NTNU
Kunnskap for en bedre verden

Department of Computer Science

TDT4258 - Low-level Programming

Assignment 2

Group Number: 2
Board Number: EFM17/1

Group Members:
Maren Vorin Fossum
Kristine Døsvik
Christian Nguyen
Linn Marie Sønsterud

April, 2020

Table of Contents

List of Figures

List of Tables

1 Overview

In this project we have created two solutions that play different kind of melodies when certain buttons are pressed. The solutions also let you choose which kind of waveform to use when playing the melodies. The waveform options are sinus, square, triangle and sawtooth.

In the first solution we use busy wait and polling. Polling by periodically checking the status of the buttons, and busy wait by always reacting on the status of the buttons. In average this have a energy consumption between 5.45 mA and 8.71 when songs are played in the specified wave formats. While a song is played a given amount of LEDs are on to symbolize which wave format that the song is played in. The idle states in this solution ligths the LEDs to show which wave format the microcontroller currently is set to use when playing a song. The energy in this idle state will therefor variate between an avarage of 6.30 mA to 9.32 mA.

The second solution uses interrupt to detect a button press and operate accordingly. This solution has been implemented with sleep mode. This solution turns off all LEDs when going into low energy mode. Therefore it differs form the busy wait polling solution because it only has one idle state. In average the idle state have an energy consumption of 1.25 uA. When playing a song LEDs are lit according to which wave format the the song is played in. The average energy consumption for playing a song variates between 5.32 mA and 8.73 mA.

2 Detailed Tasks

In this course we use the EFM32GG-DK3750 development board from Silicon Labs/Energy Micro to implement a small computer game. In this project we add the feature of sound generation. This is done by using a ribbon cable to connect the development board to a game pad with buttons and LEDs. The user controls the sound by pressing the buttons on the game pad. To implement this we use the programming language C to enable and configure the relevant registers, and make sounds by continuously write samples to a data register in the DAC. The two solutions use a general setup to initialize the correct registers for GPIO (LEDs and buttons), Timer and DAC. The solutions use the same sound and song functionality, but while the busy wait polling solution always shows what wave format the microcontroller currently have by ligthing up certain LEDs, the interrupt solution turns off all LEDs when going into low power mode.

2.1 General Setup

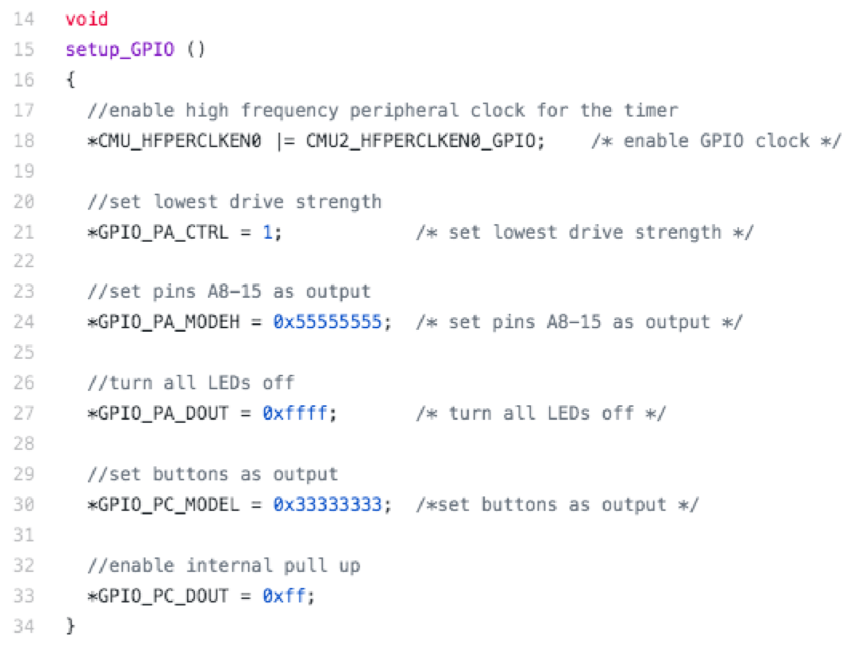
To be able to make sound on button press we need to set some of the microcontrollers registers concerning, GPIO, digital to analog converter (DAC), and timer in a certain way. The GPIO register should make sure a button push is registered and light some of the LEDs for symbolizing what happens. The DAC registers should convert the digital values into analog sounds that can be heard, and the timer should be able to control the period between the samples sendt to the DAC.

2.1.1 GPIO Setup

To set up the GPIO we first need to clock the GPIO pins. We use the internal high frequency clock in the microcontroller for this. To enable this clock we set bit 13 in register CMU_HFPERCLKEN0 high. Then we need to configure the output pins. The reason why we do this is because we want to use LEDs connected to these output pins to symbolize when something happens. The last eight GPIO pins in port A are used as output pins, these are connected to the LEDs in the gameboard. They are set as push-pull output with adjustable drive strength by writing 0x55555555 to register GPIO_PA_MODEH. As we want to use minimal power for lightning the LEDs we set drive strength of the output pins low. This is done by writing 1 to register GPIO_PA_CTRL. In addition we want the initial value of the output pins to correspond to the LEDs being off. We do this by writing 0xffff to register GPIO_PA_DOUT.

To be able to use the buttons on the gameboard, we configure the input pins. The input pins connected to the buttons are the first eight GPIO pins in port C. We set the mode of these pins to input pins with filter and adjustable pull direction. This is done by writing 0x33333333 to register GPIO_PC_MODEL. We also set pull direction to the input pins to internal pull up. This is done by writing 0xff to register GPIO_PC_DOUT. When we write 0x$Y_1Y_2Y_3Y_4$ it is only the value of Y_1 and Y_2 that affects how the LEDs are set. When we give values to Y_3 and Y_4, we write to GPIO pins that are not used in this project, and will therefor not effect the system.

Figure 1 shows the setup of the GPIO pins.



```c
void
setup_GPIO ()
{
  //enable high frequency peripheral clock for the timer
  *CMU_HFPERCLKEN0 |= CMU2_HFPERCLKEN0_GPIO;    /* enable GPIO clock */

  //set lowest drive strength
  *GPIO_PA_CTRL = 1;            /* set lowest drive strength */

  //set pins A8-15 as output
  *GPIO_PA_MODEH = 0x55555555;  /* set pins A8-15 as output */

  //turn all LEDs off
  *GPIO_PA_DOUT = 0xffff;       /* turn all LEDs off */

  //set buttons as output
  *GPIO_PC_MODEL = 0x33333333;  /*set buttons as output */

  //enable internal pull up
  *GPIO_PC_DOUT = 0xff;
}
```

Figure 1: Code for setting up the GPIO registers.

2.1.2 DAC Setup

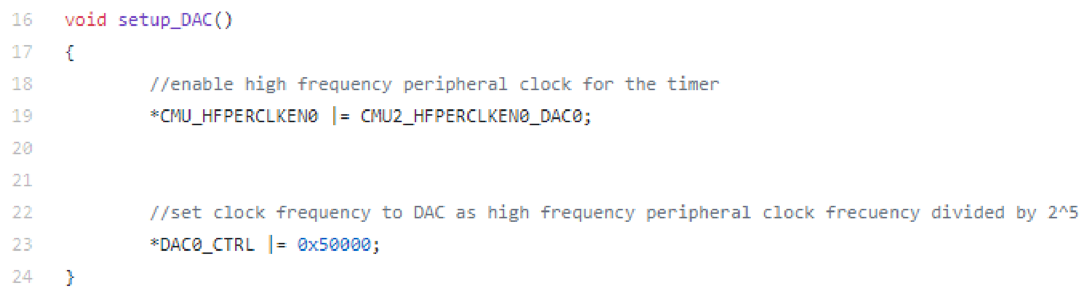
The DAC (Digital to Analog Converter) converts digital values into an analog signal which might generate sound if fed with digital values in a certain way. The reason for this is because on the development board the DAC is connected to the amplifier which drives the Audio Out connector.

When using the DAC we differ from setting up the DAC generally, and turning off and on the DAC depending on if it is used or not.

To set up the DAC we need to clock the timer. We use the internal high frequency clock in the microcontroller for this too. We enable the clock by setting bit 17 in register CMU_HFPERCLKEN0 high. We also need to set the frequency of the DAC timer to 32 times lower than what the internal high frequency clock is. This is done because we want to count in a lower frequency than the internal high frequency clock. We do this by setting bit 16 and 18 in register DAC0_CTRL high. This means that PRESC will have the value 5, which gives us the frequency 437.5KHz of the DAC clock which is found by using Equation 1.

$$f_{DAC\_CLK} = \frac{f_{HFPERCLK}}{2^{PRESC}} = \frac{14MHz}{2^5} = 437.5KHz \tag{1}$$

Figure 2 shows how this is implemented.

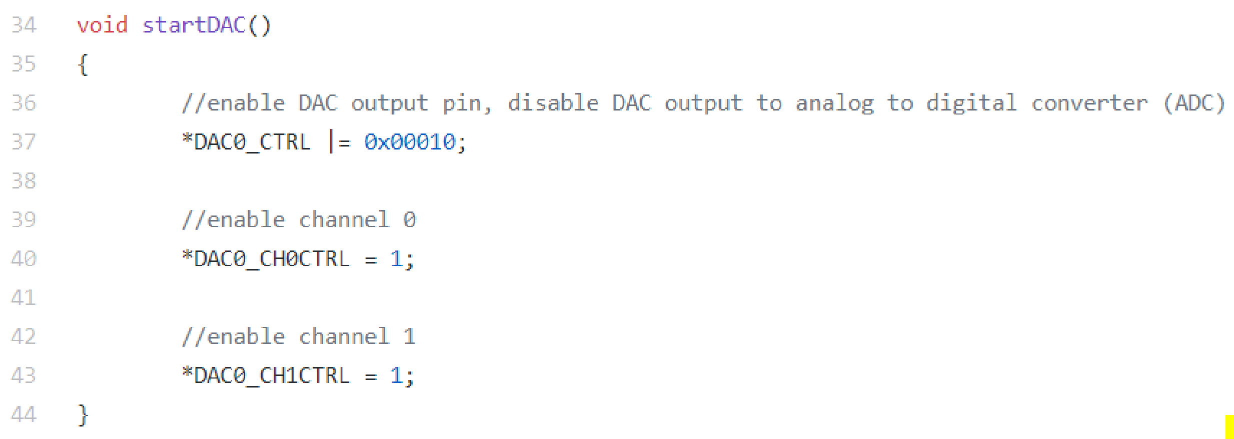


```
void setup_DAC()
{
        //enable high frequency peripheral clock for the timer
        *CMU_HFPERCLKEN0 |= CMU2_HFPERCLKEN0_DAC0;


        //set clock frequency to DAC as high frequency peripheral clock frecuency divided by 2^5
        *DAC0_CTRL |= 0x50000;
}
```

Figure 2: Code for setting up the DAC.

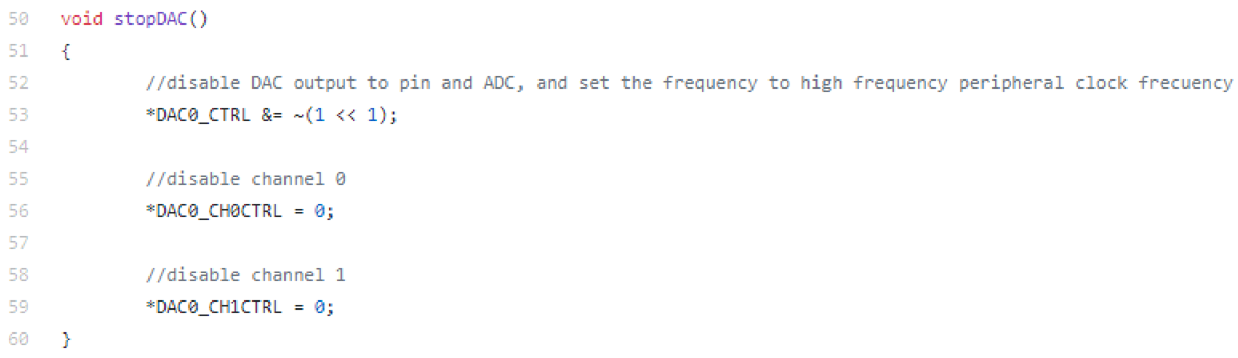
To start and stop the DAC we also need to configure some registers. The description of which registers that must configures and how is given below, while the implementation of respectively starting the DAC and stopping the DAC is shown in figure 3, and 4

The DAC output pins must either be enabled or disabled depending on if the DAC will be turned on or off. To enable the pins we set bit 4 high. To disable the pins we set bit 4 low. The channels used to transfer sound must either be enabled or disabled according to if the DAC will be turned on or off. To enable the channels we write 1 to both the registers DAC0_CH0CTRL and DAC_CH1CTRL. To disable the channels we write 0 to the same registers.



```
void startDAC()
{
        //enable DAC output pin, disable DAC output to analog to digital converter (ADC)
        *DAC0_CTRL |= 0x00010;

        //enable channel 0
        *DAC0_CH0CTRL = 1;

        //enable channel 1
        *DAC0_CH1CTRL = 1;
}
```

Figure 3: Code for starting the DAC.



```
void stopDAC()
{
        //disable DAC output to pin and ADC, and set the frequency to high frequency peripheral clock frecuency
        *DAC0_CTRL &= ~(1 << 1);

        //disable channel 0
        *DAC0_CH0CTRL = 0;

        //disable channel 1
        *DAC0_CH1CTRL = 0;
}
```

Figure 4: Code for disabling the DAC.

