



**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<sub>px</sub>CTRL-PortControlRegister* 11.5.3 *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 ???. 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.

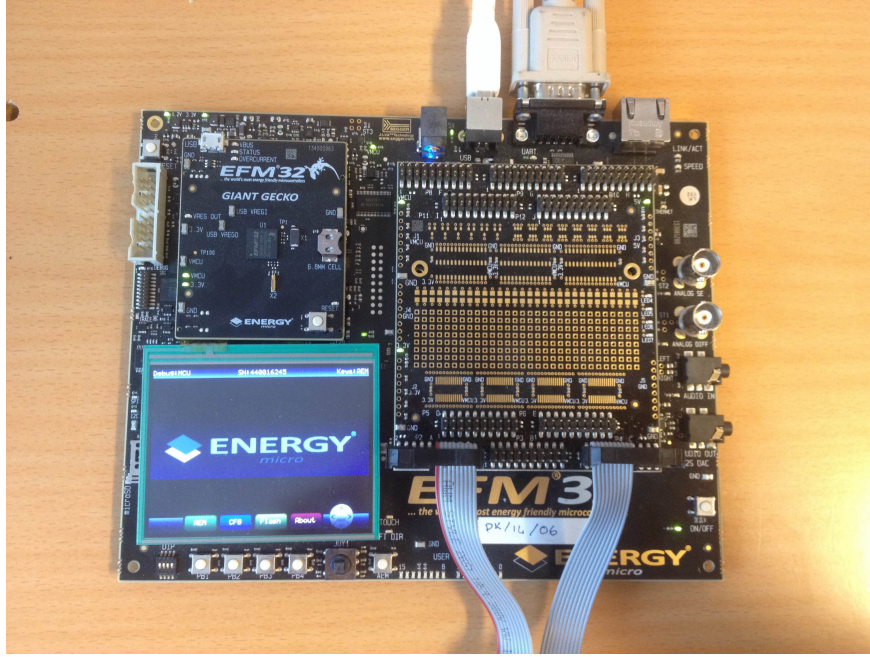


Figure 2.1: EFM32GG-DK3750

### 2.1.3 Gamepad

In addition we had a gamepad with eight buttons and eight LEDs depicted in Figure ???. 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 from 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:

$$P_{tot} = P_{dynamic} + P_{static} = \alpha CV^2 f + I_{static} V \quad (2.1)$$

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:

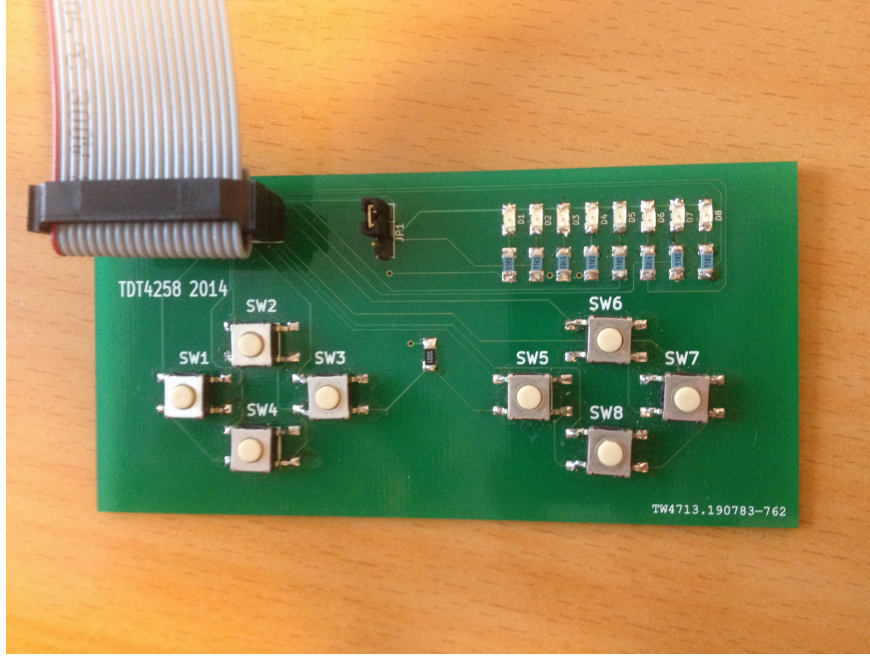


Figure 2.2: Gamepad

$$E = P_{total}t \quad (2.2)$$

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 ?? shows an overview of the EFM32GG and Figure ?? shows a color scheme with the different energy modes indicated with a respective number. The colors in Figure ?? 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.

