NTNU
Kunnskap for en bedre verden

Department of Computer Science

TDT4258 - Low-level Programming

Assignment 2

Group Number: 14
Board Number: EFM17/3

Group Members:
Stian Strømholm
Håkon Kindem
Stian Hubred
Torbjørn Bratvold

Oktober, 2020

Table of Contents

List of Figures **i**

List of Tables **i**

1 Introduction **1**

2 Detailed tasks **1**

2.1 Interrupts 5

2.2 Polling 7

3 Energy consumption **8**

4 Discussion **9**

5 Conclusion **11**

Bibliography **12**

Appendix **13**

A Appendix A 13

List of Figures

1 Interrupts idle, deep sleep 9

2 Melody, with timer interrupts 10

3 Melody being constantly restarted, interrupts, deep sleep 10

4 Energy consumption with busy-waiting on timers and polling of buttons. The energy consumption when playing a melody was indistinguishable from the energy consumption when not playing a melody. 11

List of Tables

1 Power consumption of the micro-controller for different situations 9

1 Introduction

This report contains the description of a C program which utilizes hardware timers together with the DAC of a EFM32GG board to generate sounds. Different sounds are played by pressing buttons on a gamepad. A melody is also played when starting the program.

There are two variants of the program, an inefficient implementation using busy-waiting to constantly check the status of the timer's counter. The improved version uses interrupts to monitor and handle GPIO button presses, as well as hardware timer events, where an interrupt is generated when the timer counter reaches the desired value.

A comparison has been made between the two variants, and the discussion is based around measurements of energy consumption. The two solutions are functionally comparable, with both tracking the same timer and GPIO triggers and responding to them. However, the interrupt solution is vastly more energy-efficient and versatile.

2 Detailed tasks

This section will begin by describing the parts of the two programs which are similar, and then proceed by describing the difference in the busy-waiting/interrupt implementations.

The functionality of both programs are the same. On startup, the programs will play part of "The Imperial March" from "Star Wars". When button SW1 is pressed, the song "Lisa Gikk Til Skolen" will be played. When button SW2 is pressed, the startup melody, "The Imperial March", will be played again. When pressing button SW3, part of the "Pirates of the Caribbean" theme song will be played. When button SW8 is pressed, the programs will stop playing the current song.

The EFM32GG contains a DAC which is connected to a amplifier which drives a mini-jack connector. This DAC can therefore be used to generate sound from the board by feeding it a continuous data stream of samples from a sound wave. The Timer peripheral will invoke the alternateDAC-Value function on overflow events, and so we generate frequencies by making sure the timer overflows at twice the desired output frequency. The code controlling the DAC peripheral can be seen in Listing 1.

Melodies are stored in an array of Note structs which provide timing values to timer1 and RTC. Each note struct will contain a desired frequency, and duration. Timer1 controls the half-period of the DAC, resulting in a tone with the given frequency. RTC is given a duration value which determines the duration a given tone is played. After the given duration, the onNoteCleared function in music.c is invoked by an interrupt or polling trigger by the RTC. onNoteCleared then plays the next note in the song, or stops the playing of notes if there are no more notes in the song. See the code for the music in Listing 2, the Timer in Listing 3, and the RTC in Listing 4. Note that the frequencies are stored in the header file pitch.h.

```
// Set up the DAC peripheral
void setupDAC()
{
        // Enable clock for DAC
        *CMU_HFPERCLKEN0 |= (1 << 17);

        // Prescale DAC clock
        *DAC0_CTRL = 0x50010;

        // Enable both channels of the DAC
        *DAC0_CH0CTRL = 1;
        *DAC0_CH1CTRL = 1;
}

// Alternate the sample value in the DAC
```

```
// (We do this every half period to generate the desired frequency)
void alternateDACValue() {
	static volatile uint16_t alternating_bool = 0;

	if (alternating_bool == 0) {
		alternating_bool = 1;
		*DAC0_CH0DATA = AMPLITUDE;
		*DAC0_CH1DATA = AMPLITUDE;
	}
	else {
		alternating_bool = 0;
		*DAC0_CH0DATA = 0;
		*DAC0_CH1DATA = 0;
	}
}
```