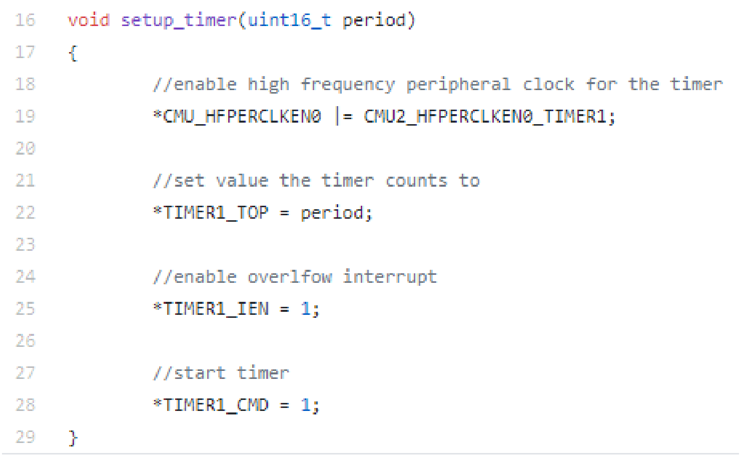
2.1.3 Timer Setup

Since our program needs to run without an operating system we need to enable a hardware timer to generate the sound within a limited period.

The timer is used to to set the period between when the DAC reads the samples sent to the data registers when sound is generated. The timer works as a counter and count up each clock period to a specific value. The timer can be configured to count within the range of 0 to $2^{16} - 1$, When the timer has counted to determined value it will set an interrupt flag, reset the register value to zero, and start a new count. Every time an interrupt flag is generated an interrupt handler will clear the interrupt flag.

To control the timer and the processor we need to set up the high frequency clock. We enable this clock by setting bit 6 in register CMU_HFPERCLKEN0 high. Then we set the value the timer will count to. This is done by passing the desired value to register TIMER1_TOP. To know when the counter has finished we need to enable overflow interrupt. Writing 1 to the register TIMER1_IEN will ensure this. When the timer is successfully configured we start it by writing 1 to register TIMER1_CMD.

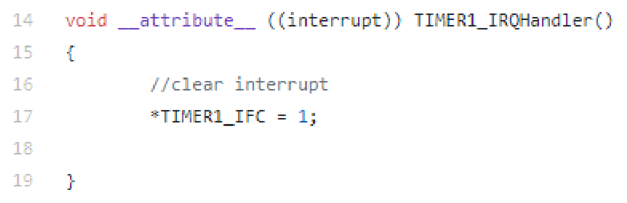
Figure 5 shows how this is implemented.



```
void setup_timer(uint16_t period)
{
        //enable high frequency peripheral clock for the timer
        *CMU_HFPERCLKEN0 |= CMU2_HFPERCLKEN0_TIMER1;

        //set value the timer counts to
        *TIMER1_TOP = period;

        //enable overlfow interrupt
        *TIMER1_IEN = 1;

        //start timer
        *TIMER1_CMD = 1;
}
```

Figure 5: Code for setting up the timer.

In addition to setting up the timer, we need to define what happens when an interrupt from the timer has occurred. The interrupt handler clears the interrupt flag. Figure 6 shows the implementation for the handler.



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

}
```

Figure 6: Code for the interrupt handler for the timer.

2.2 Sound functionality

The microcontrollers overall functionality is that it plays different songs depending on which button is pressed, and it can change the waveform it uses to play the songs. By waveform we mean periodical sound waves in shape of squares, triangles, sathooths or sinuses.

By setting up the DAC and timer as described above the microcontroller are now able to generate sound, which can further be used to generate melodies. By setting up the GPIO pins we can use buttons on the game console to decide which song to play,and the LEDs to symbolize which wave format the song is played in. The following subsections will describe respectively the sound generation for the microcontroller, the song generation, and how buttons are used to shift waveform and choose song.

2.2.1 Sound Generation

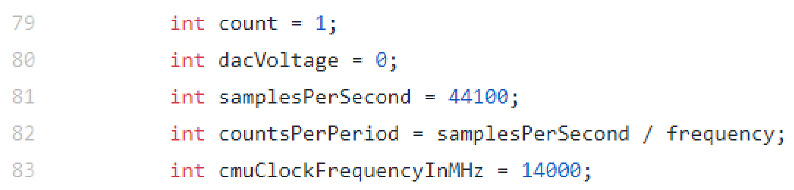
To make sound we write a value to the DAC_CHXDATA register. The DAC will read the value at the data register at specific times and convert this value to a voltage and send it out trough the speaker. When channel 0 is written to you get sound on the left earplug, and channel 1 writes to the right earplug. We want equal sound on both sides, so we always write the same value to both channels.

For the sound to propagate it need to be a type of wave, meaning the voltage need to variate. We generate the waveforms using discrete samples at regular intervals. We simulate the waveforms by using the high frequency peripheral clock to count a variable upwards. When the count reaches a certain length we change the sample volt sent to the DAC so the voltage change will generates sounds. For clear notes the sound will change periodically, meaning we can update the voltage at regular intervals. The period of the sound can be found by taking the clock frequency divided on the desired frequency.

2.3 Generations of waves

The rate the voltage variable should update depend on the waveform. For a square wave we only need to update the value every half period. For the rest we need to continuously update the values inside the period. A triangle wave and sinus wave have a growing voltage for half of the period, and a decreasing voltage for the rest. A sawtooth wave have constant increasing value for the entire period.

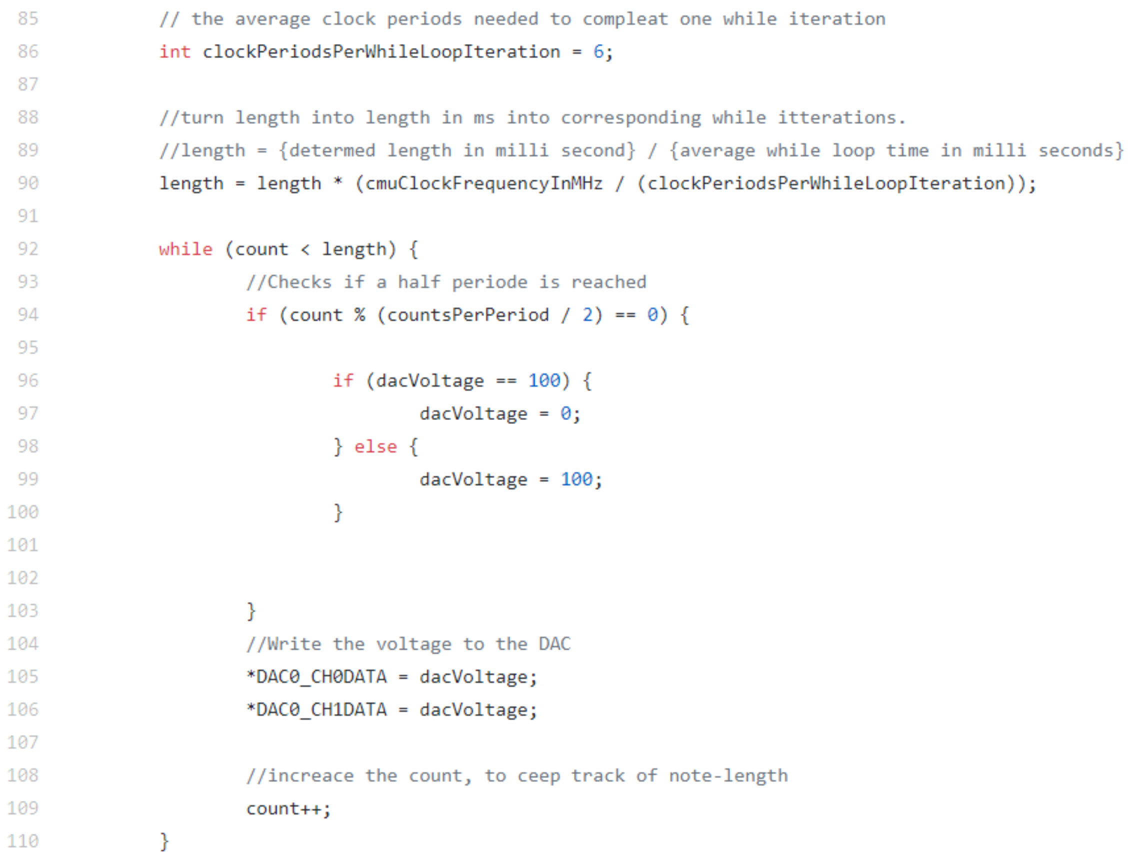
All the waves uses a while loop to update a counter until the counter reach the specified length of the tone. Inside the while loop we generate the desired waveform with a period corresponding to the frequency we send in. This period need to be related to the counting so we calculate the count period by taking the number of samples per second times the period of our desired frequency. To determine the length of the note we need to know how long it takes on average to go through one while iteration. This is done by multiplying the needed clock periods times the CMU clock period. This is done in each wave generating function and you can see the code of the common implementation of constants in Figure 7.



```
        int count = 1;
        int dacVoltage = 0;
        int samplesPerSecond = 44100;
        int countsPerPeriod = samplesPerSecond / frequency;
        int cmuClockFrequencyInMHz = 14000;
```

Figure 7: Code for generating wave parameters

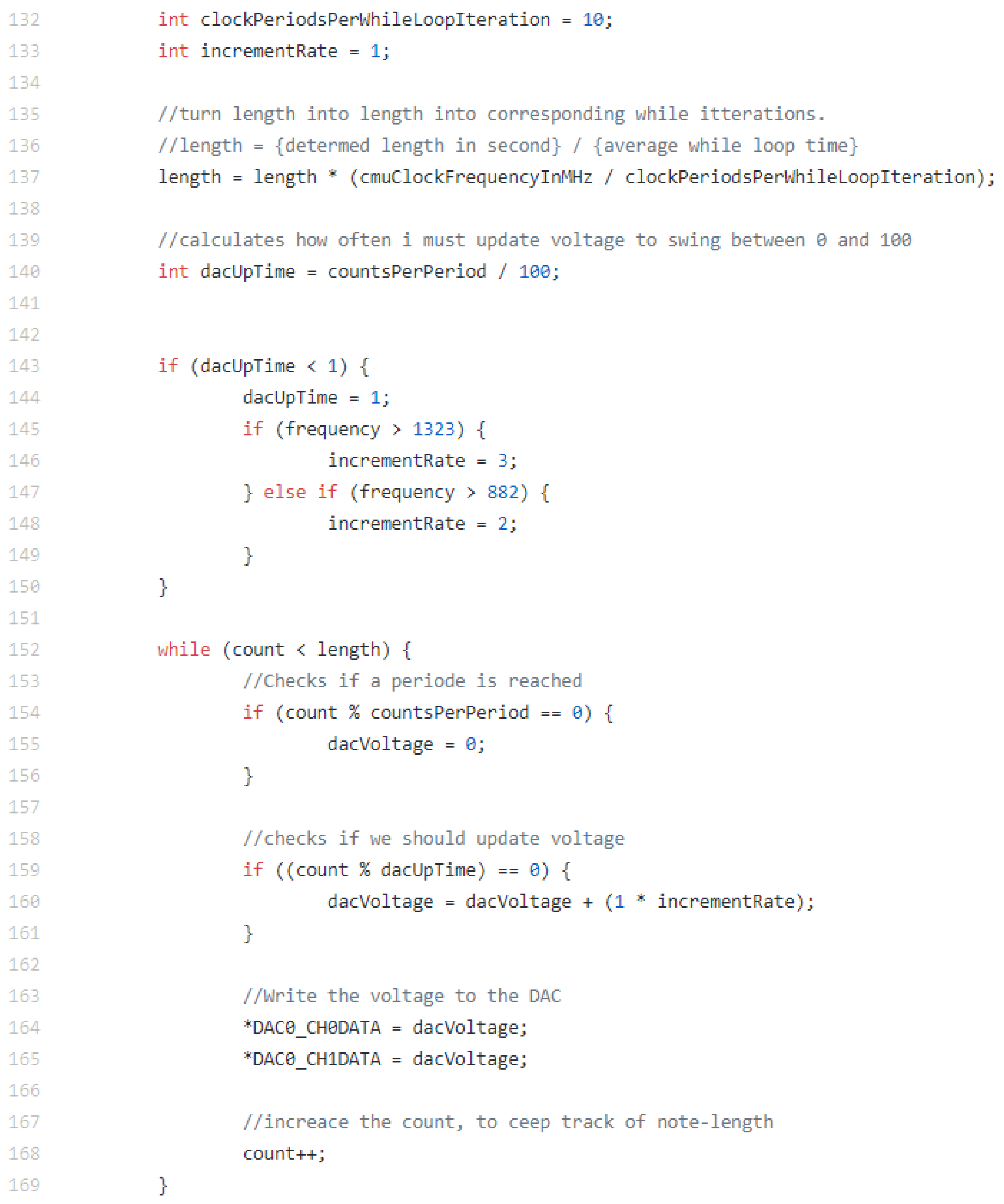
For a square wave we use an average of seven clock periods per while iterations. Since we send in the chosen length in ms, we also define the while loop time in ms. We therefore calculate that the average while iteration takes $7/14000000s = 7/14000ms$. To find the count length we then divide our desired length of note by the average while iteration time. A square wave only need to update the voltage value for each half period as showed in Figure 8.



