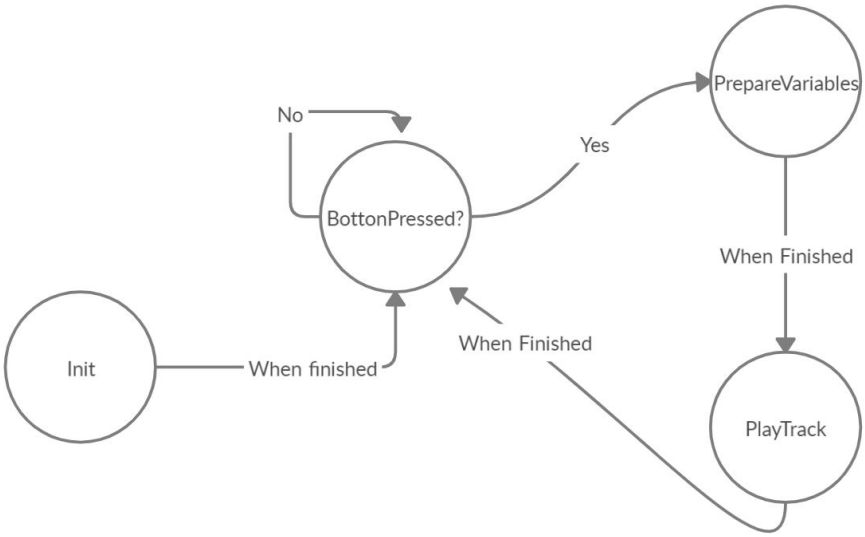

Figure 2.1: State Machine Pulling

2.2 Improved Solution

2.2.1 Use of Multidimensional Arrays

For storing the songs we utilized two dimensional arrays. These arrays store the note and the duration of it. The duration is decided with defined intervals called *bars*. These are defined with the name HALF, FOURTH, EIGHTH etc. as shown in the code below. The value "-1" terminates the song. These songs are defined in *melodies.c* and the contents are not directly accessible from main.c.

```
int end_game_sound[][2] = {
    {G3, HALF},
    {D4, FOURTH},
    {B3, HALF},
    {-1, -1}
};
```

Array of a song.

2.2.2 Pointers to Songs

To access the songs, we created a global pointer that always points to the selected song. To select the song we call the function *melodies_play()* shown in the code below.

```
void melodies_play(int *song)
{
    song_ptr = (int *)song; // Point to the selected song
    song_finish = false;
```

```
}
```

Select which song or sound to play.

This function takes the array with the sound you want to play as a parameter, this is the address of the array. The address is then passed to the global *song_ptr* so the timer interrupt handler now has access to the sound.

2.2.3 Timer Interrupt

For generating the sound waves we use a timer interrupt. The timer interrupt is scaled in such a way that it executes 44164 times a second. To generate a sound there is a counter within the interrupt handler that toggles the sound wave with *dac_square_wave(uint8_t volume)* for the amount of times necessary to create the desired tones.

The counter checks the contents of the array by adding to the pointer with separate counter variables. There are two variable, one odd and one even. By incrementing them by two, we will always point to the next note and the next duration of it. When the pointer points to the "-1" it terminates the song, and does not generate a sound wave by setting *song_finish* to true.

```
    // Sets the length of the tones
    if (iterations >= (*(song_ptr + tone_length)) && !song_finish) {
        tone_length += 2;
        tone_selection += 2;
        iterations = 0;
    }

    // Sets the tones
    if (count >= (*(song_ptr + tone_selection)) && !song_finish) {
        count = 0;

        // Generate soundwave
        if (*(song_ptr + tone_length) != -1) {
            dac_square_wave(STD_VOL);
        } else {
            // Reset pointing location
            tone_selection = 0;
            tone_length = 1;
            count = 0;
            iterations = 0;
            song_finish = true;
        }
    }
```

Timer Interrupt Handler.

2.2.4 Sound Effects and Songs

There is a start-up melody that plays upon initialization of the microcontroller. Additionally, there are sound effects that are mapped to individual buttons. The start

up melody is mapped to button number 8, so it's possible to play it again if desired. The buttons trigger an interrupt handler for odd and even buttons. Because we care about all the buttons, we created a *gpio_handler*, which is merely a function called in both the odd and the even gpio interrupt handlers. When the interrupt is triggered the *GPIO_PC_DIN* register is "*anded*" with a button mask. If it equals zero, this button was pressed, and it plays the corresponding melody as shown in the code below.

```
1  void gpio_handler()
2  {
3      gpio_map_to_led();
4
5      // Button 1
6      if ((*GPIO_PC_DIN & BUTTON1) == 0)
7          melodies_play((int*)score_sound);
8
9      // Button 2
10     else if ((*GPIO_PC_DIN & BUTTON2) == 0)
11         melodies_play((int*)end_game_sound);
```

Register button press and play sound.

