NTNU – Trondheim
Norwegian University of
Science and Technology

TDT4258 Energy Efficient Computer Design
Laboratory Report

Exercise 2 - DAC

Group 29:

Anders Lima
Kjetil Aune

March 8, 2015

Abstract

An abstract is a short (100 to 500 words), high-level summary of the entire document. For this kind of report, you would start by introducing the concept that the report talks about and the goals of the work, followed by information about how the work was done and some summary of results.

This report is a hands-on approach towards learning energy efficient programming using primarily the C language. The work is done on the EFM32GG-DK3750 microcontroller from Silicon Labs, but it is applicable for programming other types of microcontrollers as well. This report describes how to utilize the on-board DAC in order to make music and sound effects, but still maintaining a low energy consumption. It is desirable to keeping the energy consumption low for a lot of reasons. Microcontrollers are used in lot of energy-sensitive areas, often battery powered, like implantable medical devices, remote controls and toys, so you want to increase their life time by decreasing the energy usage.

NEEDS TO BE WRITTEN WHEN WE HAVE RESULTS!!!

On RUNNING

- (Tried to down-scale, but dropped due to sound quality)
- Reduced interrupt-frequency
- EM1 (gets rid of 200 interrupts per note)

On IDLE

- Turned off DAC
- Turned off Timer
- EM2 (WFI)

Generated tones beforehand, so we didn't need to do a lot of heavy divisions for each boot. Used shifting in stead of multiplication/division. Decided using integers instead of floats for the tones.

TRY Update linker script when disabling RAM-blocks 32.5.1 GPIO$_{P}x_{C}TRL-PortControlRegister11.5.3$
$HighFrequencyPeripheralClockDivisionRegister$

COULD TRY Decrease voltage

1 Introduction

Your report should start with an introduction chapter that motivates the subject in general and describes the problem you are trying to solve.

2 Background and Theory

This chapter will provide information about hardware, software and different topics important for understanding the results.

2.1 Hardware

The hardware used in this exercise is the EFM32GG-DK3750 development board from Silicon Labs depicted in Figure 2.1. It was connected to a personal computer via USB. It also includes a TFT screen which can display real-time energy consumption. The EFM32GG uses the energy efficient 32-bit ARM Cortex-M3. It's worth mentioning that the M3 uses Harvard architecture [?], which allows it to read input at the same as it can execute instructions.

2.1.1 Sound and DAC

The board has an integrated DAC (digital-to-analog converter), which will be used for playing music and sound effects. A DAC takes input as digital data, a stream of binary values, and outputs analog values in the form of voltage or current.

Sound is best described mathematically as sine waves. These waves pushes the air at a given frequency. This frequency determines the pitch or tone. The human ear can hear sound with frequencies in the range from roughly 20Hz to 20kHz.

Since a DAC takes discrete values as input, you can generate an output that best resembles the sound by taking samples of the sine wave. The more samples you have per period, the higher the quality of the output.

2.1.2 Timer and Clock

The EFM32GG has a high frequency clock, HFRCLK, which is usually driven by the high-frequency oscillator HFRCO, but can also be driven by another high-frequency oscillator, or one of the low-frequency oscillator. The HFRCLK drives the prescalers that generate the high-frequency peripheral clock (HFPERCLK) and the high-frequency core clock (HFCORECLK). The HFRCLK can be divided down to run on a lower frequency, which then in turn will lower the HFPERCLK and HFCORECLK.