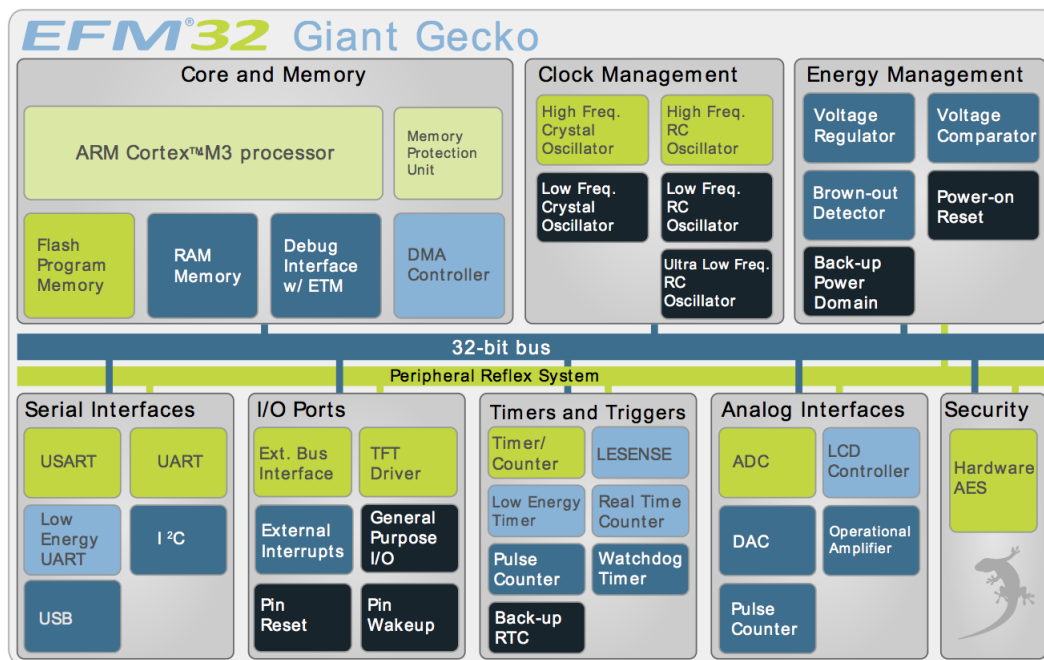


Figure 2.3: Diagram of EFM32GG

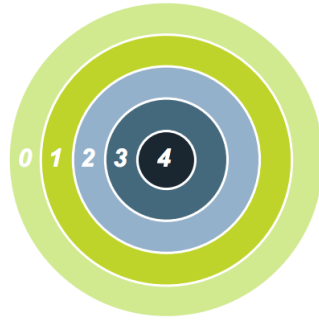


Figure 2.4: Energy Mode Indicator

### 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 ??, 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 ??.

Listing 3.1: Project structure

```
1 |-- buttons.h
2 |-- core
3 |-- dac.c
4 |-- efm32gg.h
5 |-- ex2.bin
6 |-- ex2.c
7 |-- gpio.c
8 |-- interrupt_handlers.c
9 |-- lib
10 |   |-- efm32gg.ld
11 |   |-- ex2.bin
12 |   |-- libefm32gg.a
13 |-- Makefile
14 |-- songs.h
15 |-- timer.c
16 |-- tone-generator.py
17 |-- 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 ??. `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\_handlers.c contains the code for handling interrupts*

### 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.

### 3.2.2 Generating sound

Our next goal was to get a simple tone playing. As explained in section ?? 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 ?. We sat `TIMER1_TOP` to 140, which gave us 100000 interrupts per second. In the interrupt handler we used two

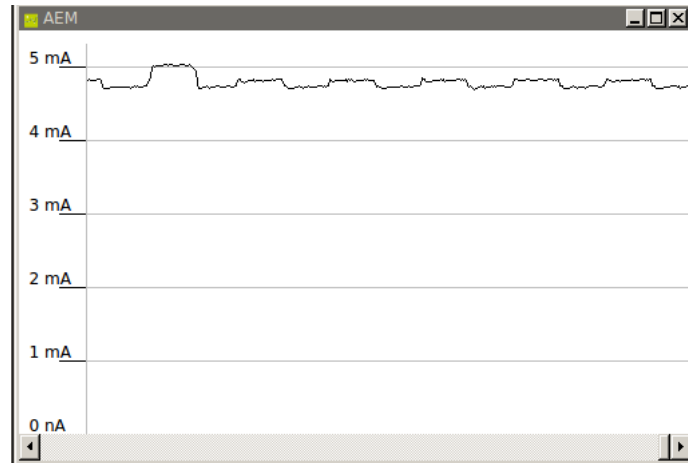


Figure 3.1: A JPEG image of a galaxy. Use vector graphics instead if you can.

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