```c
// the average clock periods needed to compleat one while iteration
int clockPeriodsPerWhileLoopIteration = 6;

//turn length into length in ms into corresponding while itterations.
//length = {determed length in milli second} / {average while loop time in milli seconds}
length = length * (cmuClockFrequencyInMHz / (clockPeriodsPerWhileLoopIteration));

while (count < length) {
        //Checks if a half periode is reached
        if (count % (countsPerPeriod / 2) == 0) {

                if (dacVoltage == 100) {
                        dacVoltage = 0;
                } else {
                        dacVoltage = 100;
                }


        }
        //Write the voltage to the DAC
        *DAC0_CH0DATA = dacVoltage;
        *DAC0_CH1DATA = dacVoltage;

        //increace the count, to ceep track of note-length
        count++;
}
```

Figure 8: Code for generating square wave.

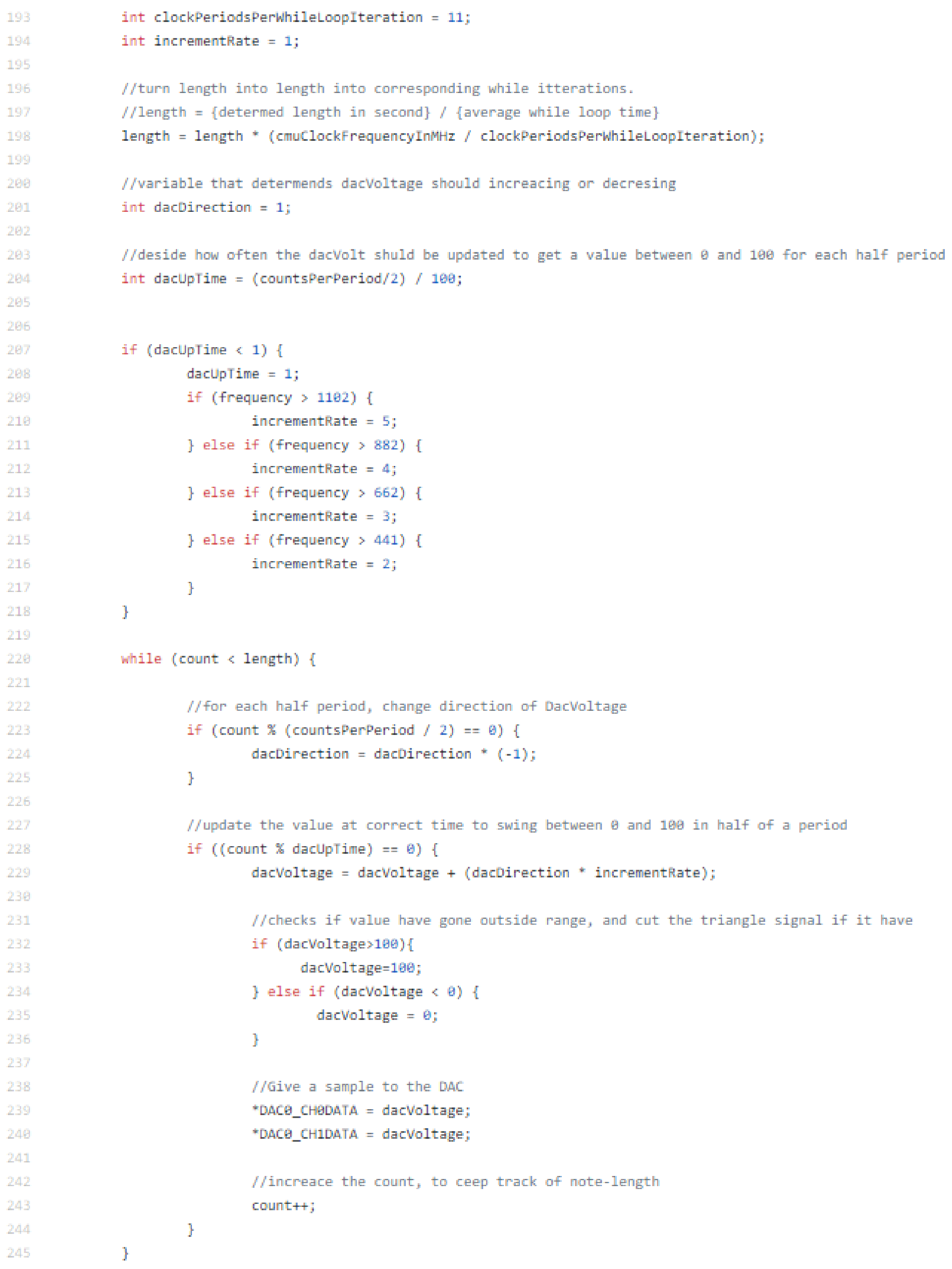
Similarly to the square wave we need to calculate the count length that correspond to our desired length. We find that the average clock periods for the while loop is 10 and calculate the length variable. In addition to determining the period and length of the tone, sawtooth wave also need to determine the update rate of the voltage. This rate is dependant on the period since we want the wave to swing between 0 and 100V during one period. To find the volt-update time value we simply take the count period divided by hundred and update the dacvoltage with one each time count reaches an multiply of the volt-update value. If the count period is less than 100, we update the voltage each while iterations, but the increase varies. if the frequency is larger than 882Hz, the count period is less than 50 so the increase each period is two per iteration. The rate is three times faster if we use a frequency over 1323Hz. The code for implementing the sawtooth wave is shown in Figure 9.



```
    int clockPeriodsPerWhileLoopIteration = 10;
    int incrementRate = 1;

    //turn length into length into corresponding while itterations.
    //length = {determed length in second} / {average while loop time}
    length = length * (cmuClockFrequencyInMHz / clockPeriodsPerWhileLoopIteration);

    //calculates how often i must update voltage to swing between 0 and 100
    int dacUpTime = countsPerPeriod / 100;


    if (dacUpTime < 1) {
        dacUpTime = 1;
        if (frequency > 1323) {
            incrementRate = 3;
        } else if (frequency > 882) {
            incrementRate = 2;
        }
    }

    while (count < length) {
        //Checks if a periode is reached
        if (count % countsPerPeriod == 0) {
            dacVoltage = 0;
        }

        //checks if we should update voltage
        if ((count % dacUpTime) == 0) {
            dacVoltage = dacVoltage + (1 * incrementRate);
        }

        //Write the voltage to the DAC
        *DAC0_CH0DATA = dacVoltage;
        *DAC0_CH1DATA = dacVoltage;

        //increace the count, to ceep track of note-length
        count++;
    }
```

Figure 9: Code for generating sawtooth wave.

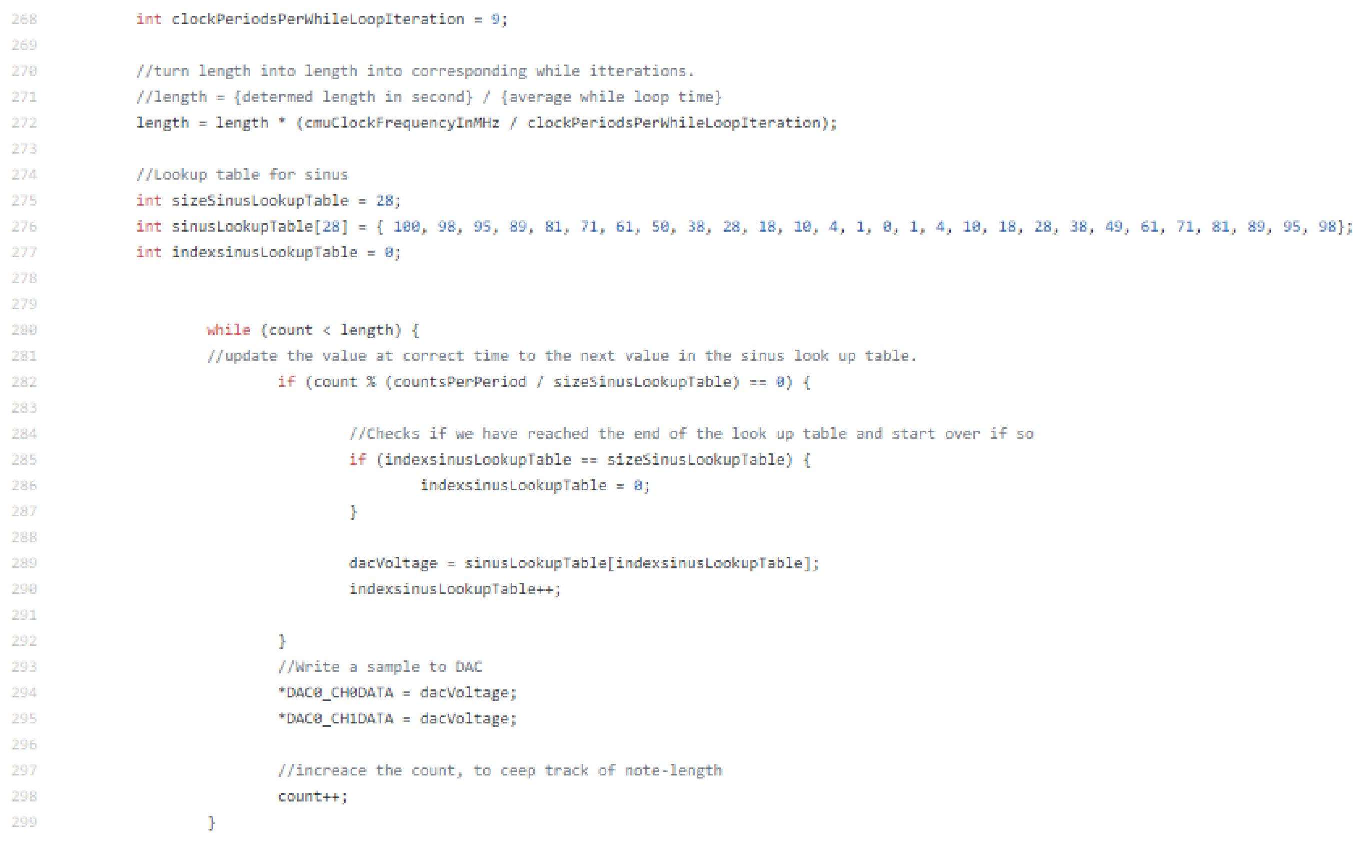
The while loop takes about 11 clock cycles for generating the triangle wave. The wave needs to change direction half period so the update time for each iterations can be found by taking $\frac{1}{2}$ count period divided by 100. If the count period is less than 200 we need to scale the rate of the voltage update as we did with the sawtooth wave. Since the count only can be integers the update rate for the voltage is also transformed into an integer. This means that we potentially can get very large voltage values for low frequencies. For example a low frequency of 150Hz have a half count period of 147 counts, but if we divide this by 100 and transform the answer into an integer we get 1. This results in that the voltage swing between 0 and 147 instead of between 0 and 100. To solve this we cut the triangle wave if it reaches over 100V or under 0V. You can see the code implemented in Figure 10.