The EFM32GG also has

20.3.1 Counter Modes START HERE NOW, AND DON'T WINE ABOUT BEING HUNG OVER.

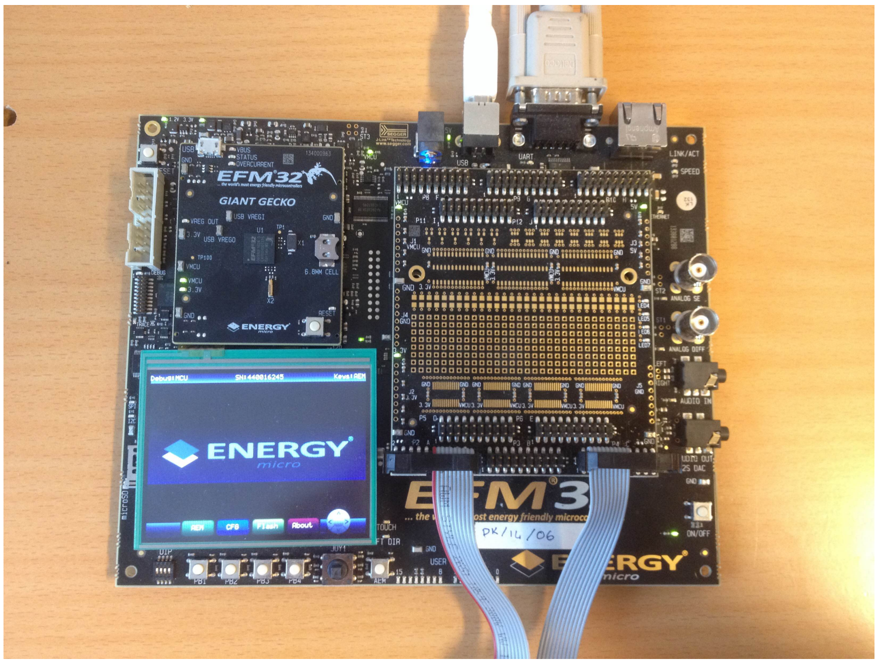

Figure 2.1: EFM32GG-DK3750

2.1.3 Gamepad

In addition we had a gamepad with eight buttons and eight LEDs depicted in Figure 2.2. This was connected to the development board's GPIO pins. The gamepad includes a jumper, that when set to the lower two pins will exclude the LEDs form the power measurement.

2.2 Static vs dynamic power consumption

The total power consumption is the sum of static- and dynamic power consumption, given by Formula 2.1, acquired from the course's lecture notes:

$$P_{tot} = P_{dynamic} + P_{static} = \alpha C V^2 f + I_{static} V \tag{2.1}$$

There is static power consumption due to leakage in the transistors. The gate oxide layer in a transistor is not a perfect insulator, and therefore there will always be some current leaking through an active transistor.

The dynamic power consumption comes from when a transistor changes state, i.e. goes from 0 to 1 or vice versa, and is dependant on the capacitance of the transistors. The thing worth mentioning is that the voltage is squared, and it is key to keep the voltage as small as possible.

The total energy consumption is given by:

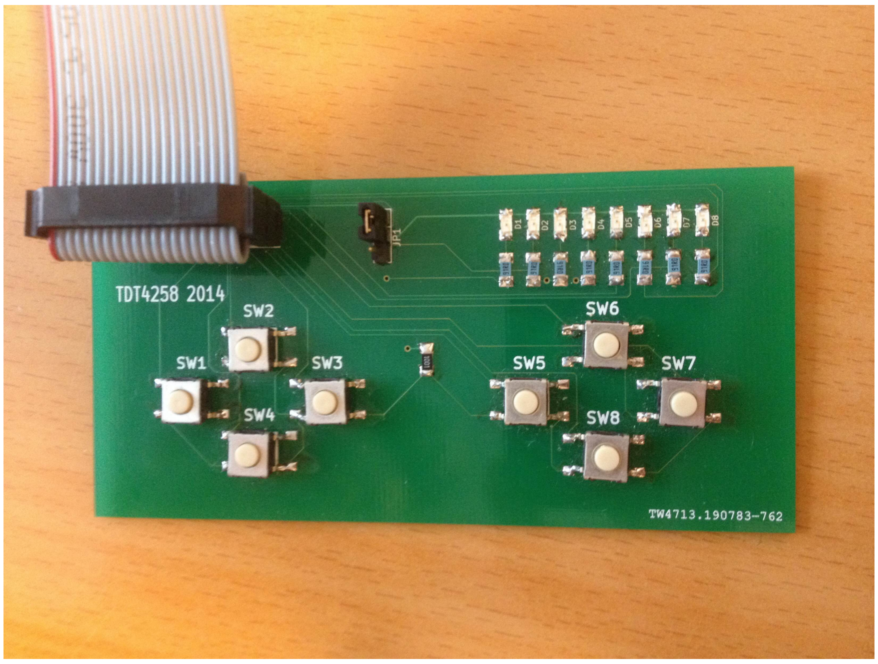

Figure 2.2: Gamepad

$$E = P_{total}t \tag{2.2}$$

The static power consumption of a system is constant, but the dynamic consumption is dependant on the clock frequency. The time it takes for a certain amount of work to complete is dependent on the clock. Therefore, reducing the clock frequency will actually increase the total energy usage, because the program will need to run for a longer period. The static power consumption is present for a longer time, and as a result, it is often best to do the calculations without downscaling the clock, and then turn the components off.

2.3 The C language

C is a low-level, imperative programming language. It is a general purpose language, which also provides a lot of control for the programmer with regards to memory control and memory access. A very useful part of C is pointers. Pointers contain addresses to memory locations, and can be dereferenced to obtain the value on the given location. This is very useful when programming microcontrollers, because it makes it easy to write and obtain values directly from memory.

2.4 Energy Efficiency

There are a lot of ways of reducing the power needed to run programs, and the following section describes some of these techniques.

2.4.1 Energy Modes on the EFM32GG

The EFM32GG has five different energy modes, all with different levels of active components and power consumption. Figure 2.3 shows an overview of the EFM32GG and Figure 2.4 shows a color scheme with the different energy modes indicated with a respective number. The colors in Figure 2.3 translates to the highest energy mode in which they are active.

Here are some of the key aspects worth noticing about each energy mode:

EM0 - Run Everything is active.

EM1 - Sleep The CPU is sleeping, the rest is active.

EM2 - Deep Sleep High frequency oscillators, flash memory, timer/counter are inactive

EM3 - Stop RAM, DAC and external interrupts are still active.

EM4 - Shutoff Everything is shut off, only a reset can turn it on again.