Listing 1: dac.c code. function for setting up and using DAC to play sounds.

```
static struct Note currentNoteArray[128];
static uint32_t currentNoteIndex;
static uint32_t currentNoteArraySize;

// Play a note
void playNote(Note n)
{
	// If 0 frequency, stop the timer (play nothing)
	if (n.freq == 0) {
		stopTimer();
	}
	// Else, start timer with desired frequency
	else {
		startTimer(n.freq);
	}
	// Start the RTC timer with the duration of the note
	startRTC(n.dur);
}

// Called when a note is cleared after a RTC interrupt
void onNoteCleared()
{
	// If song not done, play next note
	if (currentNoteIndex < currentNoteArraySize) {
		playNote(currentNoteArray[currentNoteIndex]);
		currentNoteIndex++;
	}
	// Else, stop playing
	else {
		stopTimer();
		clearRTC();
	}
}

// Stop playing music
void stopPlaying()
{
	currentNoteArraySize = 0;
	currentNoteIndex = 0;
	onNoteCleared();
```

```
}
```

Listing 2: music.c code. Code for playing music. The code implementing the songs themselves is quite long, and can be viewed in Appendix A.

```
// Set up the timer peripheral
void setupTimer()
{
        // Prescale clock
        *TIMER1_CTRL |= (0x7 << 24);

        // Enable clock for timer
        *CMU_HFPERCLKEN0 |= (1 << 6);

        // Enable interrupt generation
        *TIMER1_IEN = 1;
}

// Start the timer with the specified frequency
void startTimer(float freq)
{
        // Enable energy mode EM1 (sleep)
        // Because thet timer is not available in EM2
        *SCR = 2;

        // Calculate the needed top
        uint16_t top = freqToTop(freq);

        // Set the top, and start the counter
        *TIMER1_TOP = top;
        *TIMER1_CMD = 0b01;
}

// Stop the timer
void stopTimer()
{
        // Stop the counter
        *TIMER1_CMD = 0b10;

        // Enable energy mode EM2 (deep sleep)
        *SCR = 6;
}

// Convert a frequency to the corresponding TOP register value
uint16_t freqToTop(float freq)
{
        return (uint16_t) (CLOCK_SPEED / (16 * freq));
}
```

Listing 3: timer.c code. The code for setting up and utilising the Timer peripheral.

```
// Set up the RTC peripheral
void setupRTC()
{
    // Enable Low Frequency Clock
    *CMU_HFCORECLKEN0 |= (1 << 4); // Enable LF Clock

    // Set ULFRCO (~1 kHz) as Low Frequency Clock
```

```
    *CMU_LFCLKSEL &= ~(0b11); // Clear LFA
    *CMU_LFCLKSEL |= (1 << 16); // Set LFAE

    // Enable clock for RTC
    *CMU_LFACLKEN0 |= 0b10;

    // Set up RTC
    *RTC_CTRL = 0b100; // Set top = COMP0
    *RTC_IEN = 0b10; // Enable interrupt generation
}

// Start RTC timer with specified number of milliseconds
void startRTC(uint16_t ms)
{
    // Set the top to the specified number of milliseconds
    *RTC_COMP0 = ms;

    // Reset the counter
    *RTC_CNT = 0;

    // Start the counter
    *RTC_CTRL |= 1;
}

// Stop the RTC timer
void clearRTC()
{
    // Stop counting
    *RTC_CTRL |= 0;
}
```

Listing 4: rtc.c code. The code for setting up and utilising the RTC peripheral.

To produce different sounds with the buttons, GPIO pins are first set up as shown in Listing 5.

```
// Set up GPIO peripheral
void setupGPIO()
{
        // Enable GPIO clock
        *CMU_HFPERCLKEN0 |= CMU2_HFPERCLKEN0_GPIO;

        // Set button pins to input
        *GPIO_PC_MODEL = 0x33333333;
        // Enable internal pull-up for buttons
        *GPIO_PC_DOUT = 0xff;

        // Set interrupts for falling edge on buttons
        *GPIO_EXTIPSELL = 0x22222222;
        *GPIO_EXTIFALL = 0xff;
        *GPIO_IEN = 0xff;
}

// Function called when buttons are pressed
void onButtonPress(uint32_t flags)
{
        // SW1 pressed
        if (flags & 0x01) {
                playLisa();
        }
```

```
        // SW2 pressed
        if (flags & 0x02) {
                playImperial();
        }

        // SW3 pressed
        if (flags & 0x04) {
                playPirates();
        }

        // SW8 pressed
        if (flags & 0x80) {
                stopPlaying();
        }
}
```

Listing 5: This listing shows the function for setting up the GPIO pins, and the function that handles button presses.