```c
        int clockPeriodsPerWhileLoopIteration = 11;
        int incrementRate = 1;

        //turn length into length into corresponding while itterations.
        //length = {determed length in second} / {average while loop time}
        length = length * (cmuClockFrequencyInMHz / clockPeriodsPerWhileLoopIteration);

        //variable that determends dacVoltage should increacing or decresing
        int dacDirection = 1;

        //deside how often the dacVolt shuld be updated to get a value between 0 and 100 for each half period
        int dacUpTime = (countsPerPeriod/2) / 100;


        if (dacUpTime < 1) {
                dacUpTime = 1;
                if (frequency > 1102) {
                        incrementRate = 5;
                } else if (frequency > 882) {
                        incrementRate = 4;
                } else if (frequency > 662) {
                        incrementRate = 3;
                } else if (frequency > 441) {
                        incrementRate = 2;
                }
        }

        while (count < length) {

                //for each half period, change direction of DacVoltage
                if (count % (countsPerPeriod / 2) == 0) {
                        dacDirection = dacDirection * (-1);
                }

                //update the value at correct time to swing between 0 and 100 in half of a period
                if ((count % dacUpTime) == 0) {
                        dacVoltage = dacVoltage + (dacDirection * incrementRate);

                        //checks if value have gone outside range, and cut the triangle signal if it have
                        if (dacVoltage>100){
                              dacVoltage=100;
                        } else if (dacVoltage < 0) {
                                dacVoltage = 0;
                        }

                        //Give a sample to the DAC
                        *DAC0_CH0DATA = dacVoltage;
                        *DAC0_CH1DATA = dacVoltage;

                        //increace the count, to ceep track of note-length
                        count++;
                }
        }
```

Figure 10: Code for generating triangle wave.

For the sinus wave it takes approximately 9 clock ticks to iterate through the wile loop. So the value of length variable that determined how long the tone will play is adjusted accordingly. The sinus wave uses an lookup table with 28 different values to generate the sample output to the DAC. For one periode there are a given amout of samples generated. this samples will be split into 28 different values. At the start of a periode the first samples will have the value equal to the fist value in the lookup table. When the first 1/28 of the samples in the periode have played the next 1/28 samples in the periode will be set to the second value in the sinus lookup table. This continues until we reach the end of the periode. Then we start all over agian with the same prosedyre for the next periode. Since there are 28 values in the sinus lookup table the frequency cannot exceed a value that will use less than 28 samples to generate the sinus wave. Figure 11 shows the implementation of the sinus wave.



```
    int clockPeriodsPerWhileLoopIteration = 9;

    //turn length into length into corresponding while itterations.
    //length = {determed length in second} / {average while loop time}
    length = length * (cmuClockFrequencyInMHz / clockPeriodsPerWhileLoopIteration);

    //Lookup table for sinus
    int sizeSinusLookupTable = 28;
    int sinusLookupTable[28] = { 100, 98, 95, 89, 81, 71, 61, 50, 38, 28, 18, 10, 4, 1, 0, 1, 4, 10, 18, 28, 38, 49, 61, 71, 81, 89, 95, 98};
    int indexsinusLookupTable = 0;


        while (count < length) {
        //update the value at correct time to the next value in the sinus look up table.
            if (count % (countsPerPeriod / sizeSinusLookupTable) == 0) {

                //Checks if we have reached the end of the look up table and start over if so
                if (indexsinusLookupTable == sizeSinusLookupTable) {
                    indexsinusLookupTable = 0;
                }

                dacVoltage = sinusLookupTable[indexsinusLookupTable];
                indexsinusLookupTable++;

            }
            //Write a sample to DAC
            *DAC0_CH0DATA = dacVoltage;
            *DAC0_CH1DATA = dacVoltage;

            //increace the count, to ceep track of note-length
            count++;
        }
```

Figure 11: Code for generating sinus wave.

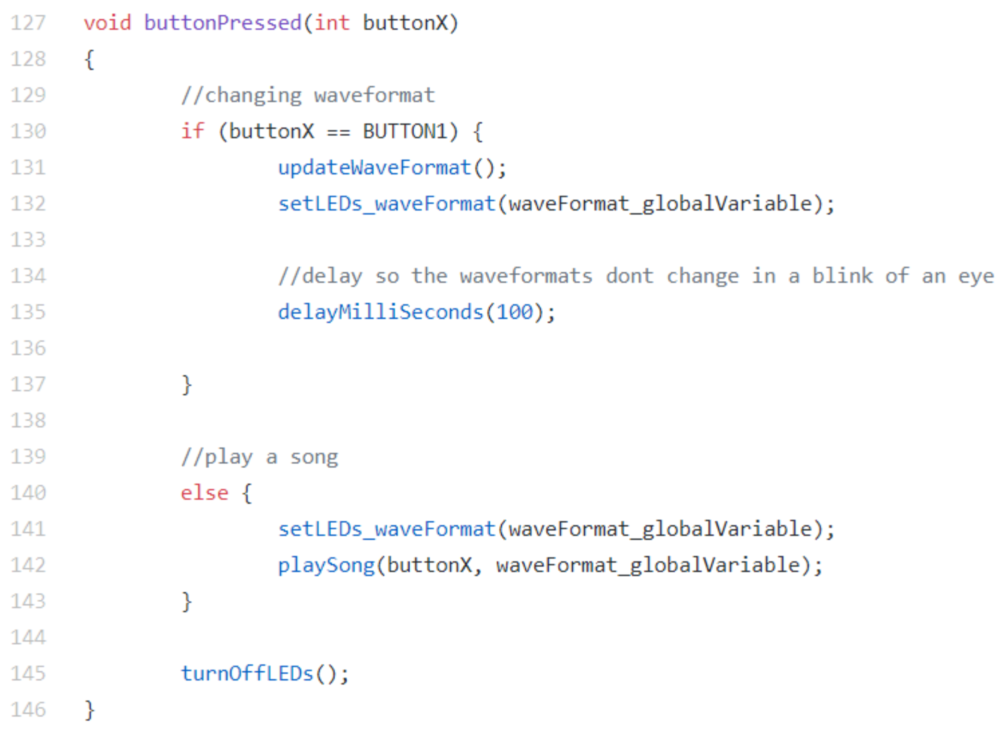
2.3.1 Song Generation

A song is generated by sending a list of notes and each note length to the sound generator. This will play the notes of the song sequentially in the current wave form.

There are 8 different songs to play. The first is the start up song that plays whenever the microcontroller is turned on or reset. This song contains the lowest to the highest frequencies we use in making the other songs, and is played using the sinus waveform. For all other songs the user can specify which of the four waveforms they want to listen to. The tempo of the songs varies between each song and is determined by how long each note is played.

2.3.2 Button Push

Button SW1 will change the waveforms between sinus, square, sawthoot and triangle wave in this order. To indicate which waveform that are currently in use the LEDs will ligth up accordingly. The two rightmost LEDs indicate a square wave, the four LEDs to the right indicate sawthoot wave. If All the LEDs are on except the two leftmost it indicates a triangle wave and all eight LEDs on indicate a sinus wave. The other buttons will each play a specific song, and is therefore sent into the playsong function as shown in Figure 12.

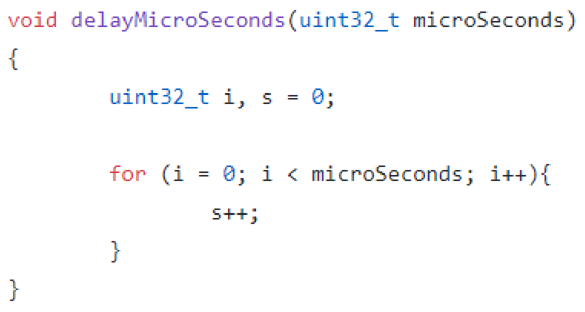


```c
void buttonPressed(int buttonX)
{
	//changing waveformat
	if (buttonX == BUTTON1) {
		updateWaveFormat();
		setLEDs_waveFormat(waveFormat_globalVariable);

		//delay so the waveformats dont change in a blink of an eye
		delayMilliSeconds(100);

	}

	//play a song
	else {
		setLEDs_waveFormat(waveFormat_globalVariable);
		playSong(buttonX, waveFormat_globalVariable);
	}

	turnOffLEDs();
}
```

Figure 12: Code for handling functionality of button press.

Here we can see that the idle states will differ for the two solutions. The busy wait solution will continuously run this function even though no buttons are pressed. This means that it will quickly turn on and off the LEDs that symbolize the wave format in use. We will see this as LEDs staying on at all times. In the interrupt solution this function only runs at button press, therefore the LEDs will be turned off, and stay off until the next button press.

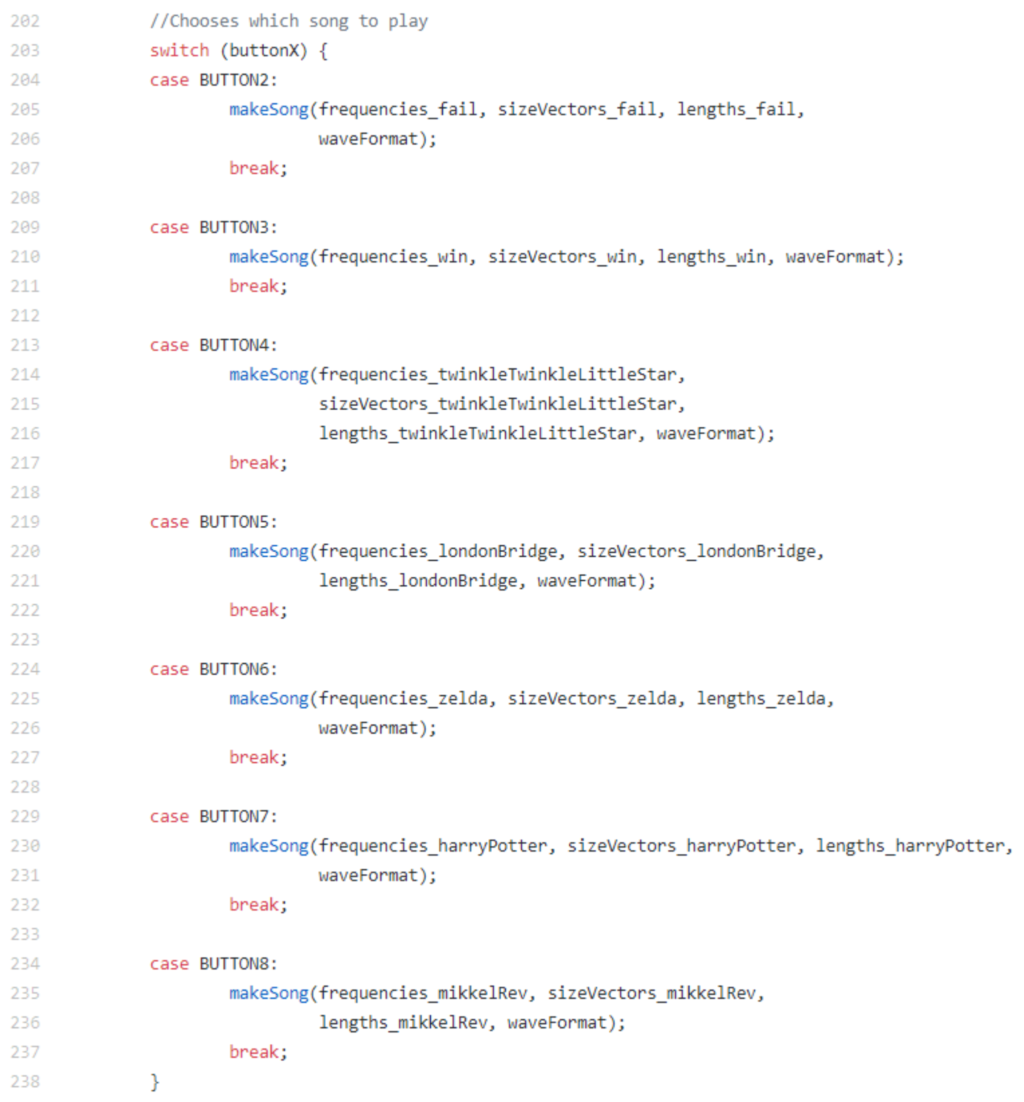
Since the polling solution checks for button push so often, we need to put in a delay after pressing button SW1. If we don't do this we can end up changing between several of the waves even though we only wanted to change one step. This is done by counting up in a for loop of our specified length. You can see the code in Figure 13.



```c
void delayMicroSeconds(uint32_t microSeconds)
{
	uint32_t i, s = 0;

	for (i = 0; i < microSeconds; i++){
		s++;
	}
}
```

Figure 13: Code for delay after button 1 is pressed

The playsong function contains seven different songs. To decide which song to play, we use a case with the buttons as input as shown in Figure 14.