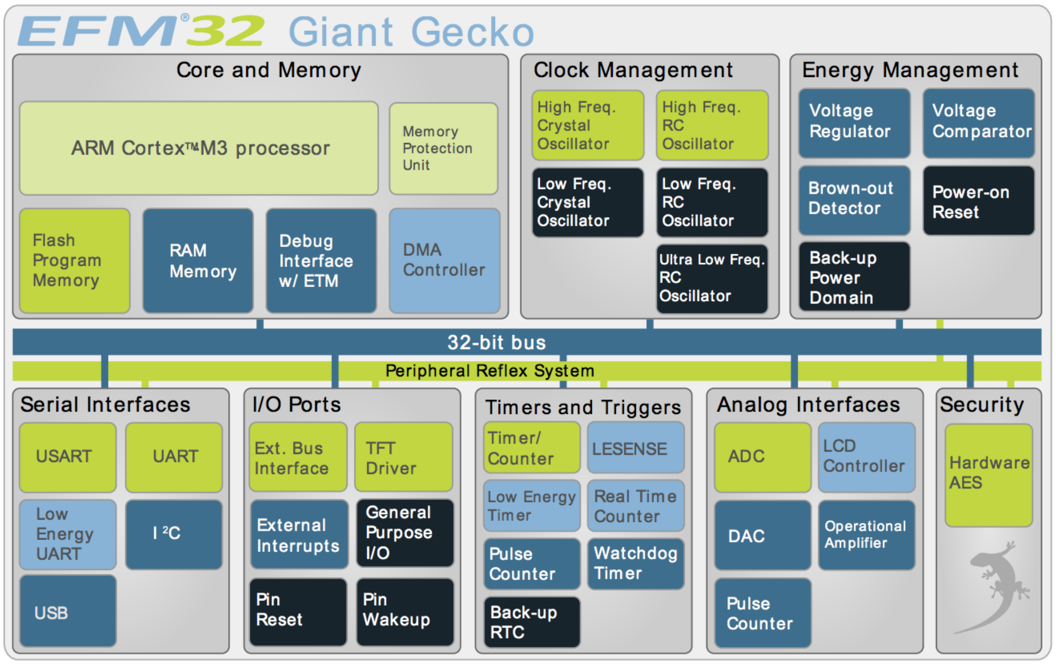

Figure 2.3: Diagram of EFM32GG

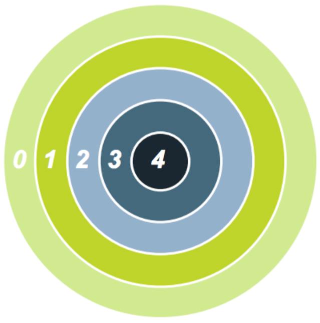

Figure 2.4: Energy Mode Indicator

2.4.2 Interrupts

When a program is idle, there are several ways to wake it up. One way is by using polling. Polling is when you repeatedly check the status of a unit to see if there are any changes or if it is ready for use. This is not very efficient with regards to energy efficiency, because you continuously check to see if there is work to be done, and you use a lot of processing power.

A good replacement for polling is the use of interrupts. An interrupt is a signal to the processor indicating that an event needs immediate attention. When the processor receives an interrupt signal, it suspends its current program and executes a handler routine defined in the interrupt handler vector. After the handler routine is completed, the processor returns to the program it was originally executing[**?**]. The use of interrupts will relieve the processor of a lot of work, and only do work when it's needed, which in turn decreases the energy consumption.

2.4.3 Reducing Static Power Consumption

As described in Section 2.2, there is always static power consumption when a transistor is active. One way of reducing the total energy consumption, is by simply turning off parts of the system that are not in use, and thus decreasing the total static power consumption.

3 Methodology

This chapter will describe the workflow, and a detailed explanation of our implementation. At the end of this chapter we will discuss the testing of our implementation.

3.1 Workflow and project structure

The code were mostly written in Emacs on Ubuntu 14.04 LTS running in VirtualBox on a Windows8.1 laptop. Git was used to transfer files between the virtual machine and other computers used write code.

The file structure is shown in 3.1.

Listing 3.1: Project structure

```
|-- buttons.h
|-- core
|-- dac.c
|-- efm32gg.h
|-- ex2.bin
|-- ex2.c
|-- gpio.c
|-- interrupt_handlers.c
|-- lib
|   |-- efm32gg.ld
|   |-- ex2.bin
|   |-- libefm32gg.a
|-- Makefile
|-- songs.h
|-- timer.c
|-- tone-generator.py
|-- tones.h
```

Header files

All the header files are used to store predefined variables. efm32gg.h contains memory locations for the most used registers. tones.h contains implementation specific values for the different tones used, explained further in section 3.2.2. songs.h contains a struct, defining how we store songs, and also contains the songs and sound effects used in this exercise. buttons.h contains the register values on different key presses, for easy comparison on GPIO interrupt.

C-Files

ex2.c is the main file, which is run at power-on. gpio.c, timer.c and dac.c has code for setting up and powering on and off the different parts. interrupt$_{h}andlers.ccontainsthecodeforhandlinginterru$

3.1.1 Bulding and flashing

The GNU-Toolchain was used to compile and build the C-code. This was done using the `make` command. The binary was then uploaded to the board, via USB, using Silicon Labs' eACommander.

3.1.2 Debugging

We were not able to install the gdb tool for ARM on the virtual machine. This made debugging quite hard, and lead to a try and fail approach. If we needed to know the value of a variable or a register, we wrote the binary value to the gamepad's LEDs.