2.1 Interrupts

The timer interrupts are set up by the setupTimer function shown in Listing 3. This function is almost identical for our polling solution, but the interrupt setup goes unused by the system when polling is used.

The function setupTimer initialises the timer so that it is ready for use, but does not start it. The function startTimer will start the timer and generate an interrupt with frequency $freq$. This will therefore decide the frequency of the generated sound wave. The function stopTimer will stop the timer, meaning we will not produce any sound. Note also that the startTimer function will enable energy mode EM1 (sleep), while stopTimer will enable energy mode EM2 (deep sleep). This is because the timer is not available in mode EM2, but when we are not using the timer we want to use the deep sleep.

RTC is set up to use a slower clock (ca. 1 kHz) which is useful for playing a tone through the DAC and timer1 for a set time duration in milliseconds. The following code shown in Listing 4 initializes and starts the RTC. As with the Timer1 setup, the setup is almost identical but the interrupts are not used by the polling solution.

The timer interrupt handler, GPIO interrupt handler and RTC interrupt handler is shown in Listing 6.

```
// Timer1 interrupt handler
void __attribute__ ((interrupt)) TIMER1_IRQHandler()
{
        *TIMER1_IFC = 1;

        alternateDACValue();
}

// GPIO even pin interrupt handler
void __attribute__ ((interrupt)) GPIO_EVEN_IRQHandler()
{
        uint32_t IF = *GPIO_IF;
        *GPIO_IFC = IF;

        onButtonPress(IF);
```

```
}

// GPIO odd pin interrupt handler
void __attribute__ ((interrupt)) GPIO_ODD_IRQHandler()
{
	uint32_t IF = *GPIO_IF;
	*GPIO_IFC = IF;

	onButtonPress(IF);
}

// RTC interrupt handler
void __attribute__ ((interrupt)) RTC_IRQHandler()
{
	*RTC_IFC = *RTC_IF;

	clearRTC();
	stopTimer();
	onNoteCleared();
}
```

Listing 6: GPIO odd and even, Timer1 and RTC interrupt handlers.

The timer1 interrupt handler alternates between sending the value AMPLITUDE and 0 to both DAC0_CH0DATA and DAC0_CH0DATA, by calling the alternateDACValue function in dac.c. These values are as previously mentioned send with the periodicity given to the startTimer function. This produces a tone which is played until the RTC interrupt handler is triggered.

The GPIO interrupt handlers are split up in even and odd numbered pins. They both hold the same functionality. Both handlers call the onButtonPress function in gpio.c, which will play different melodies depending on which button is pressed, as described earlier.

The RTC interrupt handler runs the onNoteCleared() function which iterates through the note array, resulting in a melody being played.

In the main function, we first invoke the setup function for the peripherals we have described, and enable interrupt handling. We then play the startup melody, before we enable deep sleep and wait for interrupts. This can be seen in Listing 7.

```
// Program start in main
int main(void)
{
	// Set up peripherals
	setupGPIO();
	setupDAC();
	setupTimer();
	setupRTC();

	// Enable interrupt handling
	setupNVIC();

	// Set energy mode EM2 (deep sleep)
	*SCR = 6;

	// Play start-up melody
	playImperial();

	// Wait for interrupt
	__asm__("wfi");
```

```
        return 0;
}

// Enable interrupt handling
void setupNVIC()
{
        *ISER0 |= (1 << 12); // Timer1
        *ISER0 |= 0x802; // GPIO
        *ISER0 |= (1 << 30); // RTC
}
```

Listing 7: The main function for the interrupt solution.