```
1 import math
2
3 def generate_tones():
4     #Number of tones
5     tones = [0]*70
6
7     #Start tone (must be an octave of A)
8     A1 = 55
9     tones[0] = A1
10
11     interrupts_per_sec = 100000
12
13     #12th root of 2
14     tone_increment = 1.05946309436
15
16     for i in range(len(tones)-1):
17         tones[i + 1] = tones[i]*tone_increment
18
19     for i in range(len(tones)):
20         tones[i] = int(interrupts_per_sec / tones[i])
21
22     return tones
23
24
25 def generate_c_code(tones):
26
27     tone_name_counter = 0
28     octave = 1
29     tone_names = ['A', 'BF', 'B', 'C', 'DF', 'D', 'EF', 'E', 'F', 'GF',
30                  'G', 'AF']
31
32     for i in range(len(tones)):
33
34         if len(tone_names) == tone_name_counter:
35             tone_name_counter = 0
36             if tone_names[tone_name_counter] == 'B':
37                 octave += 1
38             tones[i] = "#define " + str(tone_names[tone_name_counter]) + str(
39                 octave) + " " + str(tones[i])
40             tone_name_counter += 1
41
42     return tones
```

```

40
41
42 def main():
43     a = generate_tones()
44     a = generate_c_code(a)
45     for line in a:
46         print line
47
48 main()

```

Listing 5.2: The main file

```

1 #include <stdint.h>
2 #include <stdbool.h>
3
4 #include "efm32gg.h"
5
6 /*
7  TODO calculate the appropriate sample period for the sound wave(s)
8  you want to generate. The core clock (which the timer clock is
9  derived
10 from) runs at 14 MHz by default. Also remember that the timer
11 counter
12 registers are 16 bits.
13 */
14 /* The period between sound samples, in clock cycles */
15 #define SAMPLE_PERIOD 140
16
17 /* Declaration of peripheral setup functions */
18 void setupTimer(uint16_t period);
19 void setupDAC();
20 void setupNVIC();
21 void setupGPIO();
22 void init();
23
24 /* Your code will start executing here */
25 int main(void)
26 {
27     //init();
28     /* Call the peripheral setup functions */
29     setupGPIO();
30     setupDAC();
31     setupTimer(SAMPLE_PERIOD);
32
33     /* Enable interrupt handling */
34     setupNVIC();
35
36     /* TODO for higher energy efficiency, sleep while waiting for
37     interrupts
38     instead of infinite loop for busy-waiting
39     */
40     while(1){
41         __asm("wfi");

```

```

39  }
40
41  return 0;
42 }
43
44 void init(){
45     //Shut down SRAM-blocks 1-3
46     *EMU_MEMCTRL = 4;
47 }
48
49 void setupNVIC()
50 {
51
52     *ISERO |= 0x1802;
53
54
55     /* TODO use the NVIC ISERx registers to enable handling of interrupt
56        (s)
57        remember two things are necessary for interrupt handling:
58        - the peripheral must generate an interrupt signal
59        - the NVIC must be configured to make the CPU handle the signal
60        You will need TIMER1, GPIO odd and GPIO even interrupt handling
61        for this
62        assignment.
63        */
64 }
65
66 /* if other interrupt handlers are needed, use the following names:
67    NMI_Handler
68    HardFault_Handler
69    MemManage_Handler
70    BusFault_Handler
71    UsageFault_Handler
72    Reserved7_Handler
73    Reserved8_Handler
74    Reserved9_Handler
75    Reserved10_Handler
76    SVC_Handler
77    DebugMon_Handler
78    Reserved13_Handler
79    PendSV_Handler
80    SysTick_Handler
81    DMA_IRQHandler
82    GPIO_EVEN_IRQHandler
83    TIMERO_IRQHandler
84    USARTO_RX_IRQHandler
85    USARTO_TX_IRQHandler
86    USB_IRQHandler
87    ACMPO_IRQHandler
88    ADCO_IRQHandler
89    DACO_IRQHandler
90    I2CO_IRQHandler
91    I2C1_IRQHandler
92    GPIO_ODD_IRQHandler

```