```
//Chooses which song to play
switch (buttonX) {
case BUTTON2:
        makeSong(frequencies_fail, sizeVectors_fail, lengths_fail,
                 waveFormat);
        break;

case BUTTON3:
        makeSong(frequencies_win, sizeVectors_win, lengths_win, waveFormat);
        break;

case BUTTON4:
        makeSong(frequencies_twinkleTwinkleLittleStar,
                 sizeVectors_twinkleTwinkleLittleStar,
                 lengths_twinkleTwinkleLittleStar, waveFormat);
        break;

case BUTTON5:
        makeSong(frequencies_londonBridge, sizeVectors_londonBridge,
                 lengths_londonBridge, waveFormat);
        break;

case BUTTON6:
        makeSong(frequencies_zelda, sizeVectors_zelda, lengths_zelda,
                 waveFormat);
        break;

case BUTTON7:
        makeSong(frequencies_harryPotter, sizeVectors_harryPotter, lengths_harryPotter,
                 waveFormat);
        break;

case BUTTON8:
        makeSong(frequencies_mikkelRev, sizeVectors_mikkelRev,
                 lengths_mikkelRev, waveFormat);
        break;
}
```

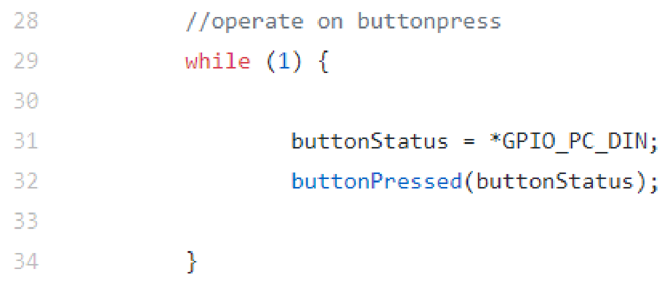
Figure 14: Code for deciding which song to play

Button two will play a self made song that we can later use to signal losing the game. This song contains three low notes with the last note longer than the rest. The next song is similarly self made and is made for the purpose of signaling winning the game. This song have nine notes and uses a quicker tempo and brighter notes. The next five songs is simplified known songs. The firs one is "twinkle twinkle little star". This uses a slow tempo and 14 notes. Opposite of the slow tempo of "twinkle twinkle little star", the song "london brige is falling down" uses a quick tempo. The song that is played when pressing the sixth button is the first part of "zeldas lullaby" and is a song from the game "the legend of zelda" series. The seventh button plays the first part of the song "hedwig's theme" from the "Harry potter" movie. The last of our songs is the norwegian children song "mikkel rev".

2.4 Busy Wait Solution

The busy wait solution plays song and switches waveformat by continuously running the function acting on button press. By doing this we have a typical busy wait situation. When a button is pressed the function makes a song play or shift waveformat, while in case of no button press it does nothing.

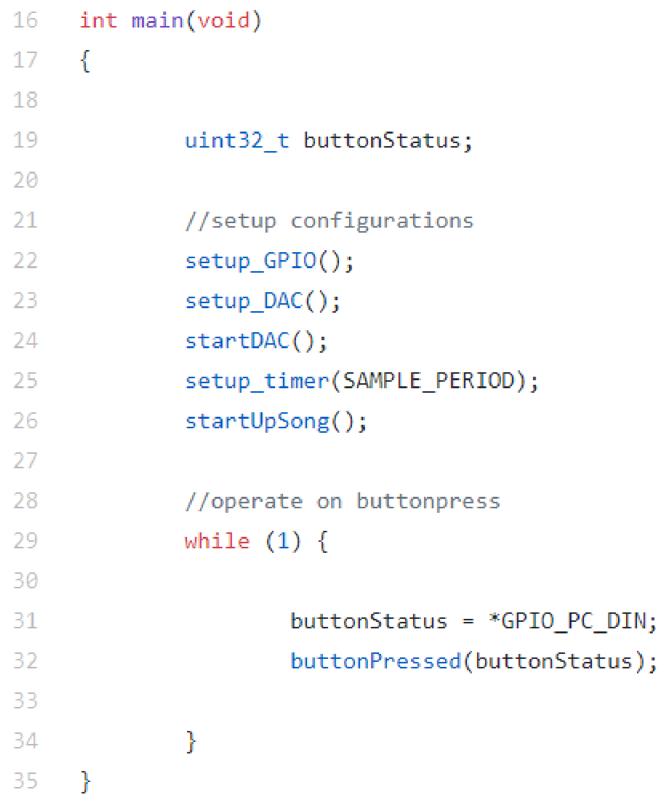
To implement this we place the function that acts on button press inside a while loop which have a condition that always makes it run. For every iteration in the while loop we get hold of the status of the button pins. This is done by checking register GPIO_PC_DIN. Next the function acting on button press is called with the status of the buttons used as an argument. The implementation of this is shown in figure 15.



```c
	//operate on buttonpress
	while (1) {

		buttonStatus = *GPIO_PC_DIN;
		buttonPressed(buttonStatus);

	}
```

Figure 15: Code for reading the input from the buttons in the busy wait solution

The main function for the busy wait solution will therefore only consist of the setup for the DAC, the timer and the GPIO, a start up song, and the busy wait structure. Figure 16 shows this.



```c
int main(void)
{

	uint32_t buttonStatus;

	//setup configurations
	setup_GPIO();
	setup_DAC();
	startDAC();
	setup_timer(SAMPLE_PERIOD);
	startUpSong();

	//operate on buttonpress
	while (1) {

		buttonStatus = *GPIO_PC_DIN;
		buttonPressed(buttonStatus);

	}
}
```

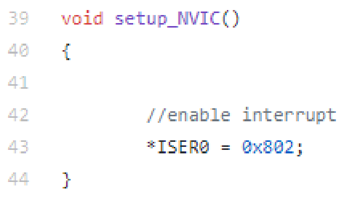
Figure 16: The main function for the polling solution

2.5 Interrupt Solution

The interrupt solution plays song and switches waveformat by running the function acting on button press only when a button actually is pressed, and not continuously like the busy wait solution. To do this the solution uses interrupts.

One of the factors that makes this interrupt solution reduces the overall energy consumption compared to the busy wait solution, is that it is constructed so it use a combination of interrupts and low power mode. To have maximum energy reduction in low power mode we use the SLEEPDEEP in this mode.

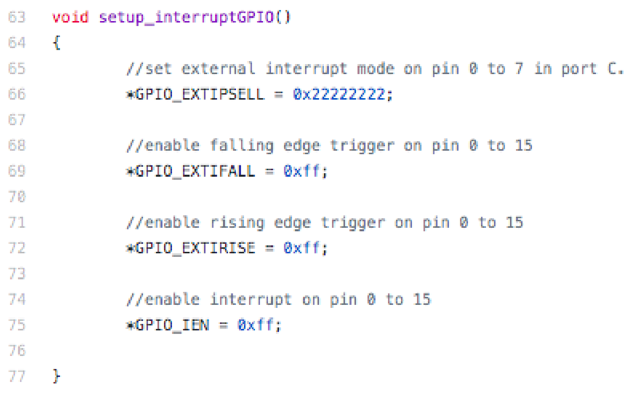
To enable the interrupt functionality in the microcontroller we need to set register ISER0 to 0x802. This is shown in figure 17.



```
void setup_NVIC()
{

        //enable interrupt
        *ISER0 = 0x802;
}
```

Figure 17: Code for enabling interrupt

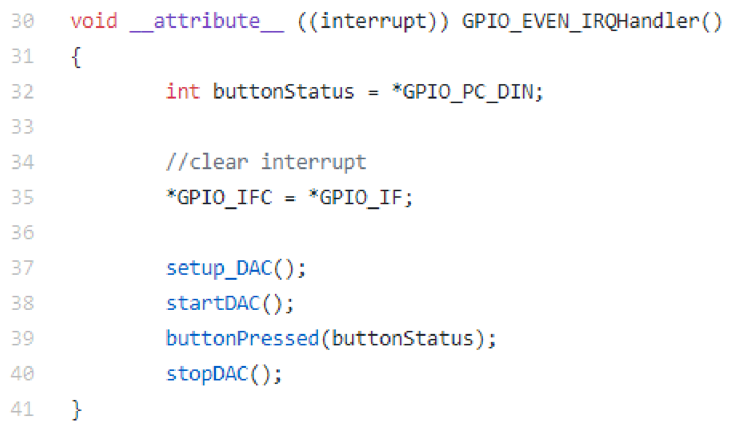
In this solution we further configure the GPIO pins so that they support interrupt behaviour. This is done by setting GPIO_IEN to 0xff. This enables interrupt on GPIO pins in general. GPIO_EXITPSELL is set to 0x22222222. This enables external interrupt mode on pin 0 to 7 in port C. Both registers GPIO_EXITFALL and GPIO_EXITRISE is set to 0xff. This enables triggers on respectively falling and rising edge. The figure 18 shows how this is implemented.



```
void setup_interruptGPIO()
{
        //set external interrupt mode on pin 0 to 7 in port C.
        *GPIO_EXTIPSELL = 0x22222222;

        //enable falling edge trigger on pin 0 to 15
        *GPIO_EXTIFALL = 0xff;

        //enable rising edge trigger on pin 0 to 15
        *GPIO_EXTIRISE = 0xff;

        //enable interrupt on pin 0 to 15
        *GPIO_IEN = 0xff;

}
```

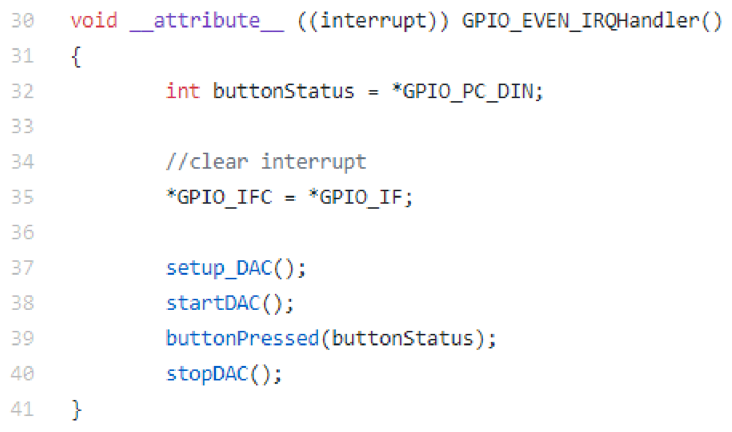
Figure 18: Code for setting up the interrupt

When GPIO pins are in interrupt mode there are special functions that handles the interrupts when they occur. There are two different interrupt handler, one that handles interrupts caused by odd pins, and one that handles interrupts caused by even pins. These to handlers will handle the interrupt in similar way. Both clears the interrupt, start the DAC, act on the given button press, and stop the DAC in the given order. The figures 19 and 20 shows respectively how the interrupt handler for the odd and even button pins are implemented.



```c
void __attribute__ ((interrupt)) GPIO_EVEN_IRQHandler()
{
        int buttonStatus = *GPIO_PC_DIN;

        //clear interrupt
        *GPIO_IFC = *GPIO_IF;

        setup_DAC();
        startDAC();
        buttonPressed(buttonStatus);
        stopDAC();
}
```

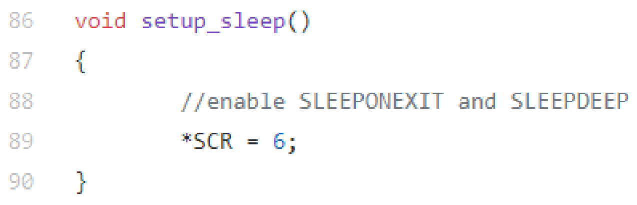
Figure 19: The GPIO interrupt handler for the even button pins.



```c
void __attribute__ ((interrupt)) GPIO_EVEN_IRQHandler()
{
        int buttonStatus = *GPIO_PC_DIN;

        //clear interrupt
        *GPIO_IFC = *GPIO_IF;

        setup_DAC();
        startDAC();
        buttonPressed(buttonStatus);
        stopDAC();
}
```

Figure 20: The GPIO interrupt handler for the odd button pins.

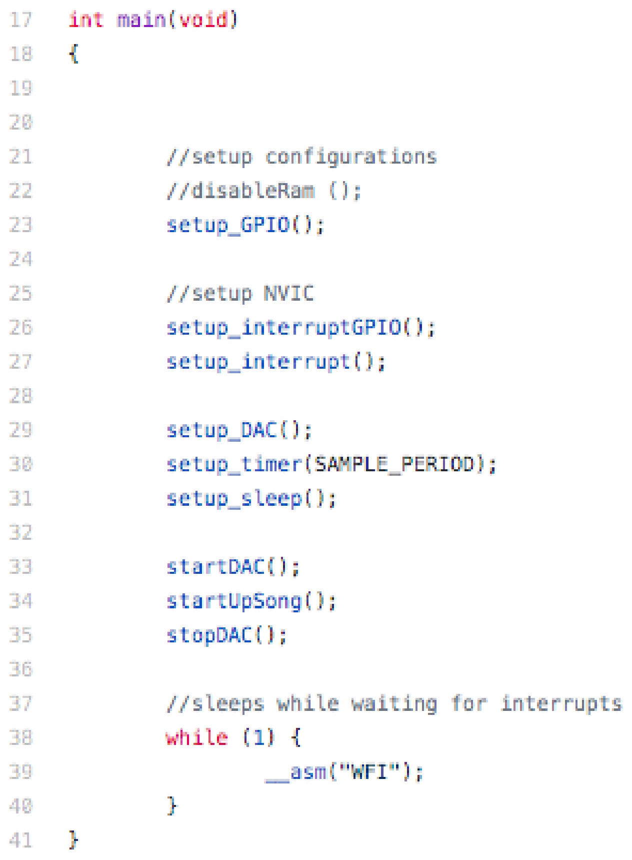
To make the microcontroller go into low power mode when not handling an interrupt, bit 1 in register SCR is set high. The low power mode is set to SLEEPDEEP mode by setting bit 2 in SCR high. How this is implemented is shown in figure 21.