2.2 Polling

Polling simply does the same as the interrupt solution, but the execution of code blocks is handled by a set of conditional statements which achieve interrupt functionality. timers and gpio pins are polled for changes. Whenever a change is detected the current value is in the case of timers compared to the interrupt trigger value, triggering the code block if the value matches. GPIO polling is implemented similarly to the solution used in ex1. Listing 8 is a code except from polling.c displaying the functions handling setup of polling and button, timer1 and RTC polling.

```
static volatile uint32_t buttonPushPrevious;
static volatile uint32_t prevTimer1Value;
static volatile uint32_t prevRTCValue;

void setupPolling()
{
    buttonPushPrevious = *GPIO_PC_DIN & 0xff;
    prevTimer1Value = *TIMER1_CNT;
    prevRTCValue = *RTC_CNT;
}

void pollButtons()
{
    uint32_t buttonPushCurrent = *GPIO_PC_DIN & 0xff;
    uint32_t buttonChange = buttonPushCurrent ^ buttonPushPrevious;
    uint32_t buttonFallingEdge = buttonChange & buttonPushPrevious;

    onButtonPress(buttonFallingEdge);

    buttonPushPrevious = buttonPushCurrent;
}

void pollTimer1()
{
    uint32_t currentTimer1Value = *TIMER1_CNT;

    if (prevTimer1Value > currentTimer1Value) {
        alternateDACValue();
    }

    prevTimer1Value = currentTimer1Value;
}

void pollRTC()
{
```

```
    uint32_t currentRTCValue = *RTC_CNT;

    if (currentRTCValue != prevRTCValue && currentRTCValue == *RTC_COMP0) {
        clearRTC();
        stopTimer();
        onNoteCleared();
    }

    prevRTCValue = currentRTCValue;
}
```

Listing 8: Polling handlers. Functionally equal to the interrupt implementation, but less energy efficient and less versatile.

Note that these functions call the same functions as the interrupt handlers did. In the main function of the polling solution, we setup the peripherals, play the opening melody, and then start the polling process.

```
// Program start in main
int main(void)
{
        // Set up peripherals
        setupGPIO();
        setupDAC();
        setupTimer();
        setupRTC();

        // Set up the polling process
        setupPolling();

        // Play start-up melody
        playImperial();

        // Start polling
        while (1) {
                pollButtons();
                pollTimer1();
                pollRTC();
        }

        return 0;
}
```

Listing 9: The main function for the polling solution.

3 Energy consumption

For the solution using interrupts we needed access to the timer interrupt for playing sound so we could only make use of the basic sleep mode while music was being played. After a song was played the controller would enter deep sleep until a button was pressed. Then it would play a new song and go into basic sleep mode for the duration.

The values in different situations are summarized in table 1

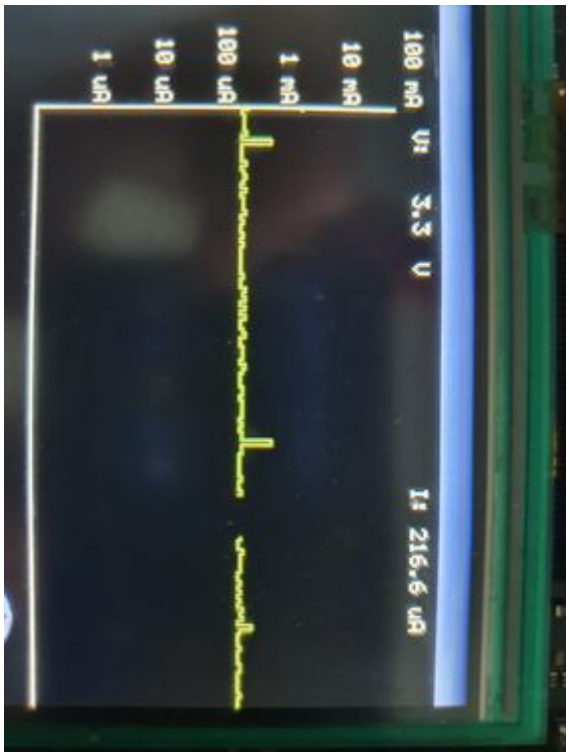

Figure 1: Interrupts idle, deep sleep

Situation	**Power consumption**
Polling, playing/not playing sound	4.9mA
Interrupts, idle	216.6μA
Interrupts, playing melody	1.8mA
Interrupts, high frequency button pressing,	1.9mA

Table 1: Power consumption of the micro-controller for different situations

For the polling solution we had the same power consumption no matter the input or output of the controller, which had a steady current consummation of 4.9mA.

In figure 1 we see the controller idling in deep sleep. Here it uses about 216.6uA. As seen in figure 2 we had an average current usage of 1.8mA during basic sleep mode while the controller is playing a melody. In figure 3 we see the buttons for restarting the melody being constantly mashed we get about the same result as while playing a melody with small spikes up to 1.9mA.