3.2 Implementation

To get started, and to get familiar with programing a microcontroller using the C language, we reimplemented the functionality of Exercise 1 [**?**]. This included the setup of GPIO, clocks and interrupts.

3.2.1 Setting Up

To make thing work, we need to run some initial code to set things up.

GPIO

To make the General-Purpose-Input-Output work, we first enable the GPIO clock by setting a value to the CMU register. Then we enable high drive strength and set pin 8 to 15 as output, on Port A, for the gamepad's LEDs. We also need to make the buttons work, which is done by writing some values to GPIO_PA registers. Finally we enable interrupt on button presses.

DAC

Next we set up the DAC. We need the high peripheral clock to drive this as well. So we write to to bit number 17 in CMU_HFPERCLKEN0. We want sound in both left and right speaker, so DAC0_CH0CTRL and DAC0_CH0CTRL is set to 1.

Timer

The timer is also driven by the high peripheral clock. Bit 6 is set to enable this. After this, TIMER1_TOP is set to 140. This will make the timer count to 140, and then restart at 0. We enable interrupt on this event.

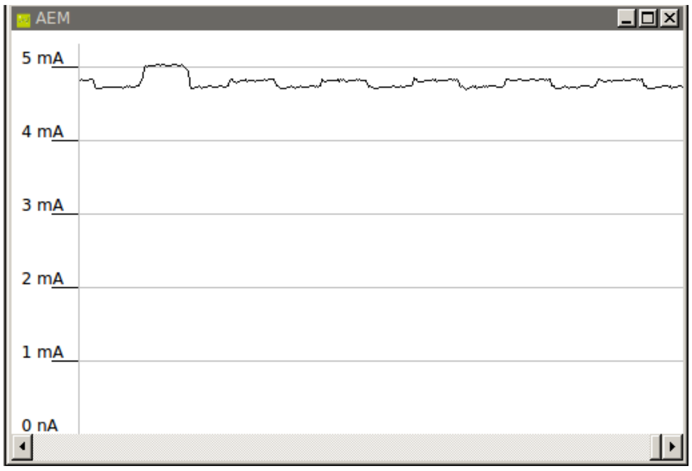

Figure 3.1: A JPEG image of a galaxy. Use vector graphics instead if you can.

3.2.2 Generating sound

Our next goal was to get a simple tone playing. As explained in section 2.1.1 sound is moving air. To get the 440Hz tone we wanted, we need to change the value in the DAC 880 times per second. This makes a square pulse with a period of 440Hz. To make this happen, we made use of the timer explained in section 3.2.1. We sat TIMER1_TOP to 140, as mentioned in 3.2.1, which gave us 100000 interrupts per second. This is because the clock that drives the timer runs at 14Mhz.

$$14000000/140 = 100000$$

In the interrupt handler we used two counters one to keep track of when to feed a new value to the DAC, and one for when to change the tone. We used a tone generator script written in python **??** to generate a header file with the values for each tone. These values indicates how many interrupts there should be between each time we feed a new value to the DAC.

3.2.3 Improvements

3.3 Testing

Add content in this section that describes how you tested and verified the correctness of your implementation, with respect to the requirements of the assignment.

4 Results

In this chapter, you should discuss the results you have obtained from your implementation. These can be correctness results, i.e whether the implementation behaved as expected, or numerical results that express runtime or energy measurements.

5 Conclusion

This chapter should be a look back at the entire report and summarizing the problem, the solution and the obtained results.

5.1 Evaluation of the Assignment

You can include comments about the assignment itself here. While this part is not obligatory and not graded, it is valuable feedback to the course staff that can be used to improve the exercises in the future.