```

91  TIMER1_IRQHandler
92  TIMER2_IRQHandler
93  TIMER3_IRQHandler
94  USART1_RX_IRQHandler
95  USART1_TX_IRQHandler
96  LESENSE_IRQHandler
97  USART2_RX_IRQHandler
98  USART2_TX_IRQHandler
99  UART0_RX_IRQHandler
100 UART0_TX_IRQHandler
101 UART1_RX_IRQHandler
102 UART1_TX_IRQHandler
103 LEUART0_IRQHandler
104 LEUART1_IRQHandler
105 LETIMERO_IRQHandler
106 PCNT0_IRQHandler
107 PCNT1_IRQHandler
108 PCNT2_IRQHandler
109 RTC_IRQHandler
110 BURTC_IRQHandler
111 CMU_IRQHandler
112 VCMP_IRQHandler
113 LCD_IRQHandler
114 MSC_IRQHandler
115 AES_IRQHandler
116 EBI_IRQHandler
117 EMU_IRQHandler
118 */

```

Listing 5.3: Python script for generating tones.h

```

1  import math
2
3  def generate_tones():
4      #Number of tones
5      tones = [0]*70
6
7      #Start tone (must be an octave of A)
8      A1 = 55
9      tones[0] = A1
10
11     interrupts_per_sec = 100000
12
13     #12th root of 2
14     tone_increment = 1.05946309436
15
16     for i in range(len(tones)-1):
17         tones[i + 1] = tones[i]*tone_increment
18
19     for i in range(len(tones)):
20         tones[i] = int(interrupts_per_sec / tones[i])
21
22     return tones

```

```

23
24
25 def generate_c_code(tones):
26
27     tone_name_counter = 0
28     octave = 1
29     tone_names = ['A', 'BF', 'B', 'C', 'DF', 'D', 'EF', 'E', 'F', 'GF',
30                  'G', 'AF']
31     for i in range(len(tones)):
32         if len(tone_names) == tone_name_counter:
33             tone_name_counter = 0
34             if tone_names[tone_name_counter] == 'B':
35                 octave += 1
36             tones[i] = "#define " + str(tone_names[tone_name_counter]) + str(
37                 octave) + " " + str(tones[i])
38             tone_name_counter += 1
39     return tones
40
41
42 def main():
43     a = generate_tones()
44     a = generate_c_code(a)
45     for line in a:
46         print line
47
48 main()

```

Listing 5.4: Python script for generating tones.h

```

1 import math
2
3 def generate_tones():
4     #Number of tones
5     tones = [0]*70
6
7     #Start tone (must be an octave of A)
8     A1 = 55
9     tones[0] = A1
10
11     interrupts_per_sec = 100000
12
13     #12th root of 2
14     tone_increment = 1.05946309436
15
16     for i in range(len(tones)-1):
17         tones[i + 1] = tones[i]*tone_increment
18
19     for i in range(len(tones)):
20         tones[i] = int(interrupts_per_sec / tones[i])
21
22     return tones

```

```

23
24
25 def generate_c_code(tones):
26
27     tone_name_counter = 0
28     octave = 1
29     tone_names = ['A', 'BF', 'B', 'C', 'DF', 'D', 'EF', 'E', 'F', 'GF',
30                  'G', 'AF']
31     for i in range(len(tones)):
32         if len(tone_names) == tone_name_counter:
33             tone_name_counter = 0
34             if tone_names[tone_name_counter] == 'B':
35                 octave += 1
36             tones[i] = "#define " + str(tone_names[tone_name_counter]) + str(
37                 octave) + " " + str(tones[i])
38             tone_name_counter += 1
39     return tones
40
41
42 def main():
43     a = generate_tones()
44     a = generate_c_code(a)
45     for line in a:
46         print line
47
48 main()

```

Listing 5.5: Python script for generating tones.h

```

1 import math
2
3 def generate_tones():
4     #Number of tones
5     tones = [0]*70
6
7     #Start tone (must be an octave of A)
8     A1 = 55
9     tones[0] = A1
10
11     interrupts_per_sec = 100000
12
13     #12th root of 2
14     tone_increment = 1.05946309436
15
16     for i in range(len(tones)-1):
17         tones[i + 1] = tones[i]*tone_increment
18
19     for i in range(len(tones)):
20         tones[i] = int(interrupts_per_sec / tones[i])
21
22     return tones

```

```

23
24
25 def generate_c_code(tones):
26
27     tone_name_counter = 0
28     octave = 1
29     tone_names = ['A', 'BF', 'B', 'C', 'DF', 'D', 'EF', 'E', 'F', 'GF',
30                  'G', 'AF']
31     for i in range(len(tones)):
32         if len(tone_names) == tone_name_counter:
33             tone_name_counter = 0
34             if tone_names[tone_name_counter] == 'B':
35                 octave += 1
36             tones[i] = "#define " + str(tone_names[tone_name_counter]) + str(
37                 octave) + " " + str(tones[i])
38             tone_name_counter += 1
39     return tones
40
41
42 def main():
43     a = generate_tones()
44     a = generate_c_code(a)
45     for line in a:
46         print line
47
48 main()

```