In figure 4 we see the energy consumption for the polling solution. The energy consumption was the same whether a song was playing or not, and whether buttons were being pressed or not. It was constantly at 4.9mA.

4 Discussion

We can see that in power consumption the the interrupt solution is clearly superior. Under interrupt idling we achieved with deep sleep a current usage of over one magnitude lower than with busy wait polling. With standard sleep mode the interrupt solution still achieved a power usage 3.0mA lower than than the polling which is approximately $\frac{2}{3}$ times lower. Rapid button pressing gave no significant spike in current usage for any of the solutions an was approximate to the 1.8mA current usage under playing of sound.

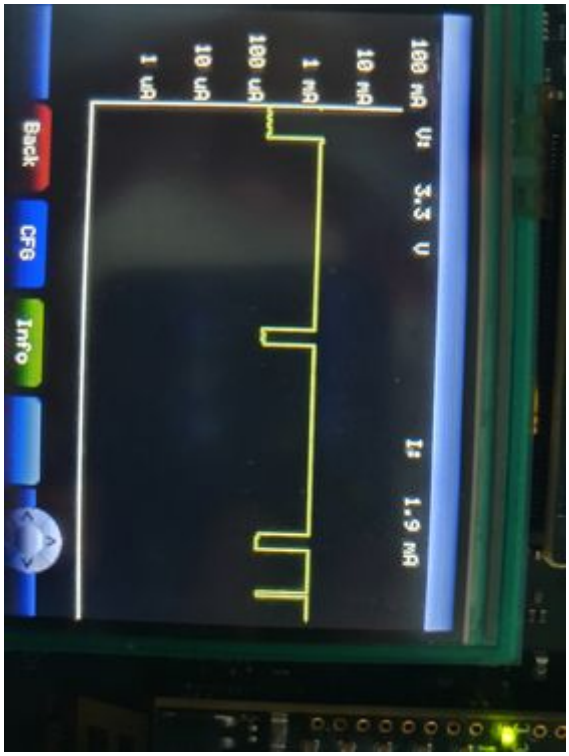

Figure 2: Melody, with timer interrupts

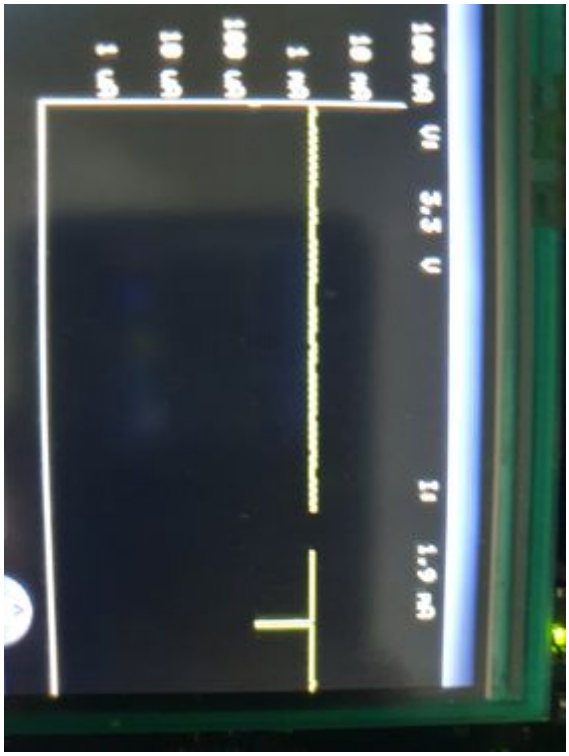

Figure 3: Melody being constantly restarted, interrupts, deep sleep

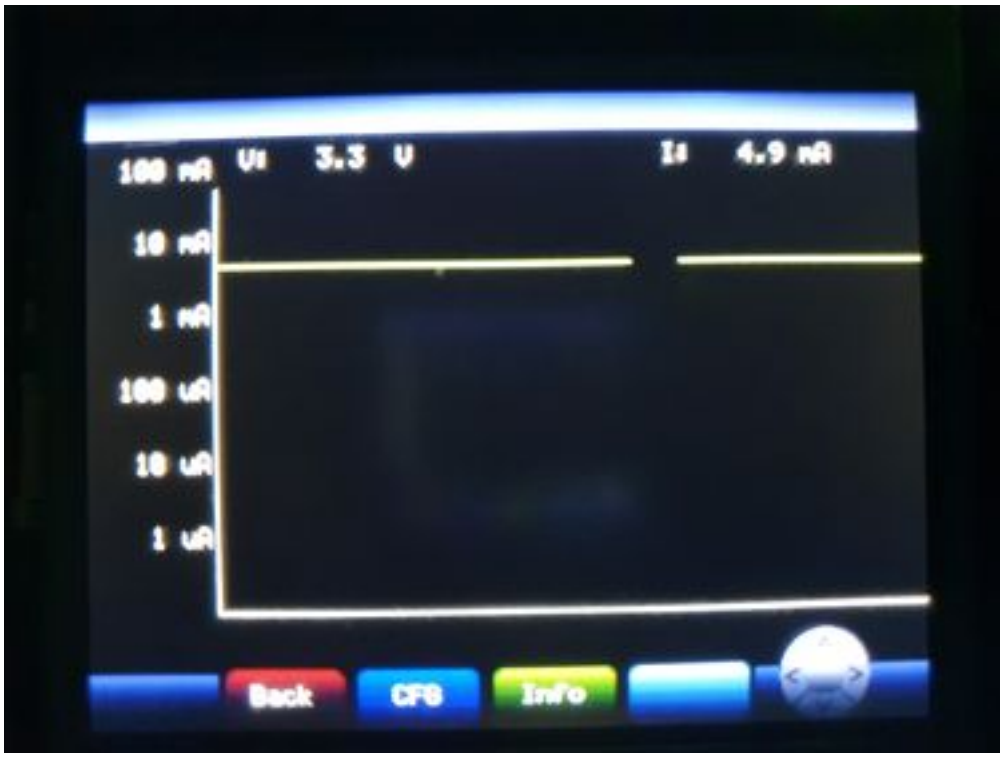

Figure 4: Energy consumption with busy-waiting on timers and polling of buttons. The energy consumption when playing a melody was indistinguishable from the energy consumption when not playing a melody.

We also had a much easier time implementing the interrupt solution as the code was easier to differentiate and structure. It also made program flow easier since you can easily jump in and out of functions with the interrupts.

5 Conclusion

In this report, we implemented two different ways for playing melodies and sound effects on start up and by pressing the buttons on the game-pad. One solution used polling while the other used interrupts and methods for energy efficiency.

For this project the interrupt implementation was both easier to implement and more energy efficient. The controller used over one magnitude of current more in idle mode with poling, and about $\frac{2}{3}$ more when actively playing sound. The interrupt solution also made planing and implementing the programflow easier.