```c
void setup_sleep()
{
        //enable SLEEPONEXIT and SLEEPDEEP
        *SCR = 6;
}
```

Figure 21: Code for enabling sleep mode.

The running function in this solution is called main. This function is similar to the busy wait running functions in the form that it first run setups, then a start melody continued by an infinity loop. What differs is that it in addition to the setups in the interrupt solution sets up interrupt functionality. In addition the while loop do not contain the act on button press function, but an interrupt function. This function let the microcontroller stay in low power mode as long as it does not handle a button press. The figure 22 shows how this is implemented.



```c
int main(void)
{


        //setup configurations
        //disableRam ();
        setup_GPIO();

        //setup NVIC
        setup_interruptGPIO();
        setup_interrupt();

        setup_DAC();
        setup_timer(SAMPLE_PERIOD);
        setup_sleep();

        startDAC();
        startUpSong();
        stopDAC();

        //sleeps while waiting for interrupts
        while (1) {
                __asm("WFI");
        }
}
```

Figure 22: The main function for the interrupt solution.

3 Comparison

3.1 Polling solution

Two LEDs are on when the square wave is selected. When the DAC is on but no sound is made we have a power consumption as shown in Figure 23. Here the CPU constantly check for the next event.

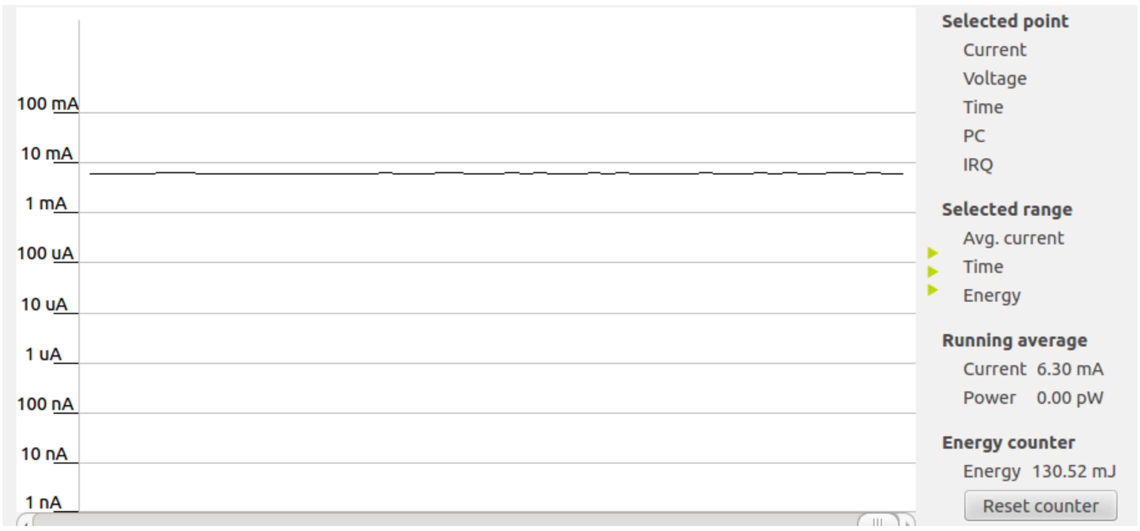


Figure 23: Idle state, with 2 LEDs on

When we do play a song, the CPU stops checking for events and instead play the song that was specified. In this case we gets a lower current consumption as shown in Figure 24. The song is playing the different notes correctly, but the sound is more high pitched than the rest of the wave forms.

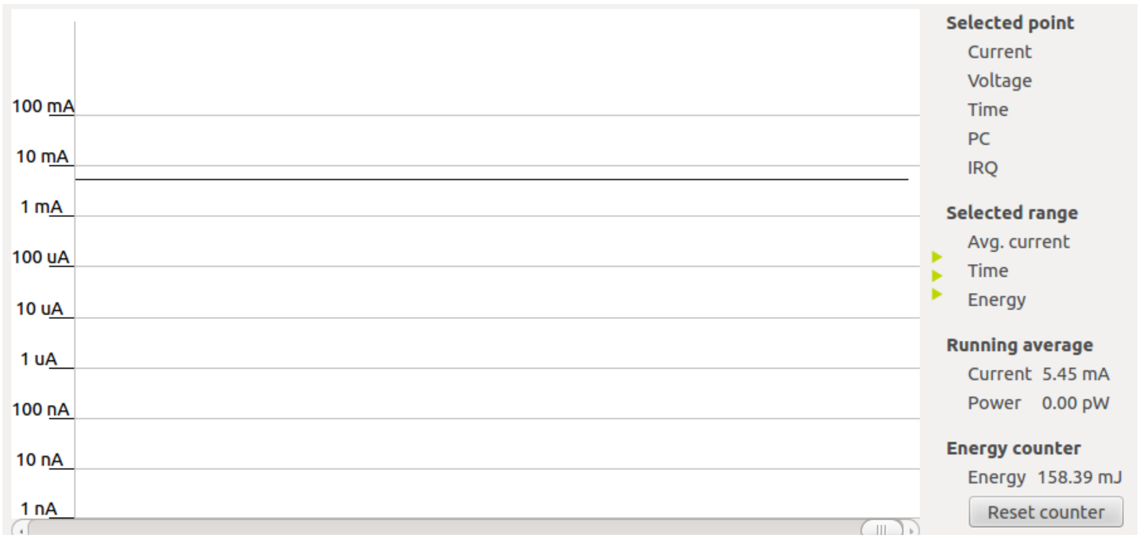


Figure 24: Playing a song with square wave (2 LEDs on)

For the sawtooth wave we have four LEDs on when this is selected. This gives us a current consumption of 0.95mA more than for when we only had two LEDs on as shown in Figure 25.

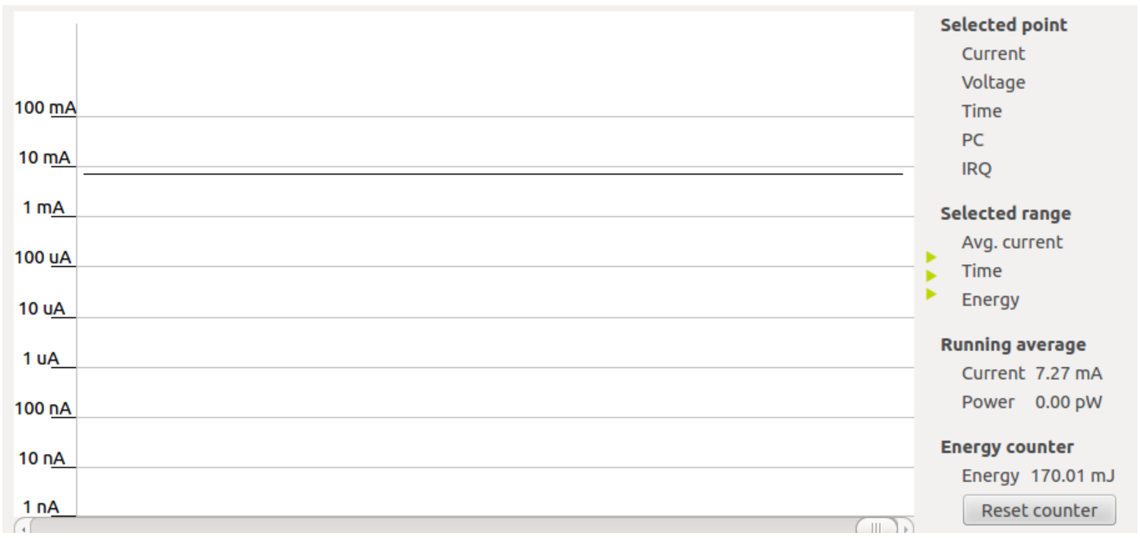


Figure 25: Idle state, with 4 LEDs on

The sawtooth wave also consumes less current when it plays a song as shown in Figure 26. If we subtract the current value with the current increase of having four LEDs instead of two LED we get a current consumption of 5.48 mA. This value is 0.03 mA larger than the current consumption of the song played using the square wave, meaning the sawtooth wave uses more current than the square wave. The sound of each note is a bit different from expected, the tone is not clean. You can also hear a jarring sound throughout the entire song.

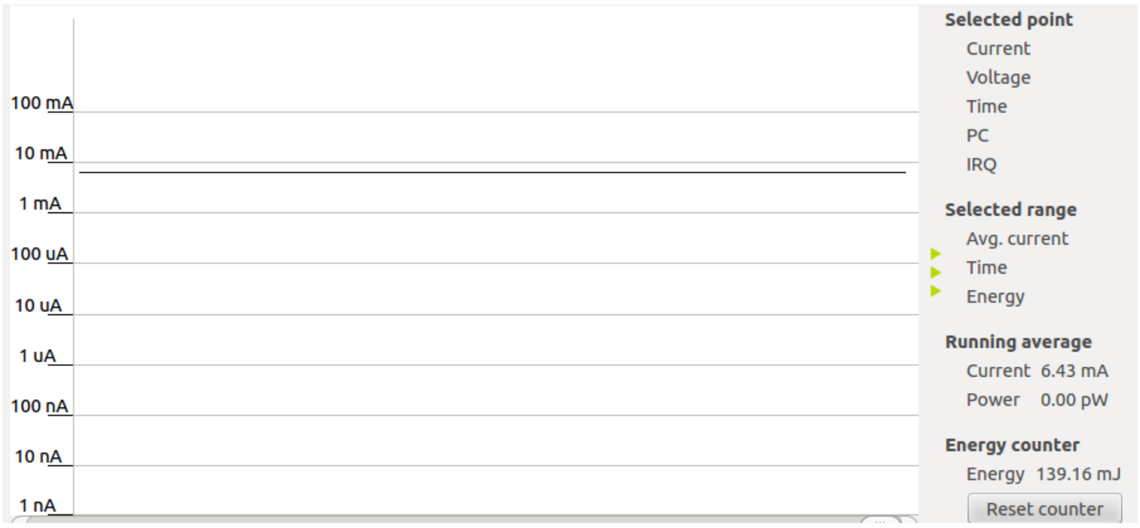
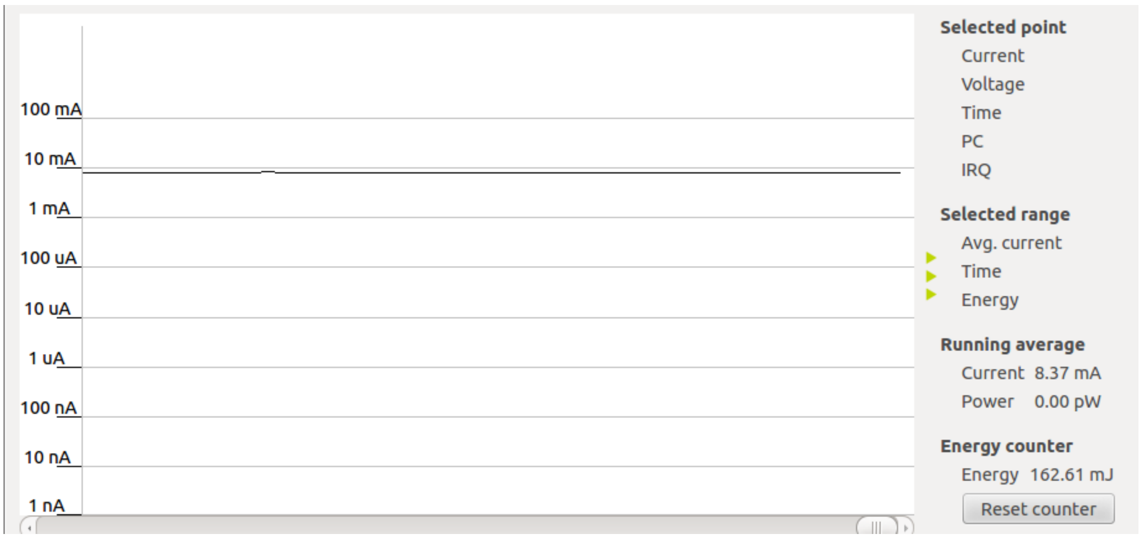


Figure 26: Playing a song with sawtooth wave (4 LEDs on)

A triangle wave is indicated by having six LEDs on. This uses 8.37 mA as shown in Figure 27. This is 1.1 mA more than when four LEDs are on, and 2.07 mA more than for when 2 LEDs are on.



Selected point
Current
Voltage
Time
PC
IRQ

Selected range
Avg. current