A Appendices

1 Code

Listing 5.1: Python script for generating tones.h

```python
import math

def generate_tones():
  #Number of tones
  tones = [0]*70

  #Start tone (must be an octave of A)
  A1 = 55
  tones[0] = A1

  interrupts_per_sec = 100000

  #12th root of 2
  tone_increment = 1.05946309436

  for i in range(len(tones)-1):
    tones[i + 1] = tones[i]*tone_increment

  for i in range(len(tones)):
    tones[i] = int(interrupts_per_sec / tones[i])

  return tones


def generate_c_code(tones):

  tone_name_counter = 0
  octave = 1
  tone_names = ['A', 'BF', 'B', 'C', 'DF', 'D', 'EF', 'E', 'F', 'GF',
      'G', 'AF']
  for i in range(len(tones)):

    if len(tone_names) == tone_name_counter:
      tone_name_counter = 0
    if tone_names[tone_name_counter] == 'B':
      octave += 1
    tones[i] = "#define " + str(tone_names[tone_name_counter]) + str(
        octave) + " " + str(tones[i])
    tone_name_counter += 1

  return tones
```

```python


def main():
  a = generate_tones()
  a = generate_c_code(a)
  for line in a:
    print line

main()
```

Listing 5.2: The main file

```c
#include <stdint.h>
#include <stdbool.h>

#include "efm32gg.h"

/*
  TODO calculate the appropriate sample period for the sound wave(s)
  you want to generate. The core clock (which the timer clock is derived
  from) runs at 14 MHz by default. Also remember that the timer counter
  registers are 16 bits.
*/
/* The period between sound samples, in clock cycles */
#define   SAMPLE_PERIOD   140

/* Declaration of peripheral setup functions */
void setupTimer(uint16_t period);
void setupDAC();
void setupNVIC();
void setupGPIO();
void init();

/* Your code will start executing here */
int main(void)
{
  //init();
  /* Call the peripheral setup functions */
  setupGPIO();
  setupDAC();
  setupTimer(SAMPLE_PERIOD);

  /* Enable interrupt handling */
  setupNVIC();

  /* TODO for higher energy efficiency, sleep while waiting for interrupts
     instead of infinite loop for busy-waiting
  */
  while(1){
     __asm("wfi");
  }

  return 0;
}

void init(){
  //Shut down SRAM-blocks 1-3
  *EMU_MEMCTRL = 4;
}

void setupNVIC()
{

  *ISER0 |= 0x1802;


  /* TODO use the NVIC ISERx registers to enable handling of interrupt
     (s)
     remember two things are necessary for interrupt handling:
      - the peripheral must generate an interrupt signal
      - the NVIC must be configured to make the CPU handle the signal
     You will need TIMER1, GPIO odd and GPIO even interrupt handling
        for this
     assignment.
  */
}

/* if other interrupt handlers are needed, use the following names:
   NMI_Handler
   HardFault_Handler
   MemManage_Handler
   BusFault_Handler
   UsageFault_Handler
   Reserved7_Handler
   Reserved8_Handler
   Reserved9_Handler
   Reserved10_Handler
   SVC_Handler
   DebugMon_Handler
   Reserved13_Handler
   PendSV_Handler
   SysTick_Handler
   DMA_IRQHandler
   GPIO_EVEN_IRQHandler
   TIMER0_IRQHandler
   USART0_RX_IRQHandler
   USART0_TX_IRQHandler
   USB_IRQHandler
   ACMP0_IRQHandler
   ADC0_IRQHandler
   DAC0_IRQHandler
   I2C0_IRQHandler
   I2C1_IRQHandler
   GPIO_ODD_IRQHandler
```

```
    TIMER1_IRQHandler
    TIMER2_IRQHandler
    TIMER3_IRQHandler
    USART1_RX_IRQHandler
    USART1_TX_IRQHandler
    LESENSE_IRQHandler
    USART2_RX_IRQHandler
    USART2_TX_IRQHandler
    UART0_RX_IRQHandler
    UART0_TX_IRQHandler
    UART1_RX_IRQHandler
    UART1_TX_IRQHandler
    LEUART0_IRQHandler
    LEUART1_IRQHandler
    LETIMER0_IRQHandler
    PCNT0_IRQHandler
    PCNT1_IRQHandler
    PCNT2_IRQHandler
    RTC_IRQHandler
    BURTC_IRQHandler
    CMU_IRQHandler
    VCMP_IRQHandler
    LCD_IRQHandler
    MSC_IRQHandler
    AES_IRQHandler
    EBI_IRQHandler
    EMU_IRQHandler
*/
```