Bibliography

Appendix

A Appendix A

Code implementing the different songs:

```
// Play Lisa gikk til skolen
void playLisa()
{
        struct Note lisa[] = {
                (struct Note) { .freq = C4, .dur = 500 },
                (struct Note) { .freq = D4, .dur = 500 },
                (struct Note) { .freq = E4, .dur = 500 },
                (struct Note) { .freq = F4, .dur = 500 },
                (struct Note) { .freq = G4, .dur = 1000 },
                (struct Note) { .freq = 0, .dur = 10 },
                (struct Note) { .freq = G4, .dur = 990 },
                (struct Note) { .freq = A4, .dur = 500 },
                (struct Note) { .freq = 0, .dur = 10 },
                (struct Note) { .freq = A4, .dur = 490 },
                (struct Note) { .freq = 0, .dur = 10 },
                (struct Note) { .freq = A4, .dur = 490 },
                (struct Note) { .freq = 0, .dur = 10 },
                (struct Note) { .freq = A4, .dur = 490 },
                (struct Note) { .freq = G4, .dur = 1000 },
                (struct Note) { .freq = F4, .dur = 500 },
                (struct Note) { .freq = 0, .dur = 10 },
                (struct Note) { .freq = F4, .dur = 490 },
                (struct Note) { .freq = 0, .dur = 10 },
                (struct Note) { .freq = F4, .dur = 490 },
                (struct Note) { .freq = 0, .dur = 10 },
                (struct Note) { .freq = F4, .dur = 490 },
                (struct Note) { .freq = E4, .dur = 1000 },
                (struct Note) { .freq = 0, .dur = 10 },
                (struct Note) { .freq = E4, .dur = 990 },
                (struct Note) { .freq = D4, .dur = 500 },
                (struct Note) { .freq = 0, .dur = 10 },
                (struct Note) { .freq = D4, .dur = 490 },
                (struct Note) { .freq = 0, .dur = 10 },
                (struct Note) { .freq = D4, .dur = 490 },
                (struct Note) { .freq = 0, .dur = 10 },
                (struct Note) { .freq = D4, .dur = 490 },
                (struct Note) { .freq = C4, .dur = 500},
                (struct Note) { .freq = 0, .dur = 10}
        };

        for (uint32_t i = 0; i < sizeof(lisa) && i < sizeof(currentNoteArray); i++) {
                currentNoteArray[i] = lisa[i];
        }
        currentNoteArraySize = sizeof(lisa) / sizeof(lisa[0]);
        currentNoteIndex = 0;

        onNoteCleared();
}

// Play the imperial march
void playImperial()
```

```
{
        struct Note imperial[] = {
                (struct Note) { .freq = A3, .dur = 750 },
                (struct Note) { .freq = 0, .dur = 20 },
                (struct Note) { .freq = A3, .dur = 730 },
                (struct Note) { .freq = 0, .dur = 20 },
                (struct Note) { .freq = A3, .dur = 730 },
                (struct Note) { .freq = F3, .dur = 500 },
                (struct Note) { .freq = C4, .dur = 250 },
                (struct Note) { .freq = A3, .dur = 750 },
                (struct Note) { .freq = F3, .dur = 500 },
                (struct Note) { .freq = C4, .dur = 250 },
                (struct Note) { .freq = A3, .dur = 1000 },
                (struct Note) { .freq = 0, .dur = 500 },

                (struct Note) { .freq = E4, .dur = 750 },
                (struct Note) { .freq = 0, .dur = 20 },
                (struct Note) { .freq = E4, .dur = 730 },
                (struct Note) { .freq = 0, .dur = 20 },
                (struct Note) { .freq = E4, .dur = 730 },
                (struct Note) { .freq = F4, .dur = 500 },
                (struct Note) { .freq = C4, .dur = 250 },
                (struct Note) { .freq = G3S, .dur = 750 },
                (struct Note) { .freq = F3, .dur = 500 },
                (struct Note) { .freq = C4, .dur = 250 },
                (struct Note) { .freq = A3, .dur = 1000 },
                (struct Note) { .freq = 0, .dur = 500 },

                (struct Note) { .freq = A4, .dur = 750 },
                (struct Note) { .freq = A3, .dur = 500 },
                (struct Note) { .freq = 0, .dur = 20 },
                (struct Note) { .freq = A3, .dur = 230 },
                (struct Note) { .freq = A4, .dur = 750 },
                (struct Note) { .freq = G4S, .dur = 500 },
                (struct Note) { .freq = G4, .dur = 250 },
                (struct Note) { .freq = F4S, .dur = 125 },
                (struct Note) { .freq = F4, .dur = 250 },
                (struct Note) { .freq = F4S, .dur = 250 }

        };

        for (uint32_t i = 0; i < sizeof(imperial) && i < sizeof(currentNoteArray); i++) {