Time
Energy

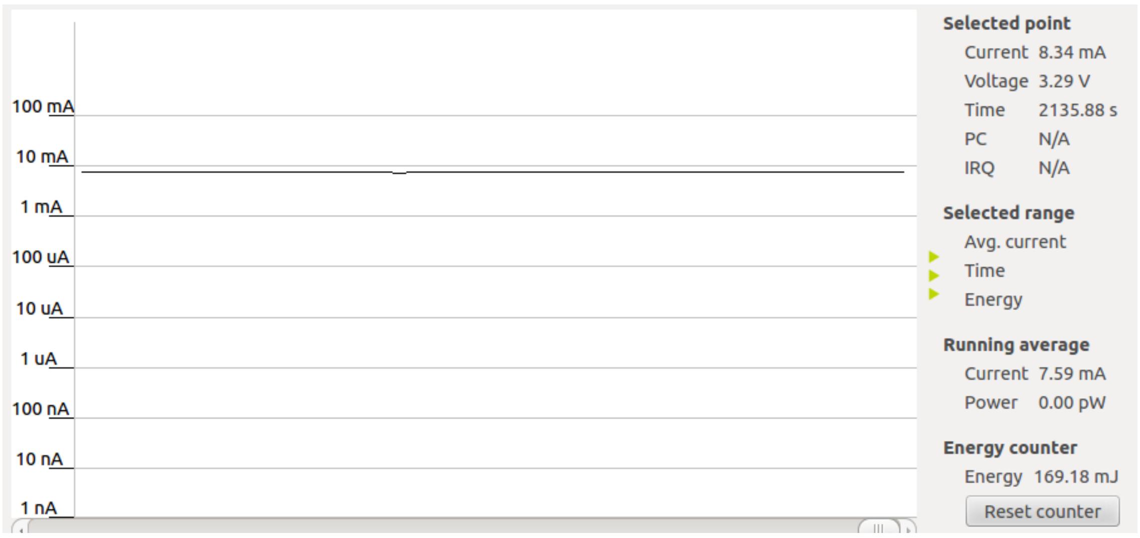
Running average
Current 8.37 mA
Power 0.00 pW

Energy counter
Energy 162.61 mJ

Reset counter

Figure 27: Idle state, with 6 LEDs on

A song played with a triangle wave uses 7.59 mA as shown in Figure 28. If we subtract the voltage needed to drive six LEDs compared to two LEDs we get that the current consumption of 5.52 mA. This is 0.07 mA more than for square waves and 0.04 mA more than sawtooth waves. The sound of the song is better than it was for square wave and sinus wave. Now the tones are clearer and less high pitched, but there are some of the tones hat still is a bit off key.



Selected point
Current 8.34 mA
Voltage 3.29 V
Time 2135.88 s
PC N/A
IRQ N/A

Selected range
Avg. current
Time
Energy

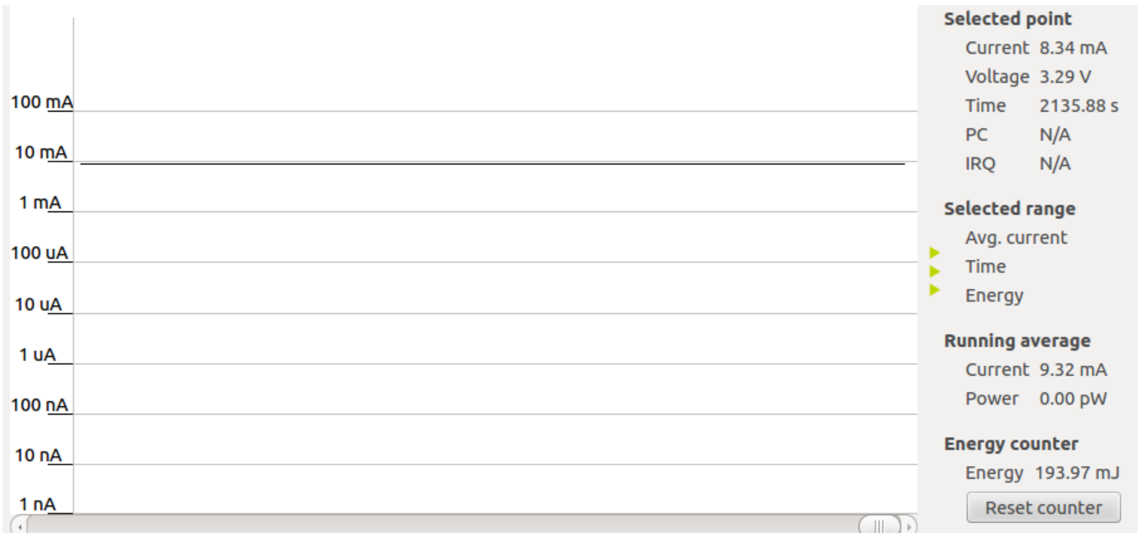
Running average
Current 7.59 mA
Power 0.00 pW

Energy counter
Energy 169.18 mJ

Reset counter

Figure 28: Playing a song with triangle wave (6 LEDs on)

The sinus wave is indicated by having 8 LEDs on and this uses 9.32 mA. this is 0.95 mA more than six LEDs, 2.05 mA more than four LEDs and 3.02 mA more than two LEDs.



Selected point
Current 8.34 mA
Voltage 3.29 V
Time 2135.88 s
PC N/A
IRQ N/A

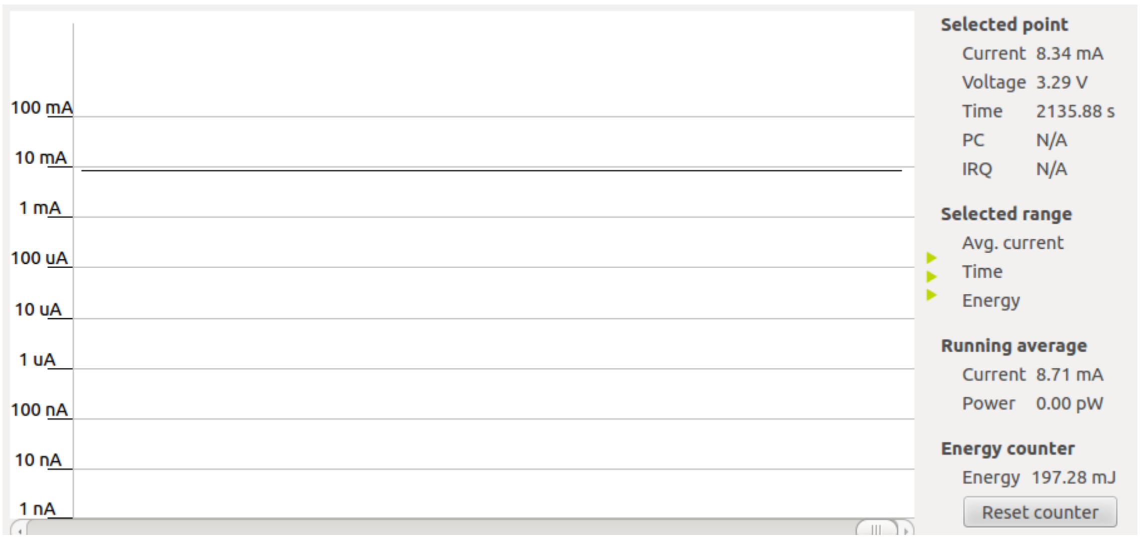
Selected range
Avg. current
Time
Energy

Running average
Current 9.32 mA
Power 0.00 pW

Energy counter
Energy 193.97 mJ

Figure 29: Idle state, with 8 LEDs on

Playing a song with the sinus wave uses 8.71 mA as shown in Figure 30. If we subtract the difference between having eighth LEDs on compared to having 2 LEDs on we get a current consumption of 5.69 mA. This is 0.24 mA more than the square wave, 0.21 mA more than a sawtooth wave and 0.17 mA more than for a triangle wave.



Selected point
Current 8.34 mA
Voltage 3.29 V
Time 2135.88 s
PC N/A
IRQ N/A

Selected range
Avg. current
Time
Energy

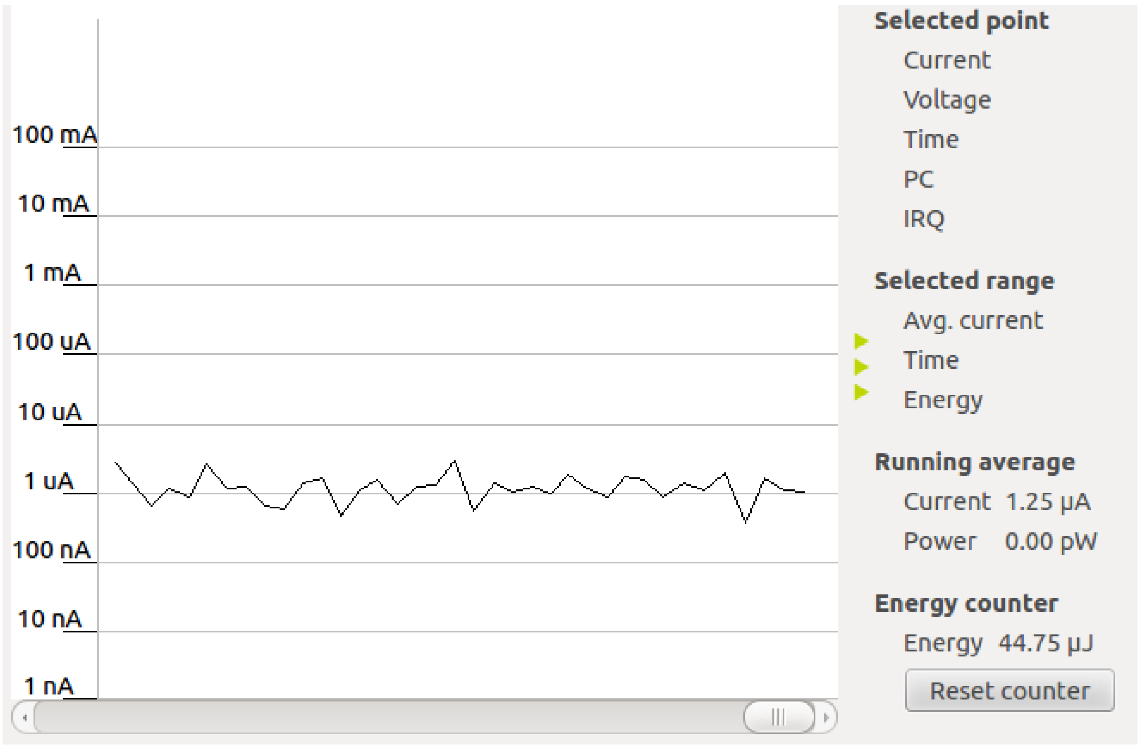
Running average
Current 8.71 mA
Power 0.00 pW

Energy counter
Energy 197.28 mJ

Figure 30: Playing a song with sinus wave (8 LEDs on)

3.2 Interrupt solution

The measurement taken in the interrupt solution are for idle state, when a song is playes with the waveforms square, sawtooth, triangle and sinus. When a song is played with a square wave form 2 LEDs are lit, with sawthoot 4 LEDs are lit, with triangle 6 LEDs, and with sinus 8 LEDs. The figures below shows the measurements taken. Which measurements the figure show is specified in the figure text.



Selected point
Current
Voltage
Time
PC
IRQ

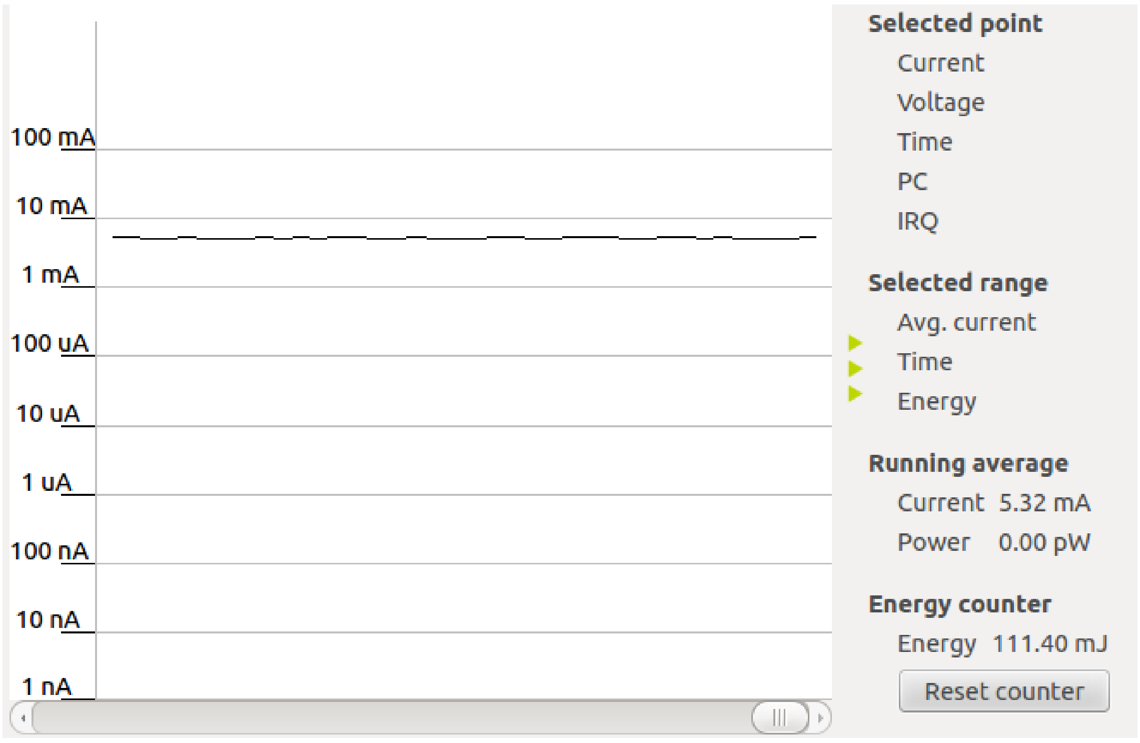
Selected range
Avg. current
Time
Energy

Running average
Current 1.25 μA
Power 0.00 pW

Energy counter
Energy 44.75 μJ

Figure 31: The idle state of the interrupt solution.

In Figure 31 we see the interrupt solution while it has entered sleep mode. Here no LEDs are on, and we have turned the DAC of. on an average we see that this state uses 1.25 μA, however it does variate a bit.



Selected point
Current
Voltage
Time
PC
IRQ

Selected range
Avg. current
Time
Energy

Running average
Current 5.32 mA
Power 0.00 pW

Energy counter
Energy 111.40 mJ