Listing 5.3: Python script for generating tones.h

```python
import math

def generate_tones():
  #Number of tones
  tones = [0]*70

  #Start tone (must be an octave of A)
  A1 = 55
  tones[0] = A1

  interrupts_per_sec = 100000

  #12th root of 2
  tone_increment = 1.05946309436

  for i in range(len(tones)-1):
    tones[i + 1] = tones[i]*tone_increment

  for i in range(len(tones)):
    tones[i] = int(interrupts_per_sec / tones[i])

  return tones
```

```python


def generate_c_code(tones):

  tone_name_counter = 0
  octave = 1
  tone_names = ['A', 'BF', 'B', 'C', 'DF', 'D', 'EF', 'E', 'F', 'GF', 'G', 'AF']
  for i in range(len(tones)):

    if len(tone_names) == tone_name_counter:
      tone_name_counter = 0
    if tone_names[tone_name_counter] == 'B':
      octave += 1
    tones[i] = "#define " + str(tone_names[tone_name_counter]) + str(octave) + " " + str(tones[i])
    tone_name_counter += 1

  return tones


def main():
  a = generate_tones()
  a = generate_c_code(a)
  for line in a:
    print line

main()
```

Listing 5.4: Python script for generating tones.h

```python
import math

def generate_tones():
  #Number of tones
  tones = [0]*70

  #Start tone (must be an octave of A)
  A1 = 55
  tones[0] = A1

  interrupts_per_sec = 100000

  #12th root of 2
  tone_increment = 1.05946309436

  for i in range(len(tones)-1):
    tones[i + 1] = tones[i]*tone_increment

  for i in range(len(tones)):
    tones[i] = int(interrupts_per_sec / tones[i])

  return tones
```

```python


def generate_c_code(tones):

  tone_name_counter = 0
  octave = 1
  tone_names = ['A', 'BF', 'B', 'C', 'DF', 'D', 'EF', 'E', 'F', 'GF', 'G', 'AF']
  for i in range(len(tones)):

    if len(tone_names) == tone_name_counter:
      tone_name_counter = 0
    if tone_names[tone_name_counter] == 'B':
      octave += 1
    tones[i] = "#define " + str(tone_names[tone_name_counter]) + str(octave) + " " + str(tones[i])
    tone_name_counter += 1

  return tones


def main():
  a = generate_tones()
  a = generate_c_code(a)
  for line in a:
    print line

main()
```

Listing 5.5: Python script for generating tones.h

```python
import math

def generate_tones():
  #Number of tones
  tones = [0]*70

  #Start tone (must be an octave of A)
  A1 = 55
  tones[0] = A1

  interrupts_per_sec = 100000

  #12th root of 2
  tone_increment = 1.05946309436

  for i in range(len(tones)-1):
    tones[i + 1] = tones[i]*tone_increment

  for i in range(len(tones)):
    tones[i] = int(interrupts_per_sec / tones[i])

  return tones
```

```python


def generate_c_code(tones):

  tone_name_counter = 0
  octave = 1
  tone_names = ['A', 'BF', 'B', 'C', 'DF', 'D', 'EF', 'E', 'F', 'GF', 'G', 'AF']
  for i in range(len(tones)):

    if len(tone_names) == tone_name_counter:
      tone_name_counter = 0
    if tone_names[tone_name_counter] == 'B':
      octave += 1
    tones[i] = "#define " + str(tone_names[tone_name_counter]) + str(octave) + " " + str(tones[i])
    tone_name_counter += 1

  return tones


def main():
  a = generate_tones()
  a = generate_c_code(a)
  for line in a:
    print line

main()
```