                currentNoteArray[i] = imperial[i];
        }
        currentNoteArraySize = sizeof(imperial) / sizeof(imperial[0]);
        currentNoteIndex = 0;

        onNoteCleared();

}

// Play the Pirates of the Caribbean theme
void playPirates()
{
        struct Note pirates[] = {
                (struct Note) { .freq = A3, .dur = 250 },
                (struct Note) { .freq = C4, .dur = 250 },
                (struct Note) { .freq = D4, .dur = 490 },
```

```
(struct Note) { .freq = 0, .dur = 10 },
(struct Note) { .freq = D4, .dur = 490 },
(struct Note) { .freq = 0, .dur = 10 },
(struct Note) { .freq = D4, .dur = 250 },
(struct Note) { .freq = E4, .dur = 250 },
(struct Note) { .freq = F4, .dur = 490 },
(struct Note) { .freq = 0, .dur = 10 },
(struct Note) { .freq = F4, .dur = 490 },
(struct Note) { .freq = 0, .dur = 10 },
(struct Note) { .freq = F4, .dur = 250 },
(struct Note) { .freq = G4, .dur = 250 },
(struct Note) { .freq = E4, .dur = 500 },
(struct Note) { .freq = 0, .dur = 10 },
(struct Note) { .freq = E4, .dur = 500 },
(struct Note) { .freq = D4, .dur = 250 },
(struct Note) { .freq = C4, .dur = 240 },
(struct Note) { .freq = 0, .dur = 10 },
(struct Note) { .freq = C4, .dur = 250 },
(struct Note) { .freq = D4, .dur = 500 },
(struct Note) { .freq = 0, .dur = 250 },

(struct Note) { .freq = A3, .dur = 250 },
(struct Note) { .freq = C4, .dur = 250 },
(struct Note) { .freq = D4, .dur = 490 },
(struct Note) { .freq = 0, .dur = 10 },
(struct Note) { .freq = D4, .dur = 490 },
(struct Note) { .freq = 0, .dur = 10 },
(struct Note) { .freq = D4, .dur = 250 },
(struct Note) { .freq = E4, .dur = 250 },
(struct Note) { .freq = F4, .dur = 490 },
(struct Note) { .freq = 0, .dur = 10 },
(struct Note) { .freq = F4, .dur = 490 },
(struct Note) { .freq = 0, .dur = 10 },
(struct Note) { .freq = F4, .dur = 250 },
(struct Note) { .freq = G4, .dur = 250 },
(struct Note) { .freq = E4, .dur = 500 },
(struct Note) { .freq = 0, .dur = 10 },
(struct Note) { .freq = E4, .dur = 500 },
(struct Note) { .freq = D4, .dur = 250 },
(struct Note) { .freq = C4, .dur = 240 },
(struct Note) { .freq = D4, .dur = 500 },
(struct Note) { .freq = 0, .dur = 500 },

(struct Note) { .freq = A3, .dur = 250 },
(struct Note) { .freq = C4, .dur = 250 },
(struct Note) { .freq = D4, .dur = 490 },
(struct Note) { .freq = 0, .dur = 10 },
(struct Note) { .freq = D4, .dur = 490 },
(struct Note) { .freq = 0, .dur = 10 },
(struct Note) { .freq = D4, .dur = 250 },
(struct Note) { .freq = F4, .dur = 250 },
(struct Note) { .freq = G4, .dur = 490 },
(struct Note) { .freq = 0, .dur = 10 },
(struct Note) { .freq = G4, .dur = 490 },
(struct Note) { .freq = 0, .dur = 10 },
(struct Note) { .freq = G4, .dur = 250 },
(struct Note) { .freq = A4, .dur = 250 },
(struct Note) { .freq = A4S, .dur = 490 },
```

```
                (struct Note) { .freq = 0, .dur = 10 },
                (struct Note) { .freq = A4S, .dur = 500 },
                (struct Note) { .freq = A4, .dur = 250 },
                (struct Note) { .freq = G4, .dur = 250 },
                (struct Note) { .freq = A4, .dur = 250 },
                (struct Note) { .freq = D4, .dur = 490 },
                (struct Note) { .freq = 0, .dur = 250 },

                (struct Note) { .freq = D4, .dur = 250 },
                (struct Note) { .freq = E4, .dur = 250 },
                (struct Note) { .freq = F4, .dur = 490 },
                (struct Note) { .freq = 0, .dur = 10 },
                (struct Note) { .freq = F4, .dur = 500 },
                (struct Note) { .freq = G4, .dur = 500 },
                (struct Note) { .freq = A4, .dur = 250 },
                (struct Note) { .freq = D4, .dur = 490 },
                (struct Note) { .freq = 0, .dur = 250 },

                (struct Note) { .freq = D4, .dur = 250 },
                (struct Note) { .freq = F4, .dur = 250 },
                (struct Note) { .freq = E4, .dur = 490 },
                (struct Note) { .freq = 0, .dur = 10 },
                (struct Note) { .freq = E4, .dur = 500 },
                (struct Note) { .freq = F4, .dur = 250 },
                (struct Note) { .freq = E4, .dur = 250 },
                (struct Note) { .freq = D4, .dur = 500 }

        };

        for (uint32_t i = 0; i < sizeof(pirates) && i < sizeof(currentNoteArray); i++) {
                currentNoteArray[i] = pirates[i];
        }
        currentNoteArraySize = sizeof(pirates) / sizeof(pirates[0]);
        currentNoteIndex = 0;

        onNoteCleared();
}
```