Additionally, the LEDs are mapped to the corresponding button press by calling the function *gpio_map_to_led()*, which directly sets the values of the buttons to the LED register. All interrupt flags are cleared at the end.

2.2.5 State Machine

The state machine for the implementation is shown in figure 2.2.

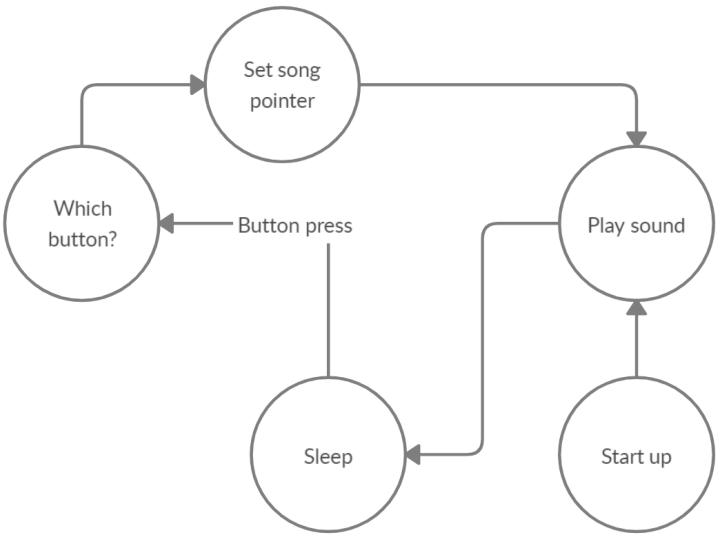

Figure 2.2: Interrupt State Machine.

3 Implementation of energy modes

First of all we have to notice that when we talk about energy modes we are referring to the interrupt version of the solution. The Energy modes are controlled by the Energy Management Unit (EMU). The EMU manages all the low energy modes in the EFM32GG microcontroller. Each energy mode manages clocks and how much of the various peripherals are available. We have 5 energy modes in total. Here we will see the implementation of EM0, EM1, and a note about the EM2.

3.1 Energy mode 0

Running the interrupt solution with no energy saving implemented it draws 4.87mA as show in in figure 3.1.

Figure 3.1: Current draw in energy mode 0.

3.2 Energy mode 1

Implementing energy mode 1 puts the processor to *sleep*. The measured current draw in this mode was 3.43mA shown in figure 3.2. That is sure less than the previous EM0, but the saved energy is not a lot.

Something to note is that to have a clear code we have implemented the function *sleep();* that allow to call the function wfi to go in deepsleep mode (the code is reported below). The *sleep();* function is called in the *while* loop in the main as reported in this code:

Figure 3.2: Current draw in energy mode 1.

```c
int main(void)
{
    // Initialize
    gpio_init();
    dac_init();
    timer_init();
    interrupt_init();
    melodies_init();
    low_energy_init();

    // Loop
    while (true) {
        sleep();
    }

    return 0;
}
```

main.c

```c
void sleep()
{
    __asm__("wfi"); // Sleep
}
```

sleep(); function

3.3 Energy mode 2

We tried also to implement the EM2 mode setting *SCR=0x6 in order to have less power consumption. This worked with only 1.6μA current draw, but the problem was that even if energy was reduced, no sound was played because timer was inactive using EM2.

4 Conclusion

The implementation of a system for playing sounds and melodies by pressing certain buttons benefited a lot from introducing an interrupt routine in terms of energy saving and implementation of songs. Introducing the EM1 does not have a large advantage over EM0 and the EM2 does not work as it disables the timer. The only big feature that can be added to improve the energy efficiency is to work with the DMA that is able to feed the DAC without using the CPU.
We tried to implement the low energy timer without having success. Still, we think that the low energy timer might not be fast enough to be utilized as a good sound generator.

The code for the polling approach can be found in the folder ../2_embedded_c/polling/ and likewise for the interrupt approach in ../2_embedded_c/interrupt/. Each of the approaches have their own makefile and should work independently of each other. The code compiles October 18, 2019with no errors and no warnings for both solutions.