Figure 32: A song with square waveform is played, and two LEDs are lit.

when we play a song using the square wave we uses 5.32 mA as shown in Figure 32. This is 0.13

mA less than what it is for the polling solution. The sound here is similar to what it was for the polling solution, meaning that it still have a high pitch.

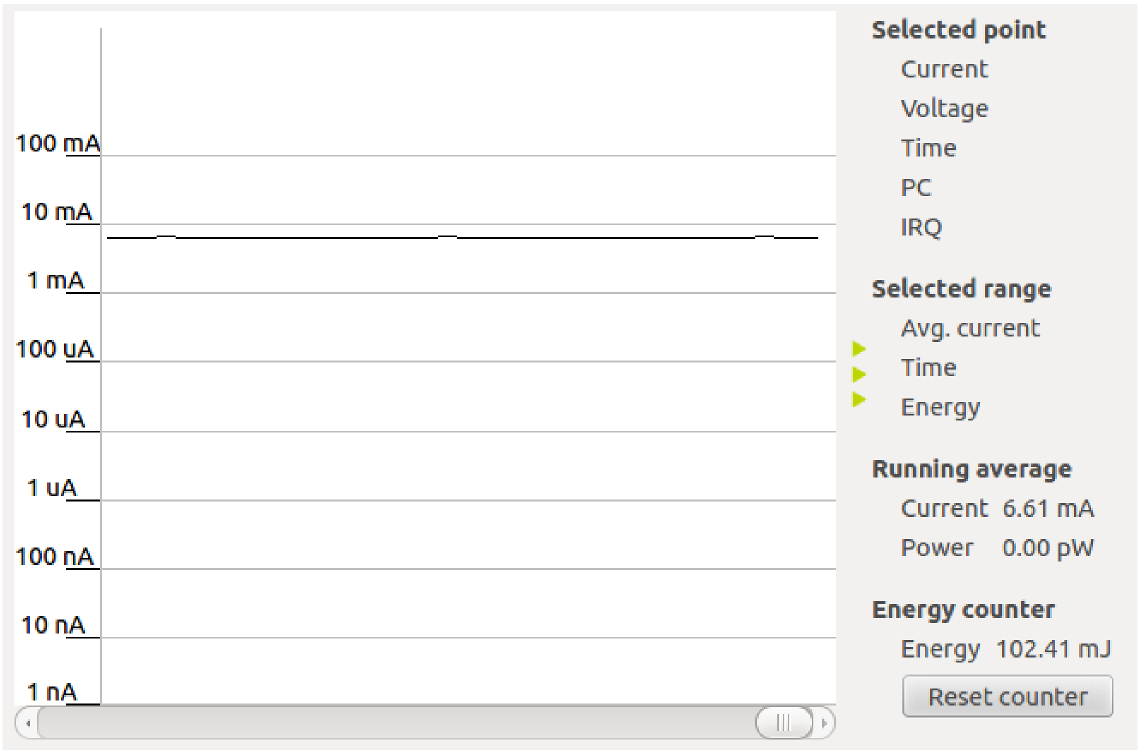
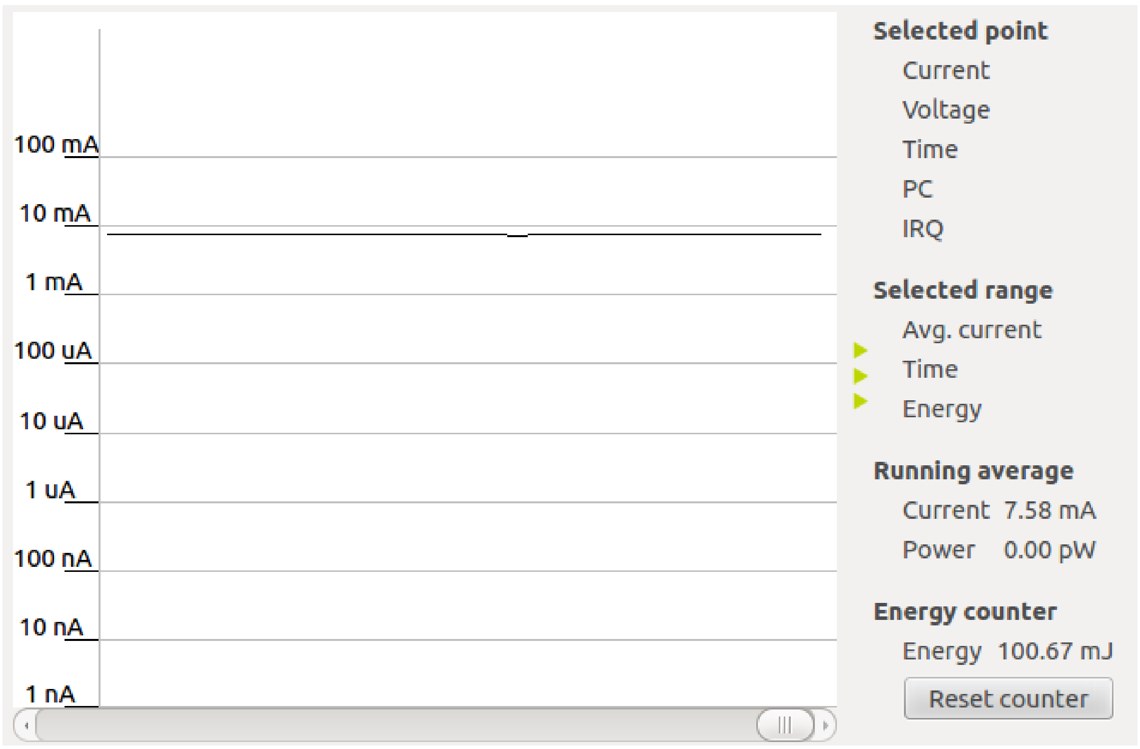


Figure 33: A song with sawtooth waveform is played, and four LEDs are lit.

The sawtooth wave uses 6.61 mA when playing a song as shown in Figure 33. This wave is indicated by four LEDs as in the polling solution, but it uses 0.18 mA more current. The sound of each note is still a bit off key.

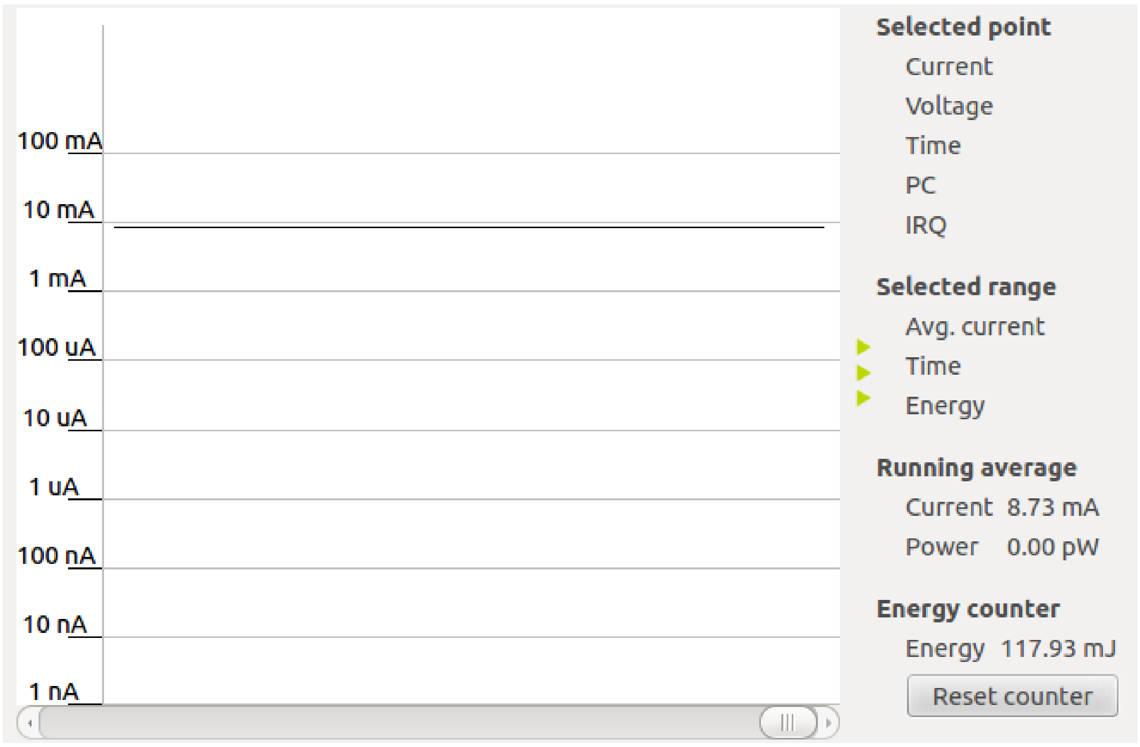


Running average
Current 7.58 mA
Power 0.00 pW

Energy counter
Energy 100.67 mJ

Figure 34: A song with triangle waveform is played, and six LEDs are lit.

In Figure 34 we show that the current consumption when playing a song using triangle wave is 7.58 mA. This value is 0.01 mA less than for the polling solution. The sound quality is the same as for the polling solution.



Running average
Current 8.73 mA
Power 0.00 pW

Energy counter
Energy 117.93 mJ

Figure 35: A song with sinus waveform is played, and eigth LEDs are lit.

Lastly we see the current consumption of using a sinus wave. This is indicated by eighth LEDs and uses 8.73 mA as shown in Figure 35. This value is also more than what we used in the polling solution, but only by 0.02 mA.

3.3 Comparison

Table 1: Comparing the current results

Wave	Polling solution	Interrupt solution	Difference
Square wave	5.45 mA	5.32 mA	-0.13 mA
Sawtooth wave	6.43 mA	6.61 mA	0.18 mA
Triangle wave	7.59 mA	7.58 mA	-0.01 mA
Sinus wave	8.71 mA	8.73 mA	0.02 mA

In Table 1 we compare the current consumption of the polling solution and the interrupt solution. We observe that the interrupt solution uses less current for the square wave and the triangle wave, but more for the sawtooth wave and the sinus wave. However the difference in preformance is less than 0.2 mA, so it is not much difference between the solutions.

4 Observation and Discussion

In the polling solution we see that playing a song uses less current than when the processor is constantly checking if there have happened an event. For the interrupt solution we have implemented a power saving techniques that allows the processor to use minimum amount of current while it is waiting for an event. The songs are played in a slow tempo compared to the processors clock. This means that there are no need to consider the respond of the GPIO at a high frequency. So with regards to power consumption we conclude that the interrupt solution is better.

The sound is equal in bout solutions. In regards to sound quality the sinus wave is best, while the cut triangle wave comes close behind. The sawtooth wave sound the most wrong and unpleasant of the four waves. However all waves manages to play the different frequencies in each song at our specified tempo, so we are able to differentiate the songs in each of the wave forms.

Bibliography

